Learning to Edit Code :

Towards Building General Purpose Models for Source Code Editing

Saikat Chakraborty

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2022

Abstract

Learning to Edit Code :

Towards Building General Purpose Models for Source Code Editing

Saikat Chakraborty

The way software developers edit code day-to-day tends to be repetitive, often using existing code elements. Many researchers have tried to automate the repetitive code editing process by mining specific change templates. However, such templates are often manually implemented for automated applications. Consequently, such template-based automated code editing is very tedious to implement. In addition, template based code editing is often narrowly-scoped and low noise-tolerant. Machine Learning, specially deep learning-based techniques, could help us solve these problems because of their generalization and noise tolerance capacities.

The advancement of deep neural networks and the availability of vast open-source evolutionary data opens up the possibility of automatically learning those templates from the wild and applying those in the appropriate context. However, deep neural network-based modeling for code changes, and code, in general, introduces some specific problems that need specific attention from the research community. For instance, source code exhibit strictly defined syntax and semantics inherited from the properties of Programming Language (PL). In addition, source code vocabulary (possible number of tokens) can be arbitrarily large.

This dissertation formulates the problem of automated code editing as a multi-modal translation problem, where, given a piece of code, the context, and some guidance, the objective is to generate edited code. In particular, we divide the problem into two sub-problems — source code understanding and generation. We empirically show that the deep neural networks (models in general) for these problems should be aware of the PL-properties (*i.e.*, syntax, semantics). This dissertation investigates two primary directions of endowing the models with knowledge about PL-properties — (i) explicit encoding: where we design models catering to a specific property, and (ii) implicit encoding: where we train a very-large model to learn these properties from very large corpus of source code in unsupervised ways.

With implicit encoding, we custom design the model to cater to the need for that property. As an example of such models, we developed CODIT — a tree-based neural model for syntactic correctness. We design CODIT based on the Context Free Grammar of the programming language. Instead of generating source code, CODIT first generates the tree structure by sampling the production rule from the CFG. Such a mechanism prohibits infeasible production rule selection. In the later stage, CODIT generates the edited code conditioned on the tree generated earlier. Such conditioning makes the edited code syntactically correct. CODIT showed promise in learning code edit patterns in the wild and effectiveness in automatic program repair. In another empirical study, we showed that a graph-based model is better suitable for source code understanding tasks such as vulnerability detection.

On the other hand, with implicit encoding, we use a very large (with several hundred million parameters) yet generic model. However, we pre-train these models on a super-large (usually hundreds of gigabytes) collection of source code and code metadata. We empirically show that if sufficiently pre-trained, such models are capable enough to learn PL properties such as syntax and semantics. In this dissertation, we developed two such pre-trained models, with two different learning objectives. First, we developed PLBART— the first-ever pre-trained encoder-decoder-based model for source code and show that such pre-train enables the model to generate syntactically and semantically correct code. Further, we show an in-depth empirical study on using PLBART in automated code editing. Finally, we develop another pre-trained model — NatGen to encode the natural coding convention followed by developers into the model. To design NatGen, we first deliberately modify the code from the developers' written version preserving the original se-

mantics. We call such transformations ‘de-naturalizing’ transformations. Following the previous studies on induced unnaturalness in code, we defined several such ‘de-naturalizing’ transformations and applied those to developer-written code. We pre-train NatGen to reverse the effect of these transformations. That way, NatGen learns to generate code similar to the developers’ written by undoing any unnaturalness induced by our forceful ‘de-naturalizing‘ transformations. NatGen has performed well in code editing and other source code generation tasks.

The models and empirical studies we performed while writing this dissertation go beyond the scope of automated code editing and are applicable to other software engineering automation problems such as Code translation, Code summarization, Code generation, Vulnerability detection, Clone detection, etc. Thus, we believe this dissertation will influence and contribute to the advancement of AI4SE and PLP.

Table of Contents

Acknowledgments xii

Dedication xv

Chapter 1: Introduction 1

1.1 Problem Formulation and Challenges 3

1.2 Solution Approach and Research Contribution 6

1.2.1 Explicit Encoding of PL Constructs 6

1.2.2 Implicit Encoding of PL Constructs 8

Chapter 2: Background and Related Work 11

2.1 Background 11

2.1.1 Modeling Code Changes. 11

2.1.2 Grammar-based modeling. 11

2.1.3 Neural Machine Translation 12

2.1.4 Transformer Model for Sequence Processing 12

2.1.5 Transfer Learning for Source Code 13

2.2 Related Works 14

I Explicit Encoding 19

Chapter 3: Code Editing with Tree-Based Neural Models 20
3.1 Motivation 20
3.2 Methodology 23
3.2.1 Tree Translation Model 24
3.2.2 Token Generation Model 26
3.2.3 Implementation 27
3.3 Experimental Design 29
3.3.1 Evaluation Metric 30
3.3.2 Baseline 31
3.4 Research Findings 32
3.4.1 Discussion & Threats to validity 45
3.5 Concluding Remarks 47

Chapter 4: Program Understanding through Explicit Program Encoding 48
4.1 Motivation 48
4.2 Methodology 51
4.2.1 DL-based Vulnerability Detection 51
4.2.2 Graph Based Representation Learning 52
4.3 Experimental Design 57
4.4 Research Findings 59
4.5 Concluding Remarks 64

II Implicit Encoding 65

Chapter 5: Unified Pre-training for Program Understanding and Generation 66

5.1 Motivation 66

5.2 Methodology 69

5.2.1 Denoising Pre-training 69

5.2.2 Fine-tuning PLBART 72

5.3 Experiment Design 73

5.3.1 Evaluation Tasks 74

5.3.2 Evaluation Metrics 75

5.3.3 Baseline Methods 76

5.4 Research Findings 77

5.5 Concluding Remarks 84

Chapter 6: On Multi-Modal Learning of Editing Source Code 86

6.1 Motivation 86

6.2 Methodology 89

6.2.1 Pre-processing 90

6.2.2 Encoder-Decoder Model 91

6.2.3 Output Generation 92

6.3 Experimental Design 92

6.3.1 Dataset 92

6.3.2 Data Preparation 93

6.3.3 Training 94

6.3.4 Evaluation Metric 94

6.4 Research Findings 94

6.4.1 Discussion & Threats to Validity 108

6.5 Concluding Remarks 110

Chapter 7: Generative pre-training by “Naturalizing” source code 111

7.1 Motivation 111

7.2 Background & Problem Formulation 114

7.2.1 The Dual Channels of Code 114

7.2.2 “Naturalizing" *vs.* De-noising 115

7.2.3 Research Questions 116

7.3 Methodology 117

7.3.1 De-Naturalizing Source Code 117

7.3.2 Pre-training 123

7.3.3 Fine-tuning 123

7.4 Experimental Design 124

7.5 Research Findings 126

7.5.1 NetGen’s Effectiveness on pre-training 126

7.5.2 Pre-training progression for all metrics 133

7.5.3 NetGen’s Effectiveness on Fine-tuning Tasks 134

7.5.4 Limitations & Threats 141

7.6 Concluding Remarks 142

Chapter 8: Conclusion 144

8.1 Dissertation Summary 144

8.2 Future Work 146

References 148

List of Figures

1.1 An example of code editing problem 3

1.2 Code example showing long range dependency between code components 4

1.3 Examples of incorrect code 5

1.4 Conceptual breakdown of different knowledge in a code editor model 8

3.1 Illustrative Example showing CODIT's working procedure. 21

3.2 Examples of different types of concrete patches generated by CODIT 34

3.3 Examples of CODIT's ability to generalize in different use cases. `Exception`, `Error`, `RuntimeException` are modified to `EOFException` under different context. 35

3.4 Performance of CODIT @top-5 ($K_{token} = 5$) for different K_{tree} 39

3.5 Percentage of correct prediction by CODIT with respect to number of nodes in the tree. . 40

3.6 Patches passing the bug-triggering tests *v.s.* time. 43

4.1 Example of CWE-761 [50]. A buffer is freed not at the start of the buffer but somewhere in the middle of the buffer. This can cause the application to crash, or in some cases, modify critical program variables or execute code. This vulnerability can be detected with data dependency. 49

4.2 Overview of the ReVeal vulnerability prediction framework. 52

4.3 Contribution of different code component in correct classification of vulnerability by different model. Red-shaded code elements are most contributing, Green-shaded are the least. **Red** colored code are the source of vulnerabilities. 61

4.4 Integer Overflow vulnerability (CWE-190). 62

4.5 Uncontrolled Memory vulnerability (CWE-789). 62

5.1 Example motivating the need to understand the association of program and natural languages for code summarization, generation, and translation. 67

6.1 Example of an identical code (marked in red) changed in two different ways (**green** and blue) in two different contexts, where both can be correct patches. However, based on developers' guidance (top line) to fix a list related problem, **green** is the correct patch in this context. 86

6.2 A motivating example. The guidance provides a brief summary of what needs to be changes. The underlined tokens are directly copied from guidance and context into the patched code. 87

6.3 Overview of MODIT pipeline 89

6.4 Schematic diagram of the three types of pre-trained models. used to evaluate MODIT. . . 96

6.5 Example patch where MODIT was able to generate correct patch, but CodeGPT could not. MODIT's patch is shown in **green**, and CodeGPT generated patch is shown in blue. 98

6.6 Correctly predicted patch by MODIT. CodeBERT could not understand and reason about the textual hint to predict the correct patch. 99

6.7 Example showing the effect of textual guidance in MODIT's performance. MODIT produced the **correct patch** with guidance, without guidance as input MODIT's produced **patch** is essentially refactored version of original input. 102

6.8 Example showing the necessity of context information in predicting the correct patch. MODIT's generated **correct patch** with the context as input. Without context, MODIT received `sendMessage(m)` and the guidance as input, did not know the variable `sent` could be the parameter of the function `sendMessage`, and predicted a wrong patch. . . . 102

6.9 An alternative architecture of code editing with multi-encoder model. We initialize each of the encoders with pre-trained Encoder model. 104

6.10 Input token representation generation in single encoder and multiple encoder. 106

7.1 Example of a natural code fragment written by developers and its ‘un-naturally’ transformed counterpart. If the `initialization` and `update` part of the `for` loop were to left empty, developers would write the `while` loop. 113

7.2 Semantic preserving transformation used to prepare the pre-training data for NetGen (Part-1). 118

7.3 Semantic preserving transformation used to prepare the pre-training data for NetGen (Part-2). 119

7.4 “De-Naturalization” workflow in NetGen. 122

7.5 Example of input and generated code by different pretrained models. 127

7.6 Example of input generated code by different pre-trained models. 128

7.7 Example of input generated code by different pre-trained models (contd.) 129

7.8 Example of input generated code by different pre-trained models (contd.) 130

7.9 Example of input generated code by different pre-trained models (contd.) 131

7.10 Performance of NetGen pre-trained model under different code transformations. 132

7.11 Progression of Different metrics of different language in Validation dataset over number pre-training steps. 133

7.12 Zero-shot transfer learning capability of NetGen in for different tasks. 137

7.13 Few shot Learning evaluation of NetGen. In each case, the pre-trained model is fine-tuned on 200 training examples for 10 epoch and the result is on the full test set. 138

7.14 NetGen’s results on different tasks with Few shot settings. X-axis shows number of training examples. 139

List of Tables

3.1 Summary of datasets used to evaluate CODIT. 29

3.2 Performance of CODIT suggesting concrete patches. For Token Based models, predominant source of information in the code are token sequences. For Tree Based models information source is code AST. For IR based method, information retrieval model is used on code. 33

3.3 Correct patches generated by the standalone token generation model when the true tree structure is known. 37

3.4 $\mathscr{M}_{tree}$ top-5 performance for different settings. 38

3.5 CODIT performance *w.r.t.* to the attention based copy mechanism @top-5 (K_{tree} =2, K_{token} =5). Lower bound is without copy. The upper bound evaluates with oracle copying predictions for `<unknown>`. For CODIT each `<unknown>` token is replaced by the source token with the highest attention. 39

3.6 Cross project generalization test for CODIT (% of correct at @top-5). 41

3.7 CODIT's performance on fixing Defects4J [104] bugs. 44

4.1 Summary Statistics of the ReVeal dataset. '25 Q.' and '75 Q.' represents 25th quantile and 75th quantile, respectively. All the statistics are on function granularity. 57

4.2 Performance of different models in Vulnerability Detection. All the numbers are reported as *Median (IQR)* format. 59

4.3 Impact of GGNN in ReVeal's performance [Median (IQR)]. 60

5.1 Statistics of the data used to pre-train PLBART. "Nb of documents" refers to the number of functions in Java and Python collected from Github and the number of posts (questions and answers) in the natural language (English) from StackOverflow. 69

5.2 Example encoder inputs and decoder outputs during denoising pre-training of PLBART. We use three noising strategies: token masking, token deletion, and token infilling (shown in the three examples, respectively). 69

5.3 Example inputs to the encoder and decoder for fine-tuning PLBART on sequence generation tasks: source code summarization (S), generation (G), and translation (T). 70

5.4 Details of the hyper-parameters used during fine-tuning for sequence generation tasks. * indicates pre-trained from scratch using source code-text pairs. 73

5.5 Statistics of the downstream benchmark datasets. 74

5.6 Results on source code summarization, evaluated with smoothed BLEU-4 score. The baseline results are reported from Feng *et al.* [63]. 78

5.7 Results on the vulnerable code detection (accuracy) and clone detection (F1 score) tasks. . 78

5.8 Results on text-to-code generation task using the CONCODE dataset [95]. 80

5.9 An example of generated code by PLBART that is syntactically and semantically valid, but does not match the reference. 80

5.10 Results on source code translation using Java and C# language dataset introduced in [142]. PBSMT refers to phrase-based statistical machine translation where the default settings of Moses decoder [119] is used. The training data is tokenized using the RoBERTa [138] tokenizer. 81

5.11 Example C# code generated by PLBARTthat does not exactly match the reference code. . 82

5.12 Results on program repair (Abstract Code). 83

5.13 Evaluation of PLBARTin generating concrete edits. 84

6.1 Statistics of the datasets studied. 93

6.2 Top-1 accuracies of different models *w.r.t.* their training type, model sizes, input modality. . 97

6.3 Contribution of different input modalities in MODIT's performance. ✓ indicates that corresponding input modality is used as encoder input, ✗ indicates otherwise. We report top-1 accuracy as performance measure. Exp. ID is used later to refer to corresponding experiment result. Exp. ID Φ_* denotes an experiment with $*$ as input modalities. 101

6.4 Comparison of multi encoder model. 105

6.5 Performance of MODIT when the input in the full code and the output is patched full code. 107

6.6 Comparison between concrete tokenization and abstract tokenization alongside pre-trained models. Results are shown as top-1 accuracy of full code generation in $B2F_s$/ $B2F_m$ datasets. 108

7.1 Statistics of fine-tuning datasets. 124

7.2 Evaluation of NetGen for code generation task. CS is the percentage of examples where output is directly copied from source, and ED is the median edit distance between input code and output code. 126

7.3 Results of Text to Code Generation. '-' implies that those results are not reported by corresponding approaches. $\mathcal{M}_{last}$ is the model after completing the fintuning, and $\mathcal{M}_{best}$ is the intermediate model with best validation performance. 134

7.4 Code Translation results. '-' implies that those results are not reported by corresponding approaches. 135

7.5 One sided Wilcoxon sign rank test statistics result comparison between NetGen and CodeT5 across different tasks and metrics. Highlighted results are statistically significant with >95% confidence. 136

7.6 Result of Bug fix (Top 1 fix accuracy). 136

Acknowledgements

When writing this dissertation, I have been reflecting on the contribution of many influential people in my life. This dissertation wouldn't have been possible without the unwavering support, advice, suggestions, and sacrifices from family, friends, advisors, mentors, colleagues, and collaborators. My heartfelt gratitude is due to my whole support system.

I begin, rightfully, with my advisor, Professor Baishakhi Ray. Without her support, I wouldn't be the researcher and person I am today. She helped me develop my own research taste and philosophy. With her guidance, I am not (anymore) afraid to embark on chasing new research challenges. I still remember the day when I first talked to her over skype, and she introduced the "Naturalness Conjecture" ([89, 190]) to me; I was so fascinated. Little did I know back then that the whole premise of my Ph.D. dissertation would be this conjecture. After I joined her lab, we started diving deep into the area of Programming Language Processing and discovered many exciting things together. She always inspired me to think deeply. She never stopped believing in me, even when I seriously doubted myself. Thanks, Baishakhi, for all you taught me and for all the benevolence and support.

My family always stood by me, supporting my decisions throughout. My parents sacrificed a lot for me. They had to endure much pain and suffering just to support me. My parents are always just one call away every time I need them. I owe my parents everything, perhaps more. My father, being an engineer himself, bestowed the engineering and scientific mindset upon me. My mother, on the other hand, introduced me to philosophy and rational thinking in my childhood from her background in liberal arts. Working towards earning "Doctor of Philosophy", I attempted to reach the crossroad between the teachings of both my parents. I hope I made them proud. I also like to acknowledge my younger sister. Despite being several years younger, she has always been generous to me, like an elder sister. My wife and my in-laws have always been there for me. My wife supported me through the critical times during my Ph.D. Without her empathy and counsel, there was a fair chance of me being derailed. She always makes me smile even during the time of great stress. I also must express my gratitude to my uncle, aunt, and cousins for their compassion.

I am very fortunate to have Professor Kai-Wei Chang, Professor Premkumar Devanbu, and Professor Gail Kaiser as my mentors and collaborators during the Ph.D. journey. Part of this dissertation is a result of my successful collaboration with Professor Chang (§5). Collaboration with him helped shape my thoughts from a Machine Learning and Language Processing perspective. I collaborated with Professor Devanbu while developing NatGen (§7), where he shepherded the core research idea to present a pre-trained model for Software Engineers. I am thankful to Professor Kaiser for her valuable feedback while developing ReVeal (§4). Throughout my Ph.D. studentship at Columbia, I would go to Professor Kaiser to solve any and every academic problem. Her prompt responses and suggestions keeping my best interest in mind, made my journey here a bit easier. I am also thankful to Professor Kathleen McKeown and Professor Junfeng Yang for their valuable feedback when I proposed this dissertation and guidance towards making this dissertation better. I express my heartfelt gratitude to all my mentors and teachers for their generosity, guidance, and succor.

I want to convey my thankfulness to my mentors, collaborators, and (I am fortunate to call them) friends — Dr. Miltos Allamanis, Dr. Rahul Krishna, Dr. Wasi Ahmad, Dr. Ripon Saha, Yangruibo(Robin) Ding, and Toufique Ahmed for extending their helping hand while developing this dissertation. I collaborated with Miltos in my first project (§3) as a Ph.D. student, where he helped me navigate the most amazing world of Machine Learning-based Program Analysis. Rahul and I collaborated on developing ReVeal (§4). I really learned how to design experiments to test every aspect of a hypothesis and how to articulate thoughts in a smooth, fluid motion. Ripon was my mentor during my first internship at FLA and has been my mentor ever since. There isn't any issue (personal and professional) for which I cannot seek his advice. Thanks, Ripon, for your generous and selfless mentorship. Wasi and I worked on several projects [4, 175] and published a crucial part of this dissertation (§5). Research synergy between Robin and me led to successful collaborations on several projects as part of this dissertation (§3, §4, §7), and beyond this thesis [57]. Robin and I often talk about research ideas casually while sitting in the office. Sometime some crazy ideas flow into witty jokes; (most importantly) sometimes such ideas flow into being successful publications (§6 [39], §7 [36]). Thanks, Robin, for the way you helped me be a better researcher, (above all) for your kind friendship. Toufique (who has been my friend since my undergraduate days) and I collaborated on the final chapter of this thesis (§7). Thanks, everyone; your thought processes definitely influenced my research philosophy, and I learned much from you.

My friends are a crucial part of my support system. They always make me a better version of myself with prodigal support, fair criticism, and above all, stellar munificence. I specifically would like to mention Himel Dev, Madhusudan Basak, Azad Salam, Probal Dhar, Tanmoy Sen, Kamrul Hasan, Arpan Paul, Deep Chakraborty, Asif Alavee, Santunu Purohit, Jui Dutta, Nayan Mallick, Anthony Saieva. Thanks, friends, for being there for me. There are so many people I want to thank for their small yet meaningful contribution in my personal, professional, and academic growth and maturity. I could write long articles acknowledging their contribution. Nevertheless, we should start talking about Automated Code Editing. Meanwhile, thanks everyone for the generosity.

Dedication

To my beloved mother, for all her sacrifices, and all she endured to bring me at this position.

Chapter 1: Introduction

The last few years have seen a lot of progress in bringing automation into the software development process [24, 10, 219]. Availability of large quantities of software life-cycle data in vast open source repository platforms (*e.g.,* GitHub, BitBucket), developer forums (*e.g.,* StackOverflow), and issue trackers (*e.g.,* Jira, Bugzilla) paves the way to automate different development tasks. Artificial Intelligence for Software Engineering (AI4SE) leverages these extensive data sources to build automation tools for Software Engineers [51, 238, 83]. Previous researches had shown that source code artifacts (*e.g.,* source code [89], code changes [169, 166, 192], bugs [190]) are repetitive, *i.e.*, follow similar patterns across different developers, repositories, and organizations. Such repetition opens up the possibility of automating developers' coding activities, offloading some of the repetitive and tedious tasks to machines, and leaving time for the developers to concentrate on more creative tasks.

To this end, *Programming Language Processing (PLP)* [239, 159, 160], an emerging research field, aims at developing automation tools for these repetitive activities, leveraging techniques from Language Processing, Machine Learning (ML), and Deep Learning (DL). In particular, PLP aims to develop techniques specially catered to Programming Languages and bring automation in software engineers' day-to-day programming activities. In this dissertation, we are aiming to build tools towards automating one such repetitive [193, 200] programming activity — Code Editing.

Developers edit source code to add new features, fix bugs, or maintain existing functionality (*e.g.,* API updates, refactoring, *etc.*) all the time. Recent research has shown that these edits are often repetitive [169, 166]. Moreover, the code components (*e.g.,* token, sub-trees, *etc.*) used to build the edits are often taken from the existing codebase [149, 25]. However, manually applying such repetitive edits can be tedious and error-prone [191]. Thus, it is important to automate code changes, as much as possible, to reduce the developers' burden. There is significant industrial and academic work on automating code changes. For example, modern IDEs support specific types of automatic changes (*e.g.,* refactoring, adding boiler-plate code [157, 66], *etc*). Many research tools aim to automate some types of edits, *e.g.,* API related changes [167, 218, 195, 19, 172, 165], refactoring [65, 194, 69, 152], frequently undergone code changes related to Pull Requests [223], *etc*. Researchers have also proposed automating generic changes by learning either from example edits [154, 200] or similar patches applied previously to source code [153, 166, 192, 223]. While the above lines of work are promising and have shown initial success, they either rely on predefined change templates or require domain-specific knowledge about the type of changes: both are hard to generalize to the larger context. However, all of them leverage, in some way, common edit patterns. Given that a large amount of code and its change history, associated high-level comments (*e.g.,* commit messages) from developers is available, thanks to software forges like GitHub, Bitbucket, *etc*., a natural question arises: *Can we learn to predict general code changes by learning them in the wild and guide the code changes with developers guidance?*

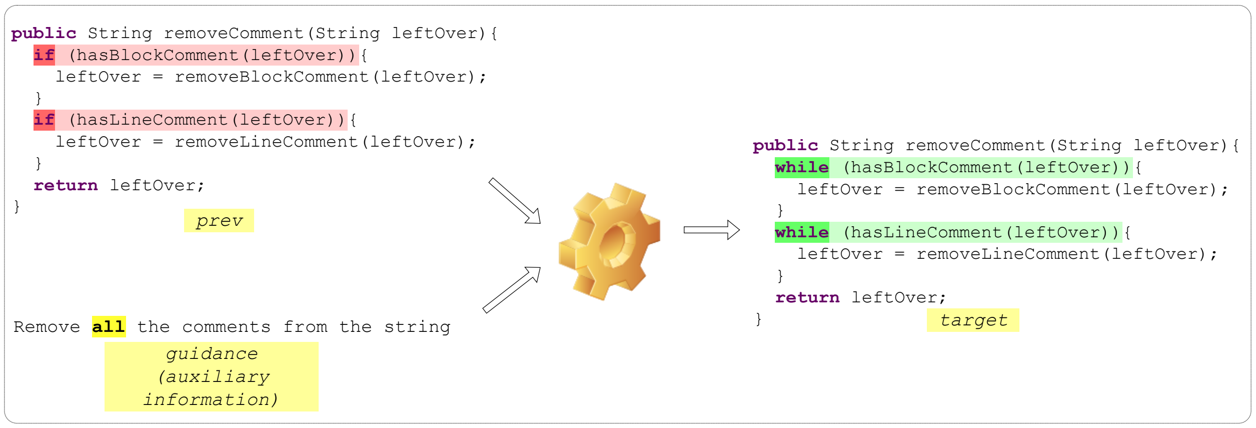
In this dissertation, I envision building tools that can change part of a software's source code following code change patterns previously adopted by developers and high-level requirements from developers. We define the Automated *Code Edit*[1] task is as modification of existing code (*i.e.*, adding, deleting, or replacing code elements) through applying such frequent change patterns [245, 252, 112, 223] . This dissertation proposes several techniques to build such editors. In this dissertation, my primary focus has been designing Machine Learning (ML) and Deep Learning (DL) based automated code editing tools. While building such an automated code editor, we investigated the design and adaptation of Programming Language (PL) specific properties into DL models. We hypothesize that the success of a DL model in a complex Software Engineering task such as Automated Code Editing depends on its capability in understanding source code, understanding other metadata associated with code, and correctly generating code. In this dissertation, we propose several novel DL models and techniques for attaining such goals and evaluate these models on automated code editing. We also evaluate the merit of these models on other crucial SE

[1]We use the term *Code Change* and *Code Edit* interchangeably

tasks (§5.4, §7.5).

1.1 Problem Formulation and Challenges

Automated Code Editing. We formulate the Automated Code Editing as a mechanized transformation of an existing source code snippet. Such a transformation can be pre-programmed based on the frequent transformation patterns learned from software's evolution history, or push-button: where a programmer provides guidance for code editing in natural language, and the editor takes both learned patterns and developers' guidance into consideration while editing the code. Formally, for a code to be automatically edited, the inputs the automated code editor tools are: the code version before the edit (*prev*), and a guidance from the developer (*guidance*), the output is the code version after the edit (*target*).



```
public String removeComment(String leftOver){
  if (hasBlockComment(leftOver)){
    leftOver = removeBlockComment(leftOver);
  }
  if (hasLineComment(leftOver)){
    leftOver = removeLineComment(leftOver);
  }
  return leftOver;
}
```

```
Remove all the comments from the string
```

```
public String removeComment(String leftOver){
  while (hasBlockComment(leftOver)){
    leftOver = removeBlockComment(leftOver);
  }
  while (hasLineComment(leftOver)){
    leftOver = removeLineComment(leftOver);
  }
  return leftOver;
}
```

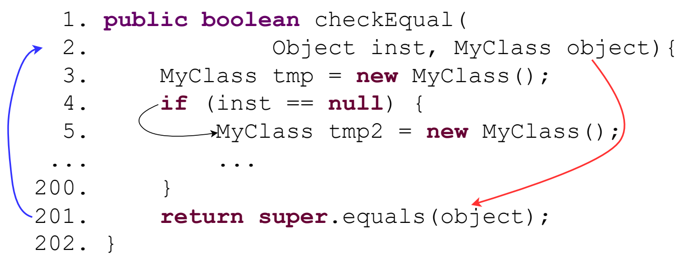
Figure 1.1: An example of code editing problem

Figure 1.1 shows one such automated edit example. To model the edits, one needs to learn the conditional probability distribution of the *target* code version ($\mathscr{T}$) given its *prev* code version ($\mathscr{P}$) and *guidance* ($\mathscr{G}$). A good probabilistic model will assign higher probabilities to plausible target versions and lower probabilities to less plausible ones. In particular, our goal is to design a parametric probabilistic model with trainable parameters θ, which maximizes the conditional probability of the edited code ($\mathscr{T}_i$) given the original code ($\mathscr{P}_i$) and other guidance(s) ($\mathscr{G}_i$) from a training dataset of examples edits ($\mathscr{D}$). Formally,

$$\theta^* = \arg\max_{\theta} \prod_{d_i \in \mathscr{D}} P(\mathscr{T}_i | \mathscr{P}_i, \mathscr{G}_i, \theta) \tag{1.1}$$

Automated Code Editing as Source Code Understanding and Generation. We deconstruct the Automated Code Editing problem into two sub-problems — Source Code Understanding and Source Code Generation. As shown in Figure 1.1, part of the model needs to understand the input code and how to process the input code. Another part of the model is responsible for generating the edited code. Encoder-decoder Neural Machine Translation models (NMT) are a promising approach to realizing such code edit models, where the previous code version (*i.e.prev*) and other inputs (*e.g.,guidance*) are encoded into latent vector representations. Then, the target version is synthesized (decoded) from the encoded representation. Previous research efforts [44, 225] investigated the initial viability of using NMT for code changes. At the core of the NMT, there is an encoder and a decoder (generator). The encoder encodes and understands the input(s); the decoder generates the changed or corrected code.

Challenges of Source Code Understanding. Modeling source code with deep learning brings a unique set of research challenges. First, unlike fuzzy semantic structures (multiple semantic parse trees based on the interpretation of sentence) of Natural Language [133, 121, 106, 86], Programming Languages exhibit precise syntactic and semantic structure [76, 158, 198]. However, the



```
public boolean checkEqual(
            Object inst, MyClass object){
    MyClass tmp = new MyClass();
    if (inst == null) {
        MyClass tmp2 = new MyClass();
        ...
    }
    return super.equals(object);
}
```

Figure 1.2: Code example showing long range dependency between code components

precision in the underlying semantic structure allows a program to be excessively long, with very long-range dependencies between components. For instance, Figure 1.2 shows a code snippet, where a variable `object` declared in line 2 is used in line 201. Such long-range dependencies demand a specific model design to understand the code properly. In addition, several types of semantic dependencies exist between the program components. For examples, in Figure 1.2, the black edge is a control flow edge, the red edge is a data flow edge, and the blue is a data-dependency edge. Thus, the model should understand and reason about these relationships between code components to fully understand a code.

Challenges of Source Code Generation. Like source code understanding, source code generation is a significant challenge for any ML/DL model because of the strict syntactic and semantic properties of source code. Unlike natural language, programs written in PL are consumed by the machine. Thus, the slightest syntactic and semantic error in a code can make the whole code be-

```
boolean f (Object target)  {
    for(Object elem : if.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}
```

(a) Syntactically Incorrect Code

```
boolean findNumbers (String target){
  for (int i = 0; i < target.length(); i++){
    char c = target.charAt(i);
    if (c >= '0' && c <= '9'){
      return true;
    }
  }
  return null;
}
```

(b) Semantically Incorrect Code

```
void processFile (String fileName){
    Scanner sc = new Scanner(new File(fileName));
    for (🟥; sc.hasNext() ;🟥) {
        String line = sc.nextLine();
        this.processLine(line);
        ...
    }
    ...
}
```

(c) Naturally Incorrect Code

Figure 1.3: Examples of incorrect code

come useless. For instance, Figure 1.3(a) shows an *almost correctly* generated code with mistake in one token. Such a mistake makes the generated code syntactically incorrect, hence unusable in development. On the other hand, code in Figure 1.3(b) is syntactically valid, but the method in the code is returning `null` , where it should return `boolean` , which is a semantic violation. Such a generated code will likely cause a *compilation error*, eventually resulting in a *build failure*. Finally, the third type of error we are concerned about is *unnatural code* — a piece of code that is syntactically and semantically correct. Still, a developer is less likely to write such a code. Such a generated code may impede the readability and maintainability of the software. For instance, if a developer had to write a `for` loop with the *initializer* and *update* blocks empty, they would likely write a `while` loop (see Figure 1.3(c)). To be fully usable in the development pipeline, the ML/DL-based code generator must demonstrate effectiveness in maintaining the correctness of the code.

1.2 Solution Approach and Research Contribution

A closer look into the challenges in DL-based source code understanding and generation (as discussed in Section 1.1) would reveal that PL-specific constructs (syntax, semantics) are the source of these challenges. Thus, our hypothesis in this dissertation is to equip the model with knowledge about these PL constructs to solve the problem. More specifically, we endow the models with knowledge about — (i) syntax, (ii) semantics, and (iii) common coding patterns followed by developers. We investigate two different approaches to such endowment – *explicit* and *implicit* encoding of PL properties in the model.

1.2.1 Explicit Encoding of PL Constructs

In this approach, we explicitly design models to adhere to PL properties. Regardless of the development environment, organization, and API libraries, few properties of PL are generic to any program in a particular programming language. For instance, every program written in `Java` must adhere to the context-free grammar provided by *Java Language Specification (JLS)*. Since these syntax rules are stringent and well defined, it is convenient to build models around these rules. Following this idea, we first developed CODIT [37][2] (details in Chapter 3). We investi-

[2]Published in TSE'20

gated the syntactic correctness guarantee of generated code through the programming language's Context-Free Grammar (CFG). Using a tree-based model, the decoder in CODIT samples from the CFG and generates the code's syntax tree. In doing so, it ensures the syntactic guarantee of the generated code. CODIT shows significant promise in automated code change and program repair. CODIT successfully fixed fifteen (15) bugs completely and ten(1) partially out of 80 bugs in the Defects4j [104] bug dataset. Note that we built CODIT to generate syntactically correct edited code and learn the code edit patterns from generic patches in GitHub. We did not use other auxiliary information (*e.g.,* bug report, failing test traces) to generate the patches.

While CODIT shows promise in learning code edit patterns and subsequently generating syntactically correct patches, we assume that CODIT should edit every piece of code it receives. To identify whether or not to edit a piece of code, we need to design a classifier that we can train to detect the need for editing. As an example of this sub-problem, we chose Vulnerability Detection [132, 253, 131, 204, 250] in the source code. In particular, given a training dataset of vulnerable (buggy) and benign (non-vulnerable) code, our goal is to train a classifier. Ideally, such a trained classifier would detect and classify the vulnerable code snippets from the rest. For this classifier to work correctly, it should reason about the syntactic and semantic structure of the code, which traditionally played a vital role in static analysis-based vulnerability detectors [53, 52]. Thus, we hypothesize that our classifier should know about semantic structure.

We performed an empirical study [38] [3] to test this hypothesis, where we compared different DL-based vulnerability detection tools (details in Chapter 4). Without much of a surprise, we found that when code is treated as a linear sequence of tokens with linear dependency and processed with recurrent neural networks(*e.g.,* RNNs, LSTMs, GRUs), they do not fully comprehend the semantics of code and thus learn based on other spurious features in the dataset. On the other hand, models are more equipped to learn about the semantic structure when we use a graph-based model (*e.g.,* GGNN [12]). We build a framework (ReVeal) for Vulnerability detection using GGNN and representation learning for learning Vulnerable (buggy) code patterns. Being a graph-based

[3]Published in TSE'21

model, ReVeal explicitly encodes the syntactic and semantic structures of the source code. The success of ReVeal over the then state-of-the-art models for vulnerability detection showed us the potential of explicit encoding of PL properties for program understanding. Through ReVeal and CODIT, we showed the prospect of code understanding and generation with explicit encoding.

1.2.2 Implicit Encoding of PL Constructs

While the syntax property of source code is strictly defined across the whole programming language, semantics largely depend on the libraries and APIs developers use. For instance, while `Java` language specification insists on returning the same or child type as declared in the method signature, different API libraries can extend a parent type into many different sub-types. In addition, coding patterns vary in different organizations' coding practices and policies. For example, some organizations may prefer handing exceptions locally in the method context. Some may prefer leaving the exception to the execution environment [205, 201]. The sheer volume of open source and proprietary libraries in modern PL[4], and the diversity of coding patterns make it extremely hard, if not impossible, to build a model to encode these rules explicitly.

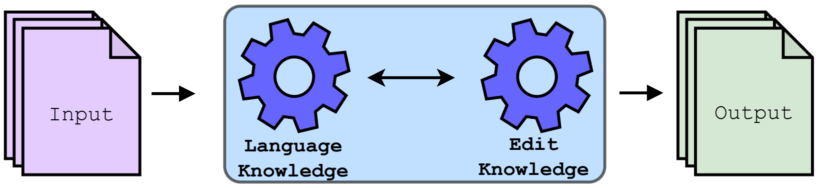


Figure 1.4: Conceptual breakdown of different knowledge in a code editor model

While designing an explicit encoding model for API semantics and coding patterns is very cumbersome, we have a crucial insight to encode these pieces of knowledge into the models. A successful Automated Code Editing Model should possess two types of knowledge – (i) Knowledge about PL and (ii) Knowledge about edit patterns. Since the API semantic patterns and coding convention patterns are source code specific and are not dependent on the "Automated Code Editing" task (or any task, for that matter), these patterns are amenable to be learned from large corpora

[4]As of 21 June 2022, there are more than 38,000 repositories in GitHub alone with tag 'java library'

of source code data. Availability of ultra-large source code and other code-related metadata allows us to implicitly learn these patterns in `task-agnostic` way. In particular, we can *pre-train* a source code model to learn these patterns in an unsupervised way from a large corpus of *unlabelled* source code data. We can further train such a pre-trained model to learn Code Edit patterns. In fact, since the knowledge we embed in a pre-trained model is independent of any particular task, developers can re-use such a pre-trained model in many different SE tasks.

Following the insight described above, we developed PLBART[5][5] – A pretraining mechanism for simultaneously understanding and generating code (details in Chapter 5). We leveraged denoising auto encoding [126] to pre-train the encoder and decoder. We applied PLBART for a wide variety of downstream software engineering tasks, where it showed great promise. We further performed an in-depth empirical investigation on using PLBART in Automated Code Editing. We developed a multi-modal code editing engine MODIT[39][6]. We showed that pre-trained models such as PLBART can learn PL constructs in an unsupervised way (details in Chapter 6). Furthermore, we showed the relative importance of different input modalities (*i.e.*, the code that needs to be edited, the surrounding code context where the patch is applied, and a guidance dictating the edit) in automated code editing. We further define a pre-training objective, "naturalization", to pre-train the models with natural coding conventions followed by the developers. In this work, we take developers' written code and apply semantic-preserving code transformation to make the code "unnatural" and "weird". We ask a model to transform those unnatural codes into their original forms. With such a training mechanism, we aim to make the model implicitly biased towards the natural coding convention followed by developers. With these insights, we developed NatGen [36][7] – a pre-trained model for source code generation (details in Chapter 7). Like PLBART, NatGen also showed great promise in different SE tasks, including Automated Code Editing.

To summarize, in this dissertation, we show that using Deep Learning models we can successfully edit source code at scale. We propose several new and novel modeling approaches for

[5]Published in NAACL'21.
[6]Published in ASE'21
[7]Published in FSE'22

Automated Source Code Editing. We investigate and demonstrate different perspectives and approaches for designing such code editing models and their respective strengths and shortcomings. The models we designed and the insights we provided in this dissertation should serve as a comprehensive guide to model designing for source code editing for future research. The research works that we present in this dissertation are as follows

1. CODIT: Code Editing with Tree-Based Neural Models - Published in IEEE Transaction of Software Engineering, 2020.

2. (Part of) ReVeal : Deep Learning Based Vulnerability Detection: Are We There Yet? - Published in IEEE Transaction of Software Engineering, 2021.

3. PLBART : Unified Pre-training for Program Understanding and Generation - Published in Annual Conference of the North American Chapter of the Association for Computational Linguistics, 2021.

4. MODIT: On Multi-Modal Learning of Editing Source Code - Published in IEEE/ACM International Conference on Automated Software Engineering, 2021.

5. NatGen : Generative pre-training by "Naturalizing" source code - Published in ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022.

In addition to being effective in automated code editing, these models showed great promise in other SE tasks, including Code Generation, Code Translation across programming languages, Code Summarization, Clone Detection, Vulnerability Detection, etc. Thus, I believe this thesis reaches beyond the scope of automated code editing problems and contributes to Artificial Intelligence of Software Engineering (AI4SE) and Programming Language Processing (PLP).

Chapter 2: Background and Related Work

2.1 Background

In this section, we will discuss different background ideas and concepts we utilize in different chapters in this thesis.

2.1.1 Modeling Code Changes.

Generating source code using machine learning models has been explored in the past [89, 222, 84, 176]. These methods model a probability distribution $p(c|\kappa)$ where c is the generated code and κ is any contextual information upon which the generated code is conditioned. In this work, we generate *code edits*. Thus, we are interested in models that predict code given its previous version. We achieve this using NMT-style models, which are a special case of $p(c|\kappa)$, where c is the new and κ is the previous version of the code. NMT allows us to represent code edits with a single end-to-end model, taking into consideration the original version of a code and defining a conditional probability distribution of the target version. Similar ideas have been explored in NLP for paraphrasing [147].

2.1.2 Grammar-based modeling.

Context Free Grammars (CFG) have been used to describe the syntax of programming languages [117] and natural language [45, 91]. A CFG is a tuple $G = (N, \Sigma, P, S)$ where N is a set of non-terminals, Σ is a set of terminals, P is a set of production rules in the form of $\alpha \rightarrow \beta$ and $a \in N$, $b \in (N \cup \Sigma)^*$, and S is the start symbol. A sentence (*i.e.* sequence of tokens) that belongs to the language defined by G can be parsed by applying the appropriate derivation rules from the start symbol S. A common technique for generation of utterances is to expand the left-most, bottom-most non-terminal until all non-terminals have been expanded. Probabilistic context-free grammar (PCFG) is an extension of CFG, where each production rule in associated with a probability, *i.e.* is defined as (N, Σ, P, Π, S) where Π defines a probability distribution for each production rule in P conditioned on α.

2.1.3 Neural Machine Translation

Neural Machine Translation(NMT) [23] is a very well studied field, which has been very successful in translating a sentence from one language to another. At a very high level, input to an NMT model is a sentence ($X = x_1, x_2, ..., x_n$), which is usually a sequence of tokens (x_i), and the output is also a sentence ($Y = y_1, y_2, ..., y_m$) – sequence of tokens (y_i). While learning to translate from X to Y, NMT models learn s learn conditional probability distribution $P(Y|X)$. Such probability distributions are learned *w.r.t.* model parameters θ, where model training process optimizes θ in such a way that maximizes the expected probability distribution of a dataset. An NMT model usually contains an encoder and a decoder. The encoder processes, understands, and generates vector representations of the input sentence. The decoder starts after the encoder and sequentially generates the target sentence by reasoning about the encoder-generated input representation. While sequentially generating the target sentence, the decoder usually performs different heuristic searches (for instance, beam search) to balance exploration and exploitation.

In recent few years, Software Engineering has seen a wide spectrum of adaptation of NMT. Some prominent application of NMT is SE include Program Synthesis [243], Code summarization [233, 4], Edit summarization [139], Code Edit Generation [225, 223, 37], Automatic Program Repair [146, 101, 44], etc. These research efforts capitalize on NMTs' capability to understand and generate complex patterns and establish NMT as a viable tool for SE-related tasks.

2.1.4 Transformer Model for Sequence Processing

Transformer [228] model revolutionized sequence processing with attention mechanism. Unlike the traditional RNN-based model where input tokens are processed sequentially, the transformer assumes soft-dependency between each pair of tokens in a sequence. Such dependency weights are learned in the form of attention weights based on the task of the transformer. While learning the representation of a token, the transformer learns to attend to all the input tokens. From a conceptual point of view, the transformer converts a sequence to a complete graph[1], where each node is a token. The weights of the edges are attention weights between tokens which are learned based on the task of the transformer. The transformer encodes each token's position in the sequence (positional encoding) as part of the input. In such a way, the transformer learns long-range dependency. Since its inception, the transformer is very successful in different NLP understanding and generation tasks. Transformers' ability of reasoning about long-range dependency is proved useful for several source code processing task including code completions [113], code generation [216], code summarization [4].

2.1.5 Transfer Learning for Source Code

In recent few years, Transfer learning [221, 234, 173] shows promise for a wide variety of SE tasks. Such transfer learning aims at learning task agnostic representation of source code and reuse such knowledge for different tasks. One way to learn such task agnostic representation of input is pre-training a model with a large collection of source code. The learning objective of such pre-training is often understanding the code or generating the correct code. A pre-trained model is expected to embed the knowledge about source code through its parameters. Such pre-trained models are later fine-tuned for task-specific objectives. CuBERT [107], CodeBERT [63], GraphCodeBERT [78] are all transformer-based encoder models which are pre-trained to understand code. Such models are primarily trained using Masked Language Model [59], replaced token prediction [63], semantic link prediction [78], etc. For code generation, CodeGPT [142, 101] pre-trains a transformer-based model to generate general-purpose code sequentially. More recently, PLBART [5] pre-trained transformer-based model jointly for understanding and generating code with denoising auto-encoding [126]. PLBART consists of an encoder and a decoder. The encoder

[1]https://en.wikipedia.org/wiki/Complete_graph

is presented with slight noise (for instance, token replacement) induced code, and the decoder is expected to generate noise-free code. Since code editing task requires both the understanding of code and code generation, we chose PLBART as the base model for MODIT. Similar to PLBART, NatGen also consist of an encoder and a decoder. In contrast to PLBART's knowledge about source code generation, NatGen exhibits an implicit bias towards learning natural coding patterns followed by developers.

2.2 Related Works

Automatic Code Change. There are a lot of research efforts to capture repetitiveness of developers' way of editing source code. These researches show the potential of automatic refactoring [69, 194], boilerplate code [152] etc. These research efforts include (semi-)automatic tools involving traditional program analysis techniques (*e.g.,* clone detection, dependency analysis, graph matching) [199, 153]. Other research direction aims at learning source code edit from previous edits and applying those edit patterns in similar context [192, 166]. Some of these efforts targets very specific code changes; For example, Nguyen *et al.* [167] proposed a graph-matching-based approach for automatically updating API usage. Tansey *et al.* [218] semantic preserving transformation of java classes for automated refactoring. Other directions of works address more general-purpose code change learned from open source repositories [223, 37]. Such approaches target solving automated code editing tasks in a data-driven approach, and the edit patterns are learned from example changes. In this research, we also investigated general purpose source code changes in the wild. More closely to CODIT, Rolim *et al.* [200]'s proposed technique constraints source code generation with additional input/output specification or test cases. Nevertheless, we argue that textual guidance could be a very good surrogate specification.

Machine Translation (MT) for source code. MT is used to translate source code from one programming language into another [108, 164, 163, 42]. These works primarily used Seq2Seq model at different code abstractions. In contrast, we propose a syntactic, tree-based model. More closely to our work, Tufano *et al.* [224, 223], and Chen *et al.* [43] showed promising results using a Seq2Seq model with attention and copy mechanism. Our baseline Seq2Seq model is very similar to these models. However, Tufano *et al.* [225, 224] employed a different form of abstraction: using a heuristic, they replace most of the identifiers including variables, methods and types with abstract names and transform previous and new code fragments to abstracted code templates. This vaguely resembles CODIT's $\mathscr{M}_{tree}$ that predicts syntax-based templates. Gupta *et al.* used Seq2Seq models to fix C syntactic errors in student assignments [80]. However, their approach can fix syntactic errors for 50% of the input codes *i.e.* for rest of the 50% generated patches were syntactically incorrect which is never the case for CODIT because of we employ a tree-based approach. Lutellier *et al.* [146] treated code needs to be changed and the context as two difference modalities and use separate encoders. However, our empirical evidence showed that using one encoder to encode all the modalities result in the best performance. More recently, Ding *et al.* [58] presented empirical evidence that instead of generating a whole code element (*i.e.* context+change) of the target version, only generating the sequence of changes might perform better for code change modeling. Other NMT application in source code and software engineering include program comprehension [15, 92, 233, 4], commit message generation [241, 139], program synthesis [243, 183] etc.

Structure Based Modeling of Code. Code is inherently structured. Many form of structured modeling is used in source code over the years for different tasks. Allamanis *et al.* [14, 8] proposed statistical modeling technique for mining source code idioms, where they leverages probabilistic Tree Substitution Grammar (pTSG) for mining code idioms. CODIT's $\mathscr{M}_{tree}$ is based on similar concept, where we model the derivation rule sequence based on a probabilistic Context Free Grammar. Brockschmidt *et al.* [31], Allmanis *et al.* [12] proposed graph neural network for modeling source code. However, their application scenario is different from CODIT's application, *i.e.* their focus is mainly on generating natural looking code and/or identify bugs in code. Recent researches that are very close to CODIT include Yin *et al.* [244]'s proposed graph neural network-based distributed representation for code edits but their work focused on change representation than generation. Other recent works that focus on program change or program repair include Graph2Diff by Tarlow *et al.* [220], Hoppity by Dinella *et al.* [56]. These research results are promising and may augment or surpass CODIT's performance, but problem formulation between these approach are fundamentally different. While these technique model the change only in the code, we formulate the problem of code change in encoder-decoder fashion, where encoder-decoder implicitly models the changes in code.

Program Repair . Automatic program repair is a well-researched field, and previous researchers proposed many generic techniques for general software bugs repair [105, 111, 124, 135, 140]. There are two different directions in program repair research : generate and validate approach, and synthesis bases approach. In generate and validate approaches, candidate patches are first generated and then validated by running test cases [111, 125, 141, 206, 235]. Synthesis based program repair tools synthesizes program elements through symbolic execution of test cases [168, 151]. CODIT can be considered a program generation tool in generate and validate based program-repair direction. Arcuri *et al.* [21], Le Goues *et al.* [125] built their tool for program repair based on this assumption. Both of these works used existing code as the search space of program fixes. Elixir [206] used 8 predefined code transformation patterns and applied those to generate patches. CapGen [235] prioritize operator in expression and fix ingredients based in the context of the fix. They also relied on predefined transformation patterns for program mutation. In contrast, CODIT learns the transformation patterns automatically. Le *et al.* [123] utilized the development history as an effective guide in program fixing. They mined patterns from existing change history and used existing mutation tool to mutate programs. They showed that the mutants that match the mined patterns are likely to be relevant patch. They used this philosophy to guide their search for program fix. The key difference between Le *et al.* and this work, is that we do not just mine change patterns, but learn a probabilistic model that learns to generalize from the limited data.

Deep Learning for Source Code Analysis Vast sources of code in open source repositories and forums make deep learning feasible for SE tasks. Code Summarization [161, 13, 97, 16, 92, 82, 3], Bug Detection [190, 132, 204, 250, 38], Code Translation [42, 60, 240], Clone Detection [248, 246, 230], Code completion [128, 85, 176] are some of the tasks that are addressed with deep neural solution. While most of the prior approaches use task-specific representation learning, a few works [17, 63, 78, 122, 47] attempted to learn transferable representations in an unsupervised fashion. More closely PLBART and NatGen, CodeBERT [63] is pre-trained on bimodal data to capture the semantic interaction between the input modalities (*i.e.* program and natural languages). More recently, GraphCodeBERT [78] improves upon CodeBERT by leveraging data flow in source code.

Transformer Models in Source Code Analysis. The approach of pre-training large Transformers without human labels started in NLP domain with BERT [54], which introduces two pre-training objectives (*i.e.*, Mask Language Modeling and Next Sentence Prediction). Later, Liu *et al.*. show that RoBERTa [138] outperforms BERT only using Mask Language Modeling (MLM) with new training strategies and hyper-parameter tuning. MLM is a self-supervised task that the model randomly masks or modifies a certain number of tokens and tries to recover them. Following the success of the pre-trained model in the NLP domain, researchers applied these models to code related tasks. CodeBERT is one of the earliest that was specially trained for code and relevant natural language descriptions. It is pre-trained with two objectives (i.e., MLM and Replaced Token Detection [46]) and demonstrated pre-training's effectiveness for code. Later, an architecturally equivalent model, GraphCodeBERT, was introduced; it improved over CodeBERT on most tasks by incorporating data-flow information.

Though CodeBERT [63] & GraphCodeBERT [78] do well at code understanding tasks, these models are not as good at generative tasks. Both models are encoder-only and have to start with an untrained decoder in fine-tuning for generative tasks, such as code repair, code generation, code summarization, and code translation. To address this limitation,we developed PLBART [5], pre-trained as a generative denoising autoencoder. A specific set of noises is introduced to code and relevant natural language description and used as the input to the model. The model's objective is to encode the noisy input in the encoder and generate noise-free code or text in the decoder. PLBART (builds on BART [126]) outperforms both CodeBERT [63] and GraphCodeBERT [78] on both understanding and generative tasks with a pre-trained encoder and decoder [5]. DOBF [203] uses de-obfuscation (recovering variable names) as their pre-training task; however, rather than generating code, they just generate a dictionary of recovered names. CodeT5 [232] (based T5 [188]) is the latest denoising model. CodeT5 uses the developer-assigned identifiers in code, adding two code-specific pre-training objectives to the original T5, identifier tagging and masked identifier prediction. CodeT5 is an encoder-decoder model and excels at both understanding and generative tasks compared to other models. Similar to CodeT5, [181, 150] are also built based on T5 architecture and perform reasonably well in the different downstream tasks. NatGen has a similar architecture to CodeT5; but rather than CodeT5's pre-training objectives, we "de-naturalize" code, using the formal channel of code to inject meaning-preserving transforms, and then force NatGen to recreate, the original, "natural" code. Rewriting semantically equivalent code requires semantic understanding, and that can be applied to code only because of its dual-channel nature. Our evaluation shows that rewriting semantically equivalent programs in the pre-training stage results in performance gains in at least three popular Software Engineering tasks.

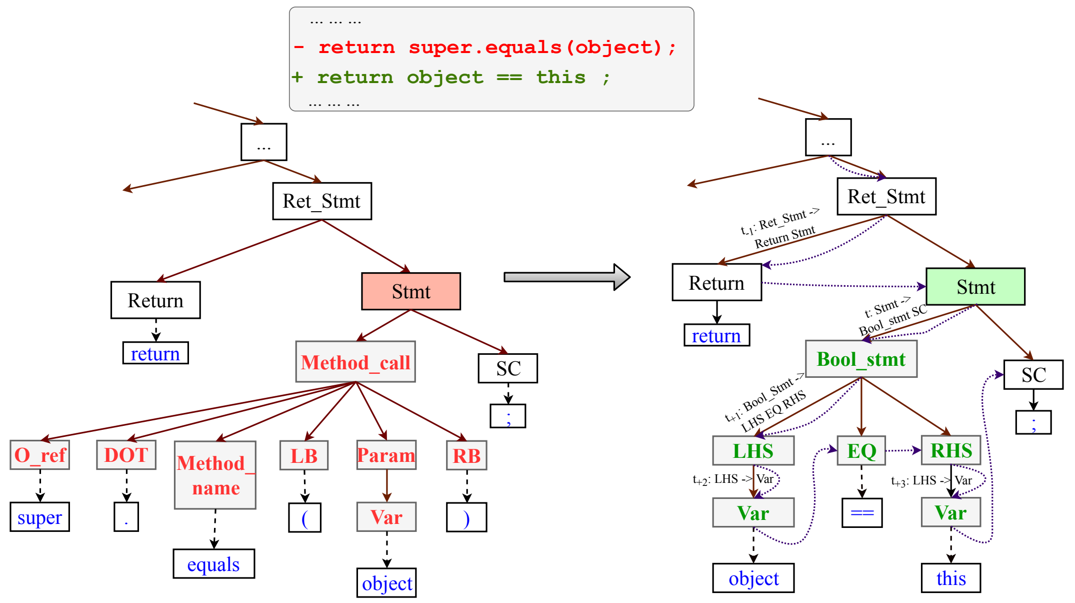
Part I
Explicit Encoding

Chapter 3: Code Editing with Tree-Based Neural Models

3.1 Motivation

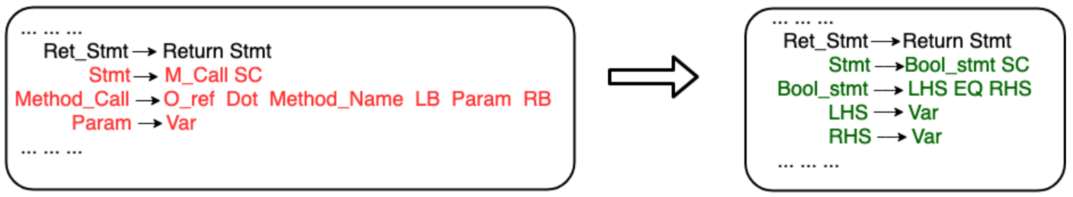
We design an encoder-decoder-based machine translation model that operates on the tree representation of the code to capture syntactic changes. Our key observation is that such tree-based models, unlike their token-based counterparts, can capture the rich structural properties of code and thus can produce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

Figure 3.1 illustrates an example of our approach. Here, the original code fragment `return super.equals(object)` is edited to `return object == this`. CODIT takes these two code fragments along with their context, for training. While suggesting changes, *i.e.*, during test time, CODIT takes as input the previous version of the code and generates its edited version. CODIT operates on the parse trees of the previous (t_p) and new (t_n) versions of the code, as shown in Figure 3.1(a) (In the rest of the paper, a subscript or superscript with p and n correspond to previous and new versions respectively). In Figure 3.1, changes are applied only to the subtree rooted at the $Method\_call$ node. The subtree is replaced by a new subtree (t_n) with `Bool_stmt` as a root.



```
... ... ...
- return super.equals(object);
+ return object == this ;
... ... ...
```

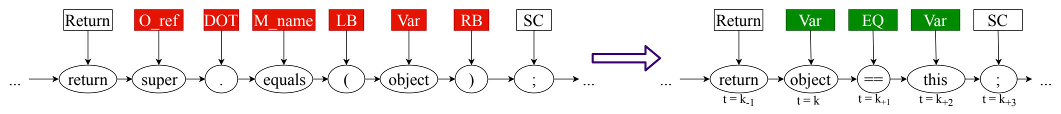
(a) **Example of correctly suggested change by CODIT along with the source and target parse trees. The deleted and added nodes are marked in red and green respectively.**



```
... ... ...
  Ret_Stmt ⟶ Return Stmt
      Stmt ⟶ M_Call SC
Method_Call ⟶ O_ref  Dot  Method_Name  LB  Param  RB
     Param ⟶ Var
... ... ...
```

```
... ... ...
  Ret_Stmt ⟶ Return Stmt
      Stmt ⟶ Bool_stmt SC
 Bool_stmt ⟶ LHS EQ RHS
       LHS ⟶ Var
       RHS ⟶ Var
... ... ...
```

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



(c) Token generation. Token probabilities are conditioned based on terminal types generated by tree translator (see figure 3.1(a))

Figure 3.1: Illustrative Example showing CODIT's working procedure.

The deleted and added subtrees are highlighted in **red** and **green** respectively.

While modeling the edit, CODIT first predicts the structural changes in the parse tree. For example, in Figure 3.1(a) CODIT first generates the changes corresponding to the subtrees with dark nodes and red edges. Next the structure is concretized by generating the token names (terminal nodes). This is realized by combining two models: (i) a tree-based model predicting the structural change (see §3.2.1) followed by a (ii) a token generation model conditioned on the structure generated by the tree translation model (see §3.2.2).

Tree Translator. The tree translator is responsible for generating structural changes to the tree structure. A machine learning model is used to learn a (probabilistic) mapping between t_p and t_n. First, a tree encoder, encodes t_p computing a distributed vector representation for each of the production rules in t_p yielding the distributed representation for the whole tree. Then, the tree decoder uses the encoded representations of t_p to sequentially select rules from the language grammar to generate t_n. The tree generation starts with the `root` node. Then, at each subsequent step, the bottom-most, left-most non-terminal node of the current tree is expanded. For instance, in Figure 3.1(a), at time step `t`, node `Stmt` is expanded with rule **`Stmt`** → **`Bool_Stmt`** **`SC`**. When the tree generation process encounters a terminal node, it records the node type to be used by the token generation model and proceeds to the next non-terminal. In this way, given the LHS rule sequences of Figure 3.1(b) the RHS rule sequences is generated.

Token Generator: The token generator predicts concrete tokens for the terminal node types generated in the previous step. The token generator is a standard seq2seq model with attention and copying [23] but constrained on the token types generated by the tree translator. To achieve this, the token generator first encodes the token string representation and the node type sequence from t_p. The token decoder at each step probabilistically selects a token from the vocabulary or copies one from the input tokens in t_p. However, in contrast to traditional seq2seq where the generation of each token is only conditioned on the previously generated and source tokens, we additionally condition on the token type that has been predicted by the tree model and generate only tokens that are valid for that toke type. Figure 3.1(c) shows this step: given the token sequence of the original code **`super`** `. equals ( object )` and their corresponding token types (given in dark box), the new token sequence that is generated is `object == ` **`this`** .

In this work, we particularly focus on smaller changes as our previous experience [192] shows that such changes mostly go through similar edits. In fact, all the previous NMT-based code transformation works [223, 224, 43] also aim to automate such changes. A recent study by Karampatsis *et al.* [110] showed that small changes are frequent—our analysis of the top 92 projects of their dataset found that, on average, in each project, one line changes take place around 26.71% of the total commits and account for up to 70% for the bug fix changes. Our focus in this work is primarily to automatically change small code fragments (often bounded by small AST sizes and/or few lines long) to reflect such repetitive patterns. Note that, in theory, our approach can be applied to any small fragment of code in the project repository with any programming language. However, for prototyping, we designed CODIT to learn changes that belong to methods of popular java project in Github.

In this work, we collect a new dataset — *Code-Change-Data*, consisting of 32,473 patches from 48 open-source GitHub projects collected from Travis Torrent [29]. Our experiments show CODIT achieves 15.94% patch suggestion accuracy in the top 5 suggestions; this result outperforms a Copy-Seq2Seq baseline model by 63.34% and a Tree2Seq based model by 44.37%. We also evaluate CODIT on *Pull-Request-Data* proposed by Tufano *et al.* [223]. Our evaluation shows that CODIT suggests 28.87% of correct patches in the top 5 outperforming Copy-Seq2Seq-based model by 9.26% and Tree2Seq based model by 22.92%. Further evaluation on CODIT's ability to suggest bug-fixing patches in Defects4J shows that CODIT suggests 15 complete fixes and 10 partial fixes out of 80 bugs in Defects4J.

3.2 Methodology

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

$$P(c_n|c_p) = P(c_n|t_n, c_p)P(t_n|t_p)P(t_p|c_p) \tag{3.1}$$

and our goal is to find $\hat{c}_n$ such that $\hat{c}_n = argmax_{c_n}P(c_n|c_p)$. Here, c_p is the previous version of the code and t_p is its parse tree, whereas c_n is the new version of the code and t_n its parse tree. Note that parsing a code fragment is unambiguous, *i.e.* $P(t_p|c_p) = 1$. Thus, our problem takes the form

$$\hat{c}_n = \arg\max_{c_n, t_n} \underbrace{P(c_n|t_n, c_p)}_{\mathscr{M}_{token}} . \underbrace{P(t_n|t_p)}_{\mathscr{M}_{tree}} \tag{3.2}$$

Equation 3.2 has two parts. First, it estimates the changed syntax tree $P(t_n|t_p)$. We implement this with a tree-based encoder-decoder model (section 3.2.1). Next, given the predicted syntax tree t_n, we estimate the probability of the concrete edited code with $p(c_n|t_n, c_p)$ (Section 3.2.2).

3.2.1 Tree Translation Model

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

For example, consider the tree fragments in Figure 3.1(a). Figure 3.1(b) shows the sequence of rules that generate those trees. For example, in the right tree of Figure 3.1(a), the node `Ret_Stmt` is first expanded by the rule: `Ret_Stmt→Return Stmt`. Since, `Return` is a terminal node, it is not expanded any further. Next, node `Stmt` is expanded with rule: `Stmt→Bool_Stmt SC`. The tree is further expanded with `Bool_Stmt→LHS EQ RHS`, `LHS→Var`, and `RHS→Var`. During the tree generation process, we apply these rules to yield the tree fragment of the next version.

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

$$P(t_n|t_p) = \prod_{k=1}^{\tau} P(R_k^n|R_1^n, ...R_{k-1}^n, t_p) \tag{3.3}$$

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

$$\boldsymbol{h}_i^p = f_{LSTM}(\boldsymbol{h}_{i-1}^p, \boldsymbol{r}_{R_i^p}) \tag{3.4}$$

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

Decoder: Our decoder has an LSTM with an attention mechanism as described by Bahdanau *et al.* [23]. The decoder LSTM is initialized with the final output from the encoder, *i.e.* $\boldsymbol{h}_0^n = \boldsymbol{h}_\tau^p$. At a given decoding step k the decoder LSTM changes its internal state in the following way,

$$\boldsymbol{h}_k^n = f_{LSTM}(\boldsymbol{h}_{k-1}^n, \boldsymbol{\psi}_k), \tag{3.5}$$

where $\boldsymbol{\psi}_k$ is computed by the attention-based weighted sum of the inputs $\boldsymbol{h}_j^p$ as [23] in , *i.e.*

$$\boldsymbol{\psi}_k = \sum_{j=1}^{\tau} softmax({\boldsymbol{h}_{k-1}^n}^T \boldsymbol{h}_j^p)\boldsymbol{h}_j^p \tag{3.6}$$

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

$$P(R_k^n | R_1^n, ...R_{k-1}^n, t_p) = softmax(W_{tree} \cdot \boldsymbol{h}_k^n + \mathbf{b}_{tree}) \tag{3.7}$$

At each timestep, we pick a derivation rule R_k^n following equation (3.7) to expand the `frontier_node` (n_f^t) in a depth-first, left-to-right fashion. When a terminal node is reached, it is recorded to be used in $\mathscr{M}_{token}$ and the decoder proceeds to next non-terminal. In Equation (3.7), W_{tree} and $\mathbf{b}_{tree}$ are parameters that are jointly learned along with the LSTM parameters of the encoder and decoder.

3.2.2 Token Generation Model

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

$$P(c_n|c_p, t_n) = \prod_{k=1}^{m'} P(x_k^n|x_1^n, ..., x_{k-1}^n, \{x_1^p, ..., x_m^p\}, \theta_k^n) \tag{3.8}$$

Here, θ_k^n is the node type corresponding to the generated terminal token x_k^n. Note that, the token generation model can be viewed as a conditional probabilistic translation model where token probabilities are conditioned not only on the context but also on the type of the token (θ_*^*). Similar to $\mathscr{M}_{tree}$, we use an encoder-decoder. The encoder encodes each token and corresponding type of the input sequence into a hidden representation with an LSTM (figure 3.1(c)). Then, for each token (x_i^p) in the previous version of the code, the corresponding hidden representation (s_i^p) is given by: $\boldsymbol{s}_i^p = f_{LSTM}(\boldsymbol{s}_{i-1}^p, enc([x_i^p, \theta_i^p]))$. Here, θ_i^p is the terminal token type corresponding to the generated token x_i^p and $enc()$ is a function that encodes the pair of x_i^p, θ_i^p to a (learnable) vector representation.

The decoder's initial state is the final state of the encoder. Then, it generates a probability distribution over tokens from the vocabulary. The internal state at time step k of the token generation is $\boldsymbol{s}_k^n = f_{LSTM}(\boldsymbol{s}_{k-1}^n, enc(x_i^n, \theta_k^n), \boldsymbol{\xi}_k))$, where $\boldsymbol{\xi}_k$ is the attention vector over the previous version of the code and is computed as in Equation (3.6). Finally, the probability of the kth target token is computed as

$$P(x_k^n|x_1^n, ..., x_{k-1}^n, \{x_1^p, ..., x_m^n\}, \theta_k^n) = softmax\left(W_{token} \cdot \boldsymbol{s}_k^n + \boldsymbol{b}_{token} + mask(\theta_k^n)\right) \tag{3.9}$$

Here, W_{token} and $\boldsymbol{b}_{token}$ are parameters that are optimized along with all other model parameters.

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

3.2.3 Implementation

Our tree-based translation model is implemented as an edit recommendation tool, CODIT. CODIT learns source code changes from a dataset of patches. Then, given a code fragment to edit, CODIT predicts potential changes that are likely to take place in a similar context. We implement CODIT extending OpenNMT [116] based on PyTorch. We now discuss CODIT's implementation in details.

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

Parse Tree Representation. As a first step of the training process, CODIT takes a dataset of patches as input and parses them. CODIT works at method granularity. For a method patch Δm, CODIT takes the two versions of m: m_p and m_n. Using GumTree, a tree-based code differencing tool [62], it identifies the edited AST nodes. The edit operations are represented as insertion, deletion, and update of nodes *w.r.t.* m_p. For example, in Figure 3.1(a), **red** nodes are identified as deleted nodes and **green** nodes are marked as added nodes. CODIT then selects the minimal subtree of each AST that captures all the edited nodes. If the size of the tree exceeds a maximum size of max_change_size, we do not consider the patch. CODIT also collects the edit context by including the nodes that connect the root of the method to the root of the changed tree. In order for doing such, we traverse from change subtree towards the root of the code tree. CODIT expands the considered context by adding immediate parent of a node in the context until it exceeds a tree size threshold (max_tree_size). During this process, CODIT excludes changes in comments and literals. Finally, for each edit pair, CODIT extracts a pair (AST_p, AST_n) where AST_p is the original AST where a change was applied, and AST_n is the AST after the changes. CODIT then converts the ASTs to their parse tree representation such that each token corresponds to a terminal node. Thus, a patch is represented as the pair of parse trees (t_p, t_n).

Information Extraction. CODIT extracts grammar rules, tokens and token types from t_p and t_n. To extract the rule sequence, CODIT traverses the tree in a depth-first pre-order way. From t_p, CODIT records the rule sequence (R_1^p, ..., R_τ^p) and from t_n, CODIT gets (R_1^n, ..., $R_{\tau'}^n$) (Figure 3.1(b)). CODIT then traverses the parse trees in a pre-order fashion to get the augmented token sequences, *i.e.* tokens along with their terminal node types: (x_*^p, θ_*^p) from t_p and (x_*^n, θ_*^n) from t_n. CODIT traverses the trees in a left-most depth-first fashion. When a terminal node is visited, the corresponding augmented token (x_*^*, θ_*^*) is recorded.

Model Training. We train the tree translation model ($\mathscr{M}_{tree}$) and token generation model ($\mathscr{M}_{token}$) to optimize Equation (3.3) and Equation (3.8) respectively using the cross-entropy loss as the objective function. Note that the losses of the two models are independent and thus we train each model separately. In our preliminary experiment, we found that the quality of the generated code is not entirely correlated to the loss. To mitigate this, we used top-1 accuracy to validate our model. We train the model for a fixed amount of n_{epoch} epochs using early stopping (with patience of $valid_{patience}$) on the top-1 suggestion accuracy on the validation data. We use stochastic gradient descent to optimize the model.

Model Testing. To test the model and generate changes, we use beam-search [196] to produce the suggestions from $\mathscr{M}_{tree}$ and $\mathscr{M}_{token}$. First given a rule sequence from the previous version of the tree, CODIT generates K_{tree} rule sequences. CODIT subsequently use these rule sequence to build the actual AST. While building the tree from the rule sequence, CODIT ignores the sequence if the rule sequence is infeasible (*i.e.*, head of the rule does not match the `frontier_node`, n_f^t). Combining the beam search in rule sequence and the tree building procedure, CODIT generate different trees reflecting different structural changes. Then for each tree, CODIT generates K_{token} different concrete code. Thus, CODIT generates $K_{tree} \cdot K_{token}$ code fragments. We sort them based on their probability, *i.e.* $log(P(c_n|c_p, t_p)) = log(P(c_n|c_p, t_n) \cdot P(t_n|t_p))$. From the sorted list of generated code, we pick the top K suggestions.

3.3 Experimental Design

Table 3.1: Summary of datasets used to evaluate CODIT.

Dataset	# Projects	# Train	# Validtion	# Test	# Tokens		# Nodes	
		Examples	Examples	Examples	Max	Average	Max	Average
Code-Change-Data	48	24072	3258	5143	38	15	47	20
Pull-Request-Data [43]	3	4320	613	613	34	17	47	23
Defects4J-data [104]	6	22060	2537	117	35	16	43	21

We evaluate CODIT for three different types of changes that often appear in practice: (i) code change in the wild, (ii) pull request edits, and (iii) bug repair. For each task, we train and evaluate CODIT on different datasets. Table 3.1 provides detailed statistics of the datasets we used.

(i) *Code Change Task.* We collected a large scale real code change dataset (*Code-Change-Data*) from 48 open-source projects from GitHub. These projects also appear in TravisTorrent [29] and have at least 50 commits in Java files. These project are excludes any forked project, toy project, or unpopular projects (all the projects have at least 10 watchers in GitHub). Moreover, these projects are big and organized enough that they use Travis Continuous integration system for maintainability. For each project, we collected the revision history of the main branch. For each commit, we record the code before and after the commit for all Java files that are affected. In total we found java 241,976 file pairs. We then use GumTree [62] tool to locate the change in the file and check whether the changes are inside a method in the corresponding files. Most of the changes that are outside a method in a java class are changes related to import statements and changes related to constant value. We consider those out of CODIT's scope. We further remove any pairs, where the change is only in literals and constants. Excluding such method pairs, we got 182,952 method pairs. We collect patches where *maximum change size* is upto 10 nodes and we allow maximum 20 nodes as context. With this data collection hyper-parameter settings, we collected 44,382 patches.

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

(ii) *Pull Request Task.* For this task, we use *Pull-Request-Data*, provided by Tufano *et al.* [223] which contains source code changes from merged pull requests from three projects from Gerrit [70]. Their dataset contains 21774 method pairs in total. Similar to the *Code-Change-Data*, we only consider *maximum change size* upto 10 nodes and context tree size upto 20 nodes to extract examples that are in CODIT's scope from this dataset. We extracted 5546 examples patch pair.

(iii) *Bug Repair Task.* For this task, we evaluate CODIT on Defects4J [104] bug-fix patches. We extract 117 patch pairs across 80 bug ids in Defect4j dataset with the same data collection configuration as Code Change Task. These are the bugs that are in CODIT's scope. To train CODIT for this task, we create a dataset of code changes from six projects repositories in Defects4J dataset containing 24597 patches. We remove the test commits from the training dataset.

3.3.1 Evaluation Metric

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

For the bug fix task, CODIT takes a buggy line as input and generates the corresponding patched code. We consider a bug to be fixed if we find a patch that passes all the test cases. While such an evaluation gives us a good estimate of how the CODIT is performing, it does not guarantee generating human written patch. To further investigate, we also manually investigate the patches to check the similarity with the developer provided patches.

3.3.2 Baseline

We consider several baselines to evaluate CODIT's performance. Our first baseline in a vanilla LSTM based Seq2Seq model with attention mechanism [23]. Results of this baseline indicate different drawbacks of considering raw code as a sequence of token. The second baseline, we consider, is proposed by Tufano *et al.* [224]. For a given code snippet (previous version), Tufano *et al.* first abstracted the identifier names and stored the mapping between original and abstract identifiers in a symbol table. The resultant abstracted code (obtained by substituting the raw identifiers with abstract names) is then translated by an NMT model. After translation, they concretize the abstract code using the information stored in the symbol table. We observe that the NMT sometimes introduces new abstract identifiers (*i.e.* identifiers that were not present in the before-patch version of code) in the patched code. The patches where the abstract symbols predicted by the model are not found in the symbol table remain undecidable. Such patches, although can be useful to guide developers similar to our $\mathscr{M}_{tree}$, cannot be automatically concretized, and thus, we do not count them as fully correct patches. Both the vanilla Seq2Seq and Tufano *et al.*'s model consider the before version of the code as input. Recently, SequenceR [43] proposed way to represent additional context to help the model generate concrete code. We design such a baseline, where we add additional context to c_P. Following SequenceR, we add copy attention, where the model learns to copy from the contexed code.

To understand the tree encoding mechanism, we used several tree encoders. First method we considered is similar to DeepCom [92], where the AST is represented as a sequential representation called Structure Based Traversal (SBT). Second tree encoding method we consider is similar to Code2Seq, where code AST is represented by a set of paths in the tree. While these tree encoding methods are used for generating Code comment, we leverage these encoding methods for code change prediction. We design a Seq2Seq method with DeepCom encoder (Tree2Seq), and Code2Seq encoder. We also enable the copy attention in both of these baselines.

The basic premise of CODIT is based on the fact that code changes are repetitive. Thus, another obvious baseline is to see how CODIT performs *w.r.t.* to code-clone based edit recommendation tool [192]. In particular, given a previous version (c_p) of an edit, we search for the closest k code fragments using similar bag-of-words at the token level similar to Sajnani *et al.* [208]. In our training data of code edits, this step searches in the previous code versions and use the corresponding code fragments of the next version as suggested changes.

Bug-fixing baselines: For the bug fix task, we compare CODIT's performance with two different baselines. Our first baseline is SequenceR [43], we compare with the results they reported. We also compare our result with other the state-of-the-art non-NMT based program repair systems — Elixir [206].

3.4 Research Findings

We evaluate CODIT's performances to generate concrete patches *w.r.t.* generic edits (RQ-3.1) and bug fixes (RQ-3.3). In RQ-3.2, we present an ablation study to evaluate our design choices.

RQ-3.1. How accurately can CODIT suggest concrete edits?

To answer this RQ, we evaluate CODIT's accuracy *w.r.t.* the evaluation dataset containing concrete patches. Table 3.2 shows the results: for *Code-Change-Data*, CODIT can successfully generate 201 (3.91%), 571 (11.10%), and 820 (15.94%) patches at top 1, 2, and 5 respectively. In contrast, at top 1, SequenceR generates 282 (5.48%) correct patches, and performs the best among all the methods. While SequenceR outperforms CODIT in top 1, CODIT outperforms SequenceR

Table 3.2: Performance of CODIT suggesting concrete patches. For Token Based models, predominant source of information in the code are token sequences. For Tree Based models information source is code AST. For IR based method, information retrieval model is used on code.

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

with significant margin at top 2 and top 5.

In *Pull-Request-Data*, CODIT generates 57 (9.3%), 134 (21.86%), and 177 (28.87%) correct patches at top 1, 2, and 5 respectively. At top 1, Tufano *et al.*'s [224] method produces 81 patches. At top 2, CODIT produces 134 (21.86%) patches, which is comparable with SequenceR's result 137 (22.35%). At top 5, CODIT outperforms all the other baselines achieving 9.2% gain over SequenceR baseline.

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

Among the baselines we compared here, SequenceR, and Tree2Seq takes the advantage of copy attention. Tufano *et al.*'s model takes the advantage of reduced vocabulary through identifier

Example 1. API Change

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

Example 2. Type Change

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

Example 3. Parameter: Add/Delete Method Parameter

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

Example 4. Refactoring: Modify Method Parameters Name

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

Example 5. Statement: Add Statement

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

Example 6. Inheritance: Abstracting a Method

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

Example 7. Exception Change: Add Try Block

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

Example 8. Other: Delete Unreferenced Variable

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

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

Figure 3.2: Examples of different types of concrete patches generated by CODIT

Ex:1. return ~~Error~~ EOFException.class ;

Ex:2. return ~~Exception~~ EOFException.class ;

Ex:3. return ~~RuntimeException~~ EOFException.class ;

Ex:4. return new ~~Error~~ EOFException(msg) ;

Ex:5. return new ~~Exception~~ EOFException(msg);

Ex:6. return new ~~RuntimeException~~ EOFException(msg);

Figure 3.3: Examples of CODIT's ability to generalize in different use cases. `Exception`, `Error`, `RuntimeException` are modified to `EOFException` under different context.

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

In code change task with NMT, the deterministic positions of code tokens are important to put attention over different parts of code. While Code2Seq [15]'s representation of input code as random collection of paths in the code showed success for code comprehension, it does not generalize well for code change due to the stochastic nature of the input. Additionally, copy attention cannot be trivially applied to Code2Seq since, like attention, copy also rely on the defined positions of tokens in the input.

While CODIT replaces any `<unknown>` prediction by the token with highest attention score (see section 3.2.2), unlike SequenceR, CODIT does not learn to copy. The rationale is, unlike SequenceR, CODIT's input code (c_p) is not enhanced by the context. Instead, we present the context to the CODIT through the token mask (see §3.9). If we enable copy attention, CODIT becomes highly biased by the tokens that are inside c_p.

Note that, *"vocabulary explosion"* still remains an open problem for code generation. Neither CODIT nor any other baselines we discussed here solve this problem. CODIT presents a way to learn the structural change and restricts the search domain for token names through a mask. For instance, where $\mathscr{M}_{token}$ needs to generate a token of *primitive data type* ($\mathscr{M}_{token}$ knows the token type because $\mathscr{M}_{tree}$ already generated a tree with terminal node types), it can restrict the search over the primitive types only. While it is expected that the decoder of an ideal Seq2Seq model would inherently learn appropriate tokens at appropriate positions as it implicitly learns code structure, in reality, because of its unrestricted search space, they tend to mispredict more tokens than CODIT.

For qualitative evaluation, we show some non-trivial patches that CODIT can successfully generate. CODIT can learn a wide range of patch patterns. Figure 3.2 shows few examples of different category of patches that CODIT can generate. CODIT also shows promise in generating non-trivial structure based changes. Consider Example 4 where `x` is renamed to `session` both the formal parameter and the usage in the body. Note that, since CODIT uses a tree-based model it is good at capturing long-distance dependencies allowing the token-level model to focus on predicting tokens, *e.g.,* such that it can rename the same variable similarly. Another interesting example where CODIT can successfully generate patches is shown in example 6, where CODIT does not only add the **`abstract`** keyword in the method signature, but also removes the body. Since CODIT is aware of code syntax, it learns that method declarations with an **`abstract`** keyword have a high probability of an empty method body.

Figure 3.3 shows some additional examples where CODIT can successfully generate patches. In these examples, different exception/error types (*i.e.* `Exception`, `Error`, `RuntimeException`) are changed to `EOFException` although their usage differs. In the first three examples `EOFException` is used as a class reference, while for the others `EOFException` is used to initialize an object. These examples also illustrate CODIT's ability to generalize to different contexts and use-cases. Other structural transformation that CODIT include, but not limited to, include scoping (example 7 in Figure 3.2), adding/deleting method parameters (example 3 in Figure 3.2), changing method/variable access modifiers (example 9, 10 in Figure 3.2), etc.

Result 3.1: CODIT *suggests 15.94% correct patches for Code-Change-Data and 28.87% for Pull-Request-Data within top 5 and outperforms best baseline by 44.37% and 9.26% respectively.*

Next, we evaluate CODIT's sub-components.

RQ-3.2. How do different design choices affect CODIT's performance?

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

Evaluating token generation model. Here we compare $\mathscr{M}_{token}$ with the baseline SequenceR model. Note that $\mathscr{M}_{token}$ generates a token given its structure. Thus, for evaluating the standalone token generation model in CODIT's framework, we assume that the true structure is given (emulating a scenario where a developer knows the kind of structural change they want to apply). Table 3.3 presents the results.

Table 3.3: Correct patches generated by the standalone token generation model when the true tree structure is known.

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

While the baseline Seq2Seq with copy-attention (SequenceR) generates 9.76% (502 out of 5143) and 26.43% (162 out of 613) correct patches for *Code-Change-Data* and *Pull-Request-Data* respectively at top 5, Table 3.3 shows that if the change structure (*i.e.* t_n) is known, the standalone $\mathscr{M}_{token}$ model of CODIT can generate 39.57% (2035 out of 6832) and 61.66% (378 out of 613) for *Code-Change-Data* and *Pull-Request-Data* respectively. This result empirically shows that if the tree structure is known, NMT-based code change prediction significantly improves. In fact, this observation led us build CODIT as a two-stage model.

Evaluating tree translation model. Here we evaluate how accurately $\mathscr{M}_{tree}$ predicts the structure

Table 3.4: $\mathscr{M}_{tree}$ top-5 performance for different settings.

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

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

of a change — shown in Table 3.4. $\mathscr{M}_{tree}$ can predict 56.78% and 55.79% of the structural changes in *Code-Change-Data* and *Pull-Request-Data* respectively. Note that, the outputs that are generated by $\mathscr{M}_{tree}$ are not concrete code, rather structural abstractions. In contrast, Tufano *et al.*'s [224] predicts 38.56% and 39.31% correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. Note that, their abstraction and our abstraction method is completely different. In their case, for some of the identifiers (those already exist in the symbol table), they have a deterministic way to concretize the code. In our case, $\mathscr{M}_{token}$ in CODIT augments $\mathscr{M}_{tree}$ by providing a stochastic way to concretize every identifier by using NMT.

Note that, not all patches contain structural changes (*e.g.,* when a single token, such as a method name, is changed). For example, 3050 test patches of *Code-Change-Data*, and 225 test patches of *Pull-Request-Data* do not have structural changes. When we use these patches to train $\mathscr{M}_{tree}$, we essentially train the model to sometimes copy the input to the output and rewarding the loss function for predicting no transformation. Thus, to report the capability of $\mathscr{M}_{tree}$ to predict structural changes, we also train a separate version of $\mathscr{M}_{tree}$ using only the training patches with at least 1 node differing between t_n and t_p. We also remove examples with no structural changes from the test set. This is our filtered dataset. In the filtered dataset, $\mathscr{M}_{tree}$ predicts 33.61% and 30.73% edited structures from *Code-Change-Data* and *Pull-Request-Data* respectively. This gives us an estimate of how well $\mathscr{M}_{tree}$ can predict *structural* changes.

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

First, we evaluate contribution of CODIT's attention-based copy mechanism. Table 3.5 shows the results. Note that, unlike SequenceR, CODIT is not trained for learning to copy. Our copy mechanism is an auxiliary step in the beam search that prohibits occurrence of `<unknown>` token in the generated code.

Table 3.5: CODIT performance *w.r.t.* to the attention based copy mechanism @top-5 (K_{tree} =2, K_{token} =5). Lower bound is without copy. The upper bound evaluates with oracle copying predictions for `<unknown>`. For CODIT each `<unknown>` token is replaced by the source token with the highest attention.

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

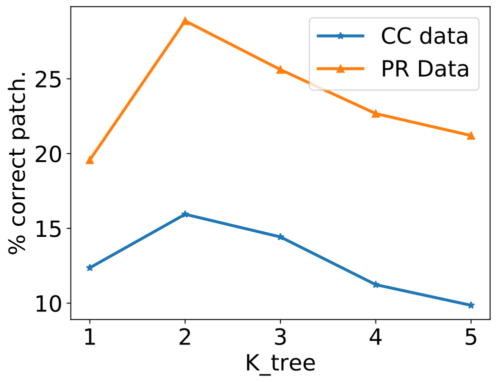
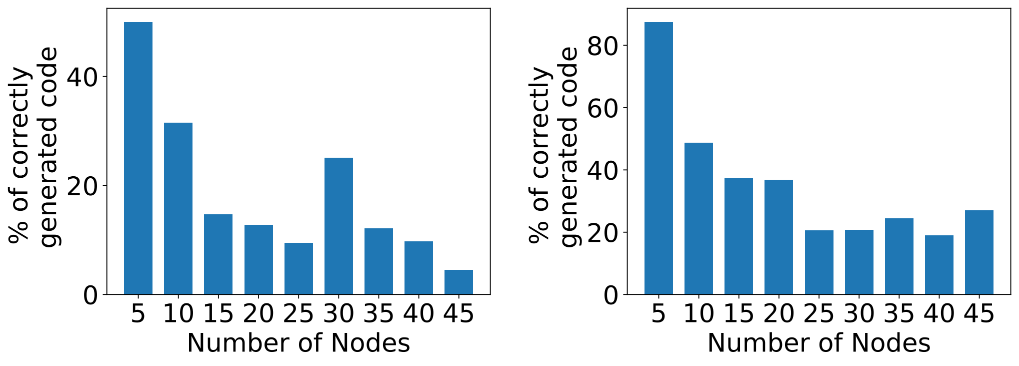


Figure 3.4: Performance of CODIT @top-5 (K_{token} = 5) for different K_{tree}

Recall that $\mathscr{M}_{token}$ generates a probability distribution over the vocabulary. Since the vocabulary is generated using the training data, any unseen tokens in the test patches are replaced by a special `<unknown>` token. In our experiment, we found that a large number (about 3% is *Code-Change-Data* and about 4% is *Pull-Request-Data*) of patches contain `<unknown>` tokens; this is undesirable since the generated code will not compile. When we do not replace `<unknown>` tokens, CODIT can predict 742 (14.42%), and 163 (26.59%) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. However, if all the `<unknown>` tokens could be replaced perfectly with the intended token, *i.e.* upper bound of the number of correct patches goes up to 898 (17.46%) and 191 (31.16%) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. This shows the need for tackling the `<unknown>` token problem. To solve this, we replace <unknown> tokens predicted by $\mathscr{M}_{token}$ with the source token with the highest attention probability following Section 3.2.2. With this, CODIT generates 820 (15.94%), and 177 (28.87%) correct patches from *Code-Change-Data* and *Pull-Request-Data* respectively (Table 3.5).

Second, we test two configuration parameters related to the beam size, K_{tree} and K_{token} *i.e.* the number of trees generated by $\mathscr{M}_{tree}$ and number of concrete token sequences generated by $\mathscr{M}_{token}$ per tree (Section 3.2). While Table 3.2 shows the effect of different values of K_{token} (*e.g.*,, 1, 2, 5), here we investigate the effect of K_{tree} for a given K_{token}. Figure 3.4 shows the parameter sensitivity of K_{tree} when K_{token} is set 5. Recall from Section 3.2, CODIT first generates K_{tree} number of trees, and then generates K_{token} number of code per tree. Among those $K_{tree} * K_{token}$ generated code, CODIT chooses top K_{token} number of code to as final output. In both *Code-Change-Data* (CC data in figure 3.4), and *Pull-Request-Data* (PR data in figure 3.4), CODIT performs best when K_{tree} = 2. When K_{tree} = 2, total generated code is 10, among which CODIT chooses top 5. With increasing number of K_{tree}, CODIT has to choose from increasing number of generated code, eventually hurting the performance of CODIT.



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

Figure 3.5: Percentage of correct prediction by CODIT with respect to number of nodes in the tree.

Next we evaluate CODIT's ability to generate correct patches *w.r.t.* the tree size. Figure 3.5 shows histogram of percentage of correctly predicted examples by CODIT *w.r.t.* size of the tree (in terms of nodes). While CODIT performs well in predicting smaller trees ($\leq$ 10 nodes), CODIT also works comparably well in larger tree sizes. In fact, CODIT produces 21.97% correct code in

Pull-Request-Data, and 16.48% correct code in *Code-Change-Data* where the tree size is larger than 30 nodes.

Table 3.6: Cross project generalization test for CODIT (% of correct at @top-5).

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

To understand how CODIT generalizes beyond a project, we do a cross-project generalization test. Instead of chronological split of examples (see section 3.3), we split the examples based on projects, *i.e.*, all the examples that belongs to a project falls into only one split (train/validation/test). We then train and test CODIT based on this data split. Table 3.6 shows the result in this settings *w.r.t.* to intra-project split. While $\mathscr{M}_{tree}$ in intra-project and cross-project evaluation setting achieves similar performance, full CODIT performance deteriorate by 68%. The main reason behind such deterioration is diverse choice of token name across different projects. Developer tend to use project specific naming convention, api etc. This also indicates that the structural change pattern that developers follow are more ubiquitous across different projects than the token changes.

Result 3.2: CODIT *yields the best performance with a copy-based attention mechanism and with tree beam size of 2.* $\mathscr{M}_{tree}$ *achieves 58.78% and 55.79% accuracy and* $\mathscr{M}_{token}$ *achieves 39.57% and 61.66% accuracy in Code-Change-Data and Pull-Request-Data respectively when tested individually.*

Finally, we evaluate CODIT's ability to fixing bugs.

RQ-3.3. How accurately CODIT suggests bug-fix patches?

We evaluate this RQ with the state-of-the-art bug-repair dataset, Defects4J [104].

Training: We collect commits from the projects' original GitHub repositories and preprocess them as described in Section 3.2. We further remove the Defects4J bug fix patches and use the rest of the patches to train and validate CODIT.

Testing: We extract the methods corresponding to the bug location(s) from the buggy-versions of Defects4J. A bug can have fixes across multiple methods. We consider each method as candidates for testing and extract their ASTs. We then filter out the methods that are not within our accepted tree sizes. In this way, we get 117 buggy method ASTs corresponding to 80 bugs. The rest of the bugs are ignored.

Here we assume that a fault-localization technique already localizes the bug [1]. In general, fault-localization is an integral part of program repair. However, in this paper, we focus on evaluating CODIT's ability to produce patches rather than an end-to-end repair tool. Since fault localization and fixing are methodologically independent, we assume that bug location is given and evaluate whether CODIT can produce the correct patch. Evaluation of CODIT's promise as a full-fledged bug repair tool remains for future work.

For a buggy method, we extract c_p. Then for a given c_p, we run CODIT and generate a ranked list of generated code fragments (c_n). We then try to patch the buggy code with the generated fragments following the rank order, until the bug-triggering test passes. If the test case passes, we mark it a potential patch and recommend it to developers. We set a specific time budget for the patch generation and testing. For qualitative evaluation, we additionally investigate manually the patches that pass the triggering test cases to evaluate the semantic equivalence with the developer-provided patches. Here we set the maximum time budget for each buggy method to 1 hour. We believe this is a reasonable threshold as previous repair tools (*e.g.,* Elixir [206]) set 90 minutes for generating patches. SimFix [100] set 5 hours as their time out for generating patches and running test cases.

CODIT can successfully generate at least 1 patch that passes the bug-triggering test case for 51 methods out of 117 buggy methods from 30 bugs, *i.e.* 43.59% buggy methods are potentially fixed. Figure 3.6 shows the number of patches passing the bug-triggering test case *w.r.t.* time. We see that, 48 out of 51 successful patches are generated within 20 minutes.

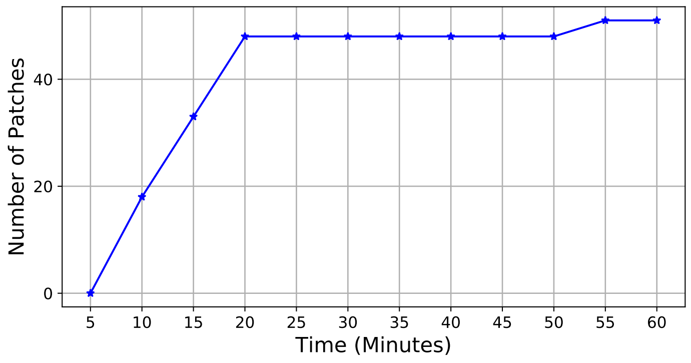


Figure 3.6: Patches passing the bug-triggering tests *v.s.* time.

We further *manually* analyze the generated patches with the developer-provided patches: among 51 potential patches, 30 patches are identical and come from 25 different bug ids (See Table 3.7). The bugs marked in green are completely fixed by CODIT with all their buggy methods being successfully fixed. For example, Math-49 has 4 buggy methods, CODIT fixes all four. For the bugs marked in blue†, CODIT fixes all the methods that are in scope. For example, for Lang-4, there are 2 methods to be fixed, 1 of them are in CODIT's scope, and CODIT fixes that. However, for two other bugs (marked in orange*), CODIT produces only a partial fix. For example, in the case of Math-46 and Mockito-6, although all the methods are within scope, CODIT could fix 1 out of 2 and 2 out of 20 methods respectively. The 'Patch Type' column further shows the type of change patterns.

SequenceR [43] is a notable NMT based program repair tool which takes the advantage of *learning to copy* in NMT. They evaluated on 75 one line bugs in Defects4J dataset and reported 19 plausible and 14 fully correct successful patches. Among those 75 bugs, 38 are in CODIT's scope. Out of those 38 bugs, CODIT can successfully generate patches for 14 bugs. Note that, we do not present CODIT as full fledged automated program repair tool, rather a tool for guiding developers. Thus, for automatic evaluation, we assumed the correct values of constants (of any data type) given.

One prominent bug repair approach [186, 206, 111] is to transform a suspicious program element following some change patterns until a patch that passes the test cases is found. For instance, Elixir [206] used 8 predefined code transformation patterns and applied those. In fact, CODIT can

Table 3.7: CODIT's performance on fixing Defects4J [104] bugs.

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

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

generate fixes for 8 bugs out of 26 bugs that are fixed by Elixir [206].

Nevertheless, CODIT can be viewed as a transformation schema which automatically learns these patterns without human guidance. We note that CODIT is *not* explicitly focused on bug-fix changes since it is trained with generic changes. Even then, CODIT achieves good performance in Defects4J bugs. Thus, we believe CODIT has the potential to complement existing program repair tools by customizing the training with previous bug-fix patches and allowing to learn from larger change sizes. Note that, current version of CODIT does not handle multi-hunk bugs. Even if a bug is multi-hunk, in current prototype, we consider each of the hunk as separate input to CODIT. For instance, consider Math-46, which is a 2-hunk bug. While all 2 methods are in CODIT's scope, CODIT can only fix one. Currently we do not consider interaction between multiple hunks [207]. We leave the investigation of NMT in such scenario for future work.

Result 3.3: CODIT *generates complete bug-fix patches for 15 bugs and partial patches for 10 bugs in Defects4J.*

3.4.1 Discussion & Threats to validity

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

Most NMT based model (or any other text decoder based models) faces the "vocabulary explosion" problem. That problem is even more prominent in code modeling, since possible names of identifiers can be virtually infinite. While this problem is a major bottleneck in DL base code generation, CODIT does not solve this problem. In fact, similar to previous researches (*i.e.*, SequenceR [43]), CODIT cannnot generate new identifiers if it is not in the vocabulary or in the input code.

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

Search Space for Code Generation. Synthesizing patches (or code in general) is challenging [64]. When we view code generation as a sequence of token generation problem, the space of the possible actions becomes too large. Existing statistical language modeling techniques endorse the action space with a probability distribution, which effectively reduces the action space significantly since it allows to consider only the subset of probable actions. The action space can be further reduced by relaxing the problem of concrete code generation to some form of abstract code generation, *e.g.,* generating code sketches [162], abstracting token names [224], *etc*. For example, Tufano *et al.* reduce the effective size of the action space to $3.53 \cdot 10^{10}$ by considering abstract token names [224]. While considering all possible ASTs allowed by the language's grammar, the space size grows to $1.81 \cdot 10^{35}$. In this work, a probabilistic grammar further reduces the effective action space to $3.18 \cdot 10^{10}$, which is significantly lower than previous methods. Such reduction of the action space allows us to search for code more efficiently.

Ensemble Learning for Program Repair. The overall performance of pre-trained deep-learning models may vary due to the different model architectures and hyper-parameters, even if they are trained on the same training corpus. Moreover, bug fixing patterns are numerous and highly dependent on the bug context and the bug type, so a single pre-trained model may only have the power to fix certain kinds of bugs and miss the others. To overcome this limitation, ensemble learning can be a potential approach to leverage the capacities of different models and learn the fixing patterns in multiple aspects [145].

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

3.5 Concluding Remarks

In this chapter, we proposed and evaluated CODIT, a tree-based hierarchical model for suggesting eminent source code changes. CODIT's objective is to suggest changes similar to change patterns observed in the wild. We evaluate our work against a large number of real-world patches. The results indicate that tree-based models are promising for generating code patches and can outperform popular seq2seq alternatives. We also apply our model to program repair tasks, and the experiments show that CODIT is capable of predicting bug fixes as well. While in this chapter, we have seen an example of explicitly encoding PL properties (*i.e.* syntax) into the model, in the next chapter, we will explore explicit encoding of semantic properties into the model and the use of such in program understanding.

Chapter 4: Program Understanding through Explicit Program Encoding

4.1 Motivation

Automated detection of security vulnerabilities is a fundamental problem in systems security. Traditional techniques are known to suffer from high false-positive/false-negative rates [102, 211]. For example, static analysis-based tools typically result in high false positives detecting non-vulnerable (hereafter, neutral) [1] cases as vulnerable, and dynamic analysis suffers from high false negatives. So far these tools remain unreliable, leaving significant overhead for developers [211]. Recent progress in Deep Learning (DL), especially in domains like computer vision and natural language processing, has sparked interest in using DL to detect security vulnerabilities automatically with high accuracy. According to Google scholar, 92 papers appeared in popular security and software engineering venues between 2019 and 2020 that apply learning techniques to detect different types of bugs[2]. In fact, several recent studies have demonstrated very promising results achieving accuracy up to 95% [132, 131, 204, 250].

Given such remarkable reported success of DL models at detecting vulnerabilities, it is natural to ask why they are performing so well, what kind of features these models are learning, and whether they are generalizable, *i.e.*, can they be used to reliably detect real-world vulnerabilities?

The generalizability of a DL model is often limited by implicit biases in the dataset, which are often introduced during the dataset generation/curation/labeling process and therefore affect both the testing and training data equally (assuming that they are drawn from the same dataset). These biases tend to allow DL models to achieve high accuracy in the test data by learning highly idiosyn-

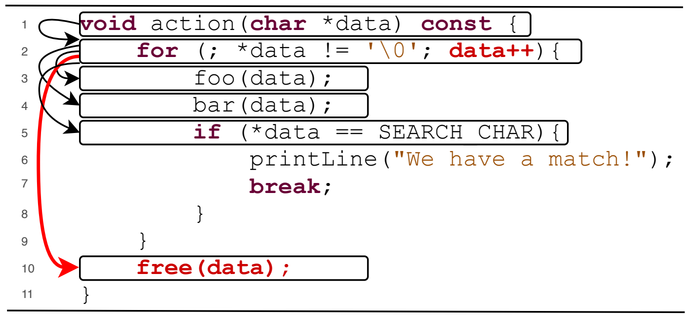
[1] We prefer to refer to non-vulnerable code as "neutral" to indicate that they contain no *known* vulnerabilities or that they do not fall in any known vulnerability category.

[2] published in TSE, ICSE, FSE, ASE, S&P Oakland, CCS, USENIX Security, etc.

cratic features specific to that dataset instead of generalizable features. For example, Yudkowsky et al. [247] described an instance where US Army found out that a neural network for detecting camouflaged tanks did not generalize well due to dataset bias even though the model achieved very high accuracy in the testing data. They found that all the photos with the camouflaged tanks in the dataset were shot in cloudy days, and the model simply learned to classify lighter and darker images instead of detecting tanks.

Our in depth investigation revealed that none of the existing models perform well in real-world settings. If we directly use a pre-trained model to detect the real-world vulnerabilities, the performance drops by ∼*73%*, on average. Even if we retrain these models with real-world data, their performance drops by ∼*54%* from the reported results. For example, VulDeePecker [132] reported a precision of 86.9% in their paper. However, when we use VulDeePecker's pre-trained model in real world datasets, its precision reduced to 11.12%, and after retraining, the precision becomes 17.68%. A thorough investigation revealed the following problems:

- *Inadequate Model.* The most popular models treat code as a sequence of tokens and do not take into account semantic dependencies that play a vital role in vulnerability predictions. Token based models assume that tokens are linearly dependent on each other, and



```
void action(char *data) const {
    for (; *data != '\0'; data++){
        foo(data);
        bar(data);
        if (*data == SEARCH_CHAR){
            printLine("We have a match!");
            break;
        }
    }
    free(data);
}
```

Figure 4.1: Example of CWE-761 [50]. A buffer is freed not at the start of the buffer but somewhere in the middle of the buffer. This can cause the application to crash, or in some cases, modify critical program variables or execute code. This vulnerability can be detected with data dependency.

thus, only lexical dependencies between the tokens are present, while the semantic dependencies are lost, which often play important roles in vulnerability detection [53].To incorporate some semantic information, VulDeePecker [132] and SySeVR [131] extracted program slices of a potentially interesting point. For example, consider the code in Figure 4.1. A slice *w.r.t.*`free` function call at line 10 gives us all the lines except lines 6 and 7. The token sequence of the slice are: `void action ( char * data )const { for ( data ; * data != '0'; data ++ ){ foo ( data ); bar ( data ); if ( * data == SEARCH_CHAR ){ free ( data );` . In this examples, while the two main components for this code being vulnerable, *i.e.* **`data++`** (line 2) and **`free(data)`** (line 10) are present in the token sequence, they are far apart from each other without explicitly maintaining any dependencies.

In contrast, as a graph based model can consider the data dependency edges (red edge), we see that there is a direct edge between those lines making those lines closer to each other making it easier for the model to reason about that connection. Note that this is a simple CWE example (CWE 761), which requires only the data dependency graph to reason about. Real-world vulnerabilities are much more complex and require reasoning about control flow, data flow, dominance relationship, and other kinds of dependencies between code elements [242]. However, graph-based models, in general, are much more expensive than their token-based counterparts and do not perform well in a resource-constrained environment.

- *Learning Irrelevant Features.* Using state-of-the-art explanation techniques [79], we find that the current models are essentially learning up irrelevant features that are not related to vulnerabilities and are likely artifacts of the datasets. We investigate the feature that the existing techniques are picking up from pre-existing datasets leveraging state-of-the-art explanation techniques [212, 79]. To our surprise, models that exhibit outstanding performance in pre-existing datasets are picking up irrelevant features. These features are not related to vulnerabilities in any way and are likely to be artifacts of those datasets.

To overcome these problems, we propose a road-map that we hope will help the DL-based vulnerability prediction researchers to avoid such pitfalls in the future. We build a graph based program representation and vulnerability detection tool, ReVeal. We further empirically establish that the use of semantic information (with graph-based models) improve vulnerability detection. Following these steps, we can boost precision and recall of the best performing model in the literature by up to 33.57% and 128.38% respectively.

4.2 Methodology

4.2.1 DL-based Vulnerability Detection

DL-based vulnerability predictors learn the vulnerable code patterns from a training data (D_{train}) set where code elements are labeled as vulnerable or neutral. Given a code element (x) and corresponding vulnerable/neutral label (y), the goal of the model is to learn features that maximize the probability $p(y|x)$ with respect to the model parameters (θ). Formally, training a model is learning the optimal parameter settings (θ^*) such that,

$$\theta^* = argmax_\theta \prod_{(x,y)\in D_{train}} p(y|x,\theta) \tag{4.1}$$

First, a code element (x^i) is transformed to a real valued vector ($h^i \in \mathbb{R}^n$), which is a compact representation of x^i. How a model transforms x^i to h^i depends on the specifics of the model. This h^i is transformed to a scalar $\hat{y} \in [0,1]$ which denotes the probability of code element x^i being vulnerable. In general, this transformation and probability calculation is achieved through a feed forward layer and a softmax [30] layer in the model. Typically, for binary classification task like Vulnerability Detection, optimal model parameters are learned by minimizing the cross-entropy loss [215]. Cross-entropy loss penalizes the discrepancy in the model's predicted probability and the actual probability (0. for neutral 1. for vulnerable examples) [182].

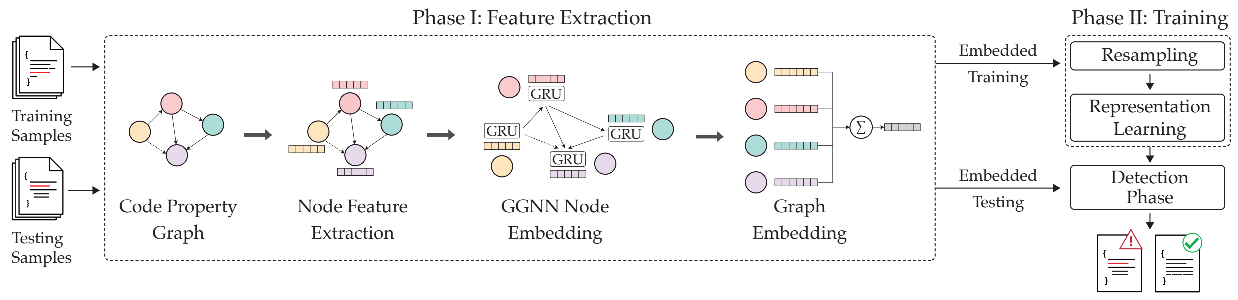


Figure 4.2: Overview of the ReVeal vulnerability prediction framework.

4.2.2 Graph Based Representation Learning

We now present a brief overview of the ReVeal pipeline that aims to lay a *roadmap* for accurately detecting vulnerabilities in real-world projects. Figure 4.2 illustrates the ReVeal pipeline. It operates in two phases namely, feature extraction (Phase-I) and training (Phase-II). In the first phase we translate real-world code into a graph-embedding. In the second phase, we train a representation learner on the extracted features to learn a representation that most ideally demarcates the vulnerable examples from neutral.

Feature Extraction (Phase-I)

The goal of this phase is to convert code into a compact and a uniform length feature vector while maintaining the semantic and syntactic information. Our proposed road map extracts a feature vector using a graphical representation of code. Note that, the feature extraction scheme presented below represents the most commonly used series of steps for extracting features from a graph representation [250]. ReVeal uses this scheme to extract the graph embedding of each function in code (graph based feature vector that represent the entirety of a function in a code).

To extract the syntax and semantics in the code, we generate a code property graph (hereafter, CPG) [242]. The CPG is a particularly useful representation of the original code since it offers a combined and a succinct representation of the code consisting of elements from the control-flow and data-flow graph in addition to the AST and program dependency graph (or PDG). Each of the above elements offer additional context about the overall semantic structure of the code [242].

Formally, a CPG is denoted as $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, where V represent the vertices (or nodes) in the graph and E represents the edges. Each vertex V in the CPG is comprised of the vertex type (*e.g.,* `ArithmeticExpression, CallStatement` etc.) and a fragment of the original code. To encode the type information, we use a one-hot encoding vector denoted by T_v. To encode the code fragment in the vertex, we use a word2vec embedding denoted by C_v. Next, to create the vertex embedding, we concatenate T_v and C_v into a joint vector notation for each vertex.

The current vertex embedding is not adequate since it considers each vertex in isolation. It therefore lacks information about its adjacent vertices and, as a result, the overall graph structure. This may be addressed by ensuring that each vertex embedding reflects both its information and those of its neighbors. We use gated graph neural networks (hereafter GGNN) [130] for this purpose. Feature vectors for all the nodes in the graph (X) along with the edges (E) are the input to the GGNN [130, 251]. For every vertex in the CPG, GGNN assigns a gated recurring unit (GRU) that updates the current vertex embedding by assimilating the embedding of all its neighbors. Formally,

$$x'_v = GRU(x_v, \sum_{(u,v)\in \mathrm{E}} g(x_u))$$

Where, $GRU(\cdot)$ is a Gated Recurrent Function, x_v is the embedding of the current vertex v, and $g(\cdot)$ is a transformation function that assimilates the embeddings of all of vertex v's neighbors [249, 130, 12]. x'_v is the GGNN-transformed representation of the vertex v's original embedding x_v. x'_v now incorporates v's original embedding x_v as well as the embedding of its neighbors.

The final step in preprocessing is to aggregate all the vertex embedding x'_v to create a single vector representing the whole CPG denoted by x_g, *i.e.*:

$$x_g = \sum_{v\in \mathrm{V}} x'_v$$

Note that ReVeal uses a simple element-wise summation as the aggregation function, but in practice it is a configurable parameter in the pipeline. The result of the pipeline presented so far is an $m-$dimensional feature vector representation of the original source code. To pre-train the GGNN, we augment a classification layer on top of the GGNN feature extraction. This training mechanism is similar to Devign [250]. Such pre-training deconstructs the task of "learning code representation", and "learning vulnerability", and is also used by Russell *et al.* [204]. While, we pre-train GGNN in a supervised fashion, unsupervised program representation learning [122] can also be done to learn better program presentation. However, such learning is beyond the scope of this research and we leave that for future research.

Training (Phase-II)

In real-world data, the number of neutral samples (*i.e.* negative examples) far outnumbers the vulnerable examples (*i.e.* positive examples) . If left unaddressed, this introduces an undesirable bias in the model limiting its predictive performance. Further, extracted feature vectors of the vulnerable and neutral examples exhibit a significant overlap in the feature space. This makes it difficult to demarcate the vulnerable examples from the neutral ones. Training a DL model without accounting for the overlap makes it susceptible to poor predictive performance. To mitigate the above problems, we propose a two step approach. First, we use re-sampling to balance the ratio of vulnerable and neutral examples in the training data. Next, we train a representation learning model on the re-balanced data to learn a representation that can most optimally distinguish vulnerable and neutral examples.

Reducing Class Imbalance In order to handle imbalance in the number of vulnerable and neutral classes, we use the "synthetic minority over-sampling technique" (for short, SMOTE) [40]. It operates by changing the frequency of the different classes in the data. Specifically, SMOTE sub-samples the majority class (i.e., randomly deleting some examples) while super-sampling the minority class (by creating synthetic examples) until all classes have the same frequency. In the case of vulnerability prediction, the minority class is usually the vulnerable examples. SMOTE has shown to be effective in a number of domains with imbalanced datasets [217, 144]. During super-sampling, SMOTE picks a vulnerable example and finds k nearest vulnerable neighbors. It then builds a synthetic member of the minority class by interpolating between itself and one of its random nearest neighbors. During under-sampling, SMOTE randomly removes neutral examples from the training set. This process is repeated until a balance is reached between the vulnerable and neutral examples. Note that, while we use off-the-shelf SMOTE for re-balancing training data, other data re-balancing technique (*e.g.,* MWMOTE [27], ProWSyn [26]). Nevertheless, SMOTE as a re-balancing module in ReVeal's pipeline is configurable and can easily be replaced by other re-balancing techniques. Comparison between different re-balancing techniques themselves is beyond the scope of this research.

Representation Learning Model The graph embedding of the vulnerable and neutral code samples at the end of Phase-I tend to exhibit a high degree of overlap in feature space. This makes the models "brittle". To improve the predictive performance, we seek a model that can project the features from the original non-separable space into a latent space which offers a better separability between vulnerable and neutral samples. For this, we use a multi-layer perceptron (MLP) [215], designed to transform input feature vector (x_g) to a latent representation denoted by $h(x_g)$. The MLP consists of three groups of layers namely, the input layer (x_g), a set of intermediate layers which are parameterized by θ (denoted by $f(\cdot, \theta)$, and a final output layer denoted by $\hat{y}$.

The proposed representation learner works by taking as input the original graph embedding x_g and passing it through the intermediate layers $f(\cdot, \theta)$. The intermediate layer project the original graph embedding x_g onto a latent space $h(x_g)$. Finally, the output layer uses the features in the latent space to predict for vulnerabilities as, $\hat{y} = \sigma\left(\mathrm{W} * h(x_g) + b\right)$. Where σ represents the softmax function, h_g is the latent representation, W and b represent the model weights and bias respectively.

To maximize the separation between the vulnerable and the neutral examples in the latent space, we adopt the triplet loss [148] as our loss function. Triplet loss has been widely used in machine learning, specifically in representation learning, to create a maximal separation between classes [90, 229]. The triplet loss is comprised of three individual loss functions: (a) cross entropy loss ($\mathcal{L}_{CE}$); (b) projection loss ($\mathcal{L}_p$); and (c) regularization loss ($\mathcal{L}_{reg}$). It is given by:

$$\mathcal{L}_{trp} = \mathcal{L}_{CE} + \alpha * \mathcal{L}_p + \beta * \mathcal{L}_{reg} \tag{4.2}$$

α and β are two hyperparameters indicating the contribution of projection loss and regularization loss respectively. The first component of the triplet loss is to measure the cross-entropy loss to penalize miss-classifications. Cross-entropy loss increases as the predicted probability diverges from the actual label. It is given by,

$$\mathcal{L}_{CE} = -\sum \hat{y} \cdot log(y) + (1 - \hat{y}) \cdot log(1 - y) \tag{4.3}$$

Here, y is the true label and $\hat{y}$ represents the predicted label.The second component of the triplet loss is used the quantify how well the latent representation can separate the vulnerable and neutral examples. A latent representation is considered useful if all the vulnerable examples in the latent space are close to each other while simultaneous being farther away from all the neutral examples, *i.e.* examples from same class are very close (*i.e.* similar) to each other and examples from different class are far away from each other. Accordingly, we define a loss function $\mathcal{L}_p$ which is defined by.

$$\mathcal{L}_p = \left| \mathbb{D}(h(x_g), h(x_{same})) - \mathbb{D}(h(x_g), h(x_{diff})) + \gamma \right| \tag{4.4}$$

Here, $h(x_{same})$ is the latent representation of an example that belongs to the same class as x_g and $h(x_{diff})$ is the latent representation of an example that belongs to a different class as that of x_g. Further, γ is a hyperparameter used to define a minimum separation boundary. Lastly, $\mathbb{D}(\cdot)$ represents the cosine distance between two vectors and is given by,

$$\mathbb{D}(v_1, v_2) = 1 - \left| \frac{v_1.v_2}{||v_1|| * ||v_2||} \right| \tag{4.5}$$

If the distance between two examples that belong to the same class is large (*i.e.* $\mathbb{D}(h(x_g), h(x_{same}))$

Table 4.1: Summary Statistics of the ReVeal dataset. '25 Q.' and '75 Q.' represents 25th quantile and 75th quantile, respectively. All the statistics are on function granularity.

Summary Statistic	Count				
	Min.	25 Q.	Median	75 Q.	Max.
# of lines of code	2	7	15	33	915
# of tokens	7	56	121	274	8830
# of nodes	2	9	18	38	499
# of edges	5	41	87	198	8257
Total Number of Examples				18169	
Number of Vulnerable Examples				1658	

is large) or if the distance between two examples that belong to different classes is small (*i.e.* $\mathbb{D}(h(x_g), h(x_{diff}))$ is small), $\mathcal{L}_p$ would be large to indicate a sub-optimal representation.

The final component of the triplet loss is the regularization loss ($\mathcal{L}_{reg}$) that is used to limit the magnitude of latent representation ($h(x_g)$). It has been observed that, over several iterations, the latent representation $h(x_g)$ of the input x_g tend to increase in magnitude arbitrarily [209]. Such arbitrary increase in $h(x_g)$ prevents the model from converging [71]. Therefore, we use a regularization loss ($\mathcal{L}_{reg}$) to penalize latent representations ($h(x_g)$) that are larger in magnitude. The regularization loss is given by:

$$\mathcal{L}_{reg} = ||h(x_g)|| + ||h(x_{same})|| + ||h(x_{diff})|| \tag{4.6}$$

With the triplet loss function, ReVeal trains the model to optimize for it parameters (*i.e.* θ, W, b) by minimizing equation 4.2.

4.3 Experimental Design

We evaluate the existing methods (*i.e.*, VulDeePecker [132], SySeVR [131], Russell *et al.* [204], and Devign [250]) and ReVeal's performance on two real world datasets. First dataset, ReVeal-data which we collected from issue-trackers of Chromium and Debian projects (see Table 4.1 for statistics on this dataset). Second dataset is vulnerabilities collected from FFMPeg and Qemu project proposed by Zhou *et al.* [250].

To understand a model's performance, researchers and model developers need to understand the performance of a model against a known set of examples.

Problem Formulation and Evaluation Metric: Most of the approaches formulate the problem as a classification problem, where given a code example, the model will provide a binary prediction indicating whether the code is vulnerable or not. This prediction formulation relies on the fact that there are sufficient number of examples (both vulnerable and neutral) to train on. In this study, we are focusing on the similar formulation. While both VulDeePecker and SySeVR formulate the problem as classification of code slices, we followed the problem formulation used by Russell *et al.* [204], and Devign [251], where we classify the function. We note that slices are paths in the control/data flow/dependency graphs, and a slice lacks the rich connectivity of nodes that is present in the whole graph. Thus we chose to classify the whole graph in contrast to the slices.

We study approaches based on four popular evaluation metrics for classification task [184] – Accuracy, Precision, Recall, and F1-score. Precision, also known as Positive Predictive rate, is calculated as *true positive / (true positive + false positive)*, indicates correctness of predicted vulnerable samples. Recall, on the other hand, indicates the effectiveness of vulnerability prediction and is calculated as *true positive / (true positive + false negative)*. F1-score is defined as the geometric mean of precision and recall and indicates balance between those.

Evaluation Procedure: Since DL models highly depend on the randomness [28], to remove any bias created due to the randomness, we run 30 trials of the same experiment. At every run, we randomly split the dataset into disjoint train, validation, and test sets with 70%, 10%, and 20% of the dataset respectively. We report the median performance and the inter-quartile range (IQR) of the performance. When comparing the results to baselines, we use statistical significance test [118] and effect size test [87]. Significance test tells us whether two series of samples differ merely by random noises. Effect sizes tells us whether two series of samples differ by more than just a trivial amount. To assert statistically sound comparisons, following previous approaches [2, 99], we use a non-parametric bootstrap hypothesis test [103] in conjunction with the A12 effect size test [20].

We distinguish results from different experiments if both significance test and effect size test agreed that the division was statistically significant (99% confidence) and is not result of a “small” effect (A12 ≥ 60%) (similar to Agrawal *et al.* [2]).

Table 4.2: Performance of different models in Vulnerability Detection. All the numbers are reported as *Median (IQR)* format.

Dataset	Input	Approach	Accuracy	Precision	Recall	F1-score
ReVeal dataset	Token	Russell *et al.*	90.98 *(0.75)*	24.63 *(5.35)*	10.91 *(2.47)*	15.24 *(2.74)*
	Slice + Token	VulDeePecker	89.05 *(0.80)*	17.68 *(7.51)*	13.87 *(8.53)*	15.7 *(6.41)*
		SySeVR	84.22 *(2.48)*	24.46 *(4.85)*	40.11 *(4.71)*	30.25 *(2.35)*
	Graph	Devign	88.41 *(0.66)*	34.61 *(3.24)*	26.67 *(6.01)*	29.87 *(4.34)*
		ReVeal	84.37 *(1.73)*	30.91 *(2.76)*	60.91 *(7.89)*	41.25 *(2.28)*
FFMpeg + Qemu	Token	Russell *et al.*	58.13 *(0.88)*	54.04 *(2.09)*	39.50 *(2.17)*	45.62 *(1.33)*
	Slice + Token	VulDeePecker	53.58 *(0.61)*	47.36 *(1.80)*	28.70 *(12.08)*	35.20 *(8.82)*
		SySeVR	52.52 *(0.81)*	48.34 *(1.51)*	65.96 *(7.12)*	56.03 *(3.20)*
	Graph	Devign	58.57 *(1.03)*	53.60 *(3.21)*	62.73 *(2.99)*	57.18 *(2.58)*
		ReVeal	62.51 *(0.90)*	56.85 *(1.54)*	74.61 *(4.31)*	64.42 *(1.33)*

4.4 Research Findings

Form the context of this dissertation, in this work, we investigate two major research questions. First, we investigate the effect of Graph-based models to semantic understanding of source code. Thus, we ask,

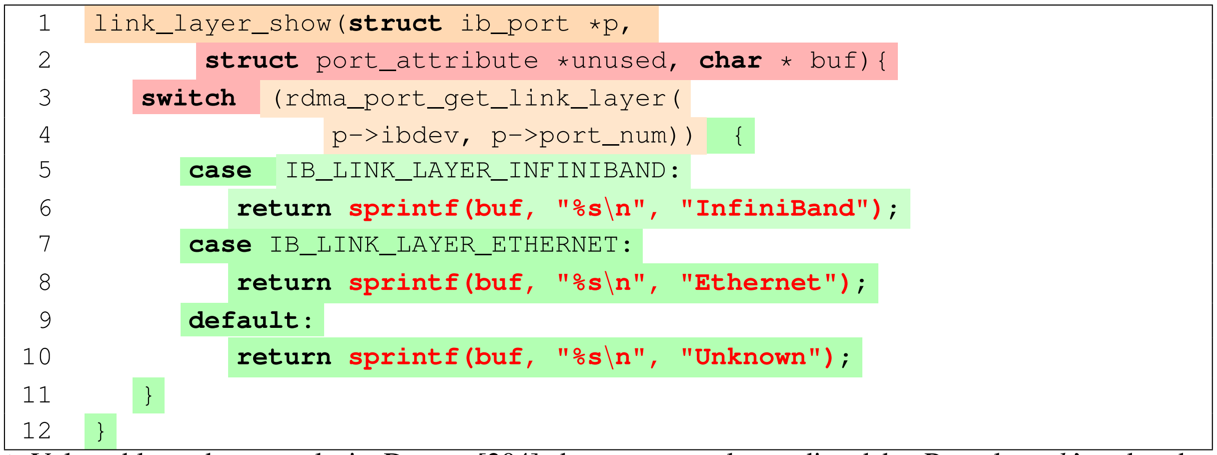
Table 4.3: Impact of GGNN in ReVeal's performance [Median (IQR)].

Dataset	Approach	Accuracy	Precision	Recall	F1-score
ReVeal dataset	ReVeal w/o GGNN	83.69 *(1.60)*	29.48 *(2.69)*	57.69 *(7.85)*	39.09 *(2.30)*
	ReVeal with GGNN	**84.37** ***(1.73)***	30.91 *(2.76)*	**60.91** ***(7.89)***	**41.25** ***(2.28)***
FFMpeg+ Qemu	ReVeal w/o GGNN	53.87 *(2.69)*	49.60 *(1.68)*	**89.25** ***(3.78)***	63.69 *(0.85)*
	ReVeal with GGNN	**62.51** ***(0.90)***	**56.85** ***(1.54)***	74.61 *(4.31)*	**64.42** ***(1.33)***

RQ-4.1. Does explicit encoding of PL properties through help better understand source code?

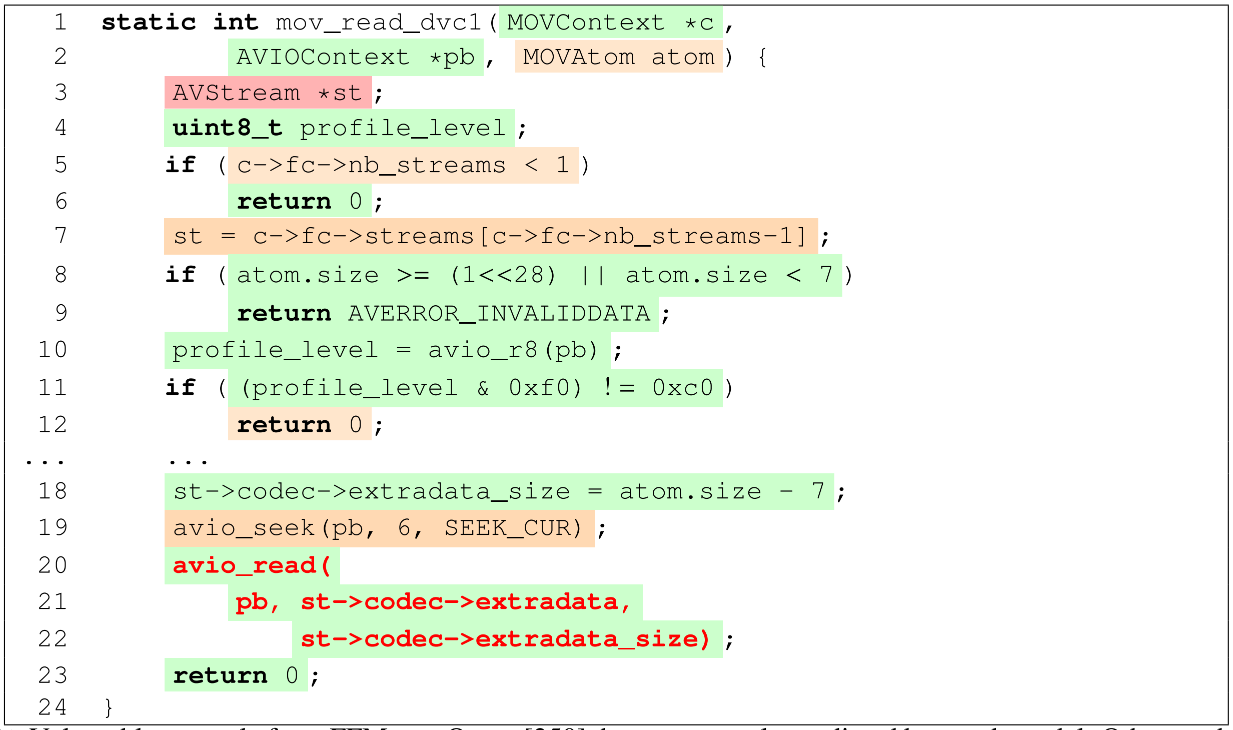
Table 4.2 tabulates our findings on techniques on Vulnerability detection task. The results show that, graph-based models (both Devign and ReVeal) perform better than token based models in both the dataset. In ReVeal-data, ReVeal achieves median F1 score of 41.25. In contract, the best token-based model's (SySeVR) F1 score is 30.25. For the FFMpeg+Qemu dataset, again, SySeVR is the best performing token based model with median F1 score of 56.03. In contrast, ReVeal' median F1 score in this dataset is 64.42. To understand the contribution of Graph-Based modeling even further, we perform and ablation study, where we removed the GGNN from ReVeal's pipeline. Table 4.3 shows the result of this ablation study. When GGNN is removed from ReVeal's pipeline median F1-score drops 5.03% in ReVeal-data and 1.14% in FFMpeg+Qemu dataset. We conjecture that representing source code as a graph and building a graph-based model makes ReVeal better suitable for understanding the syntax and semantics of source code.

Result 4.1: *DL model understands PL properties (syntax, and semantics) better when such properties are explicitly encoded in the model. Since syntax and semantic properties are exhibited through a graph, building a graph-based model performs best in source code understanding task such as Vulnerability Detection.*



```
link_layer_show(struct ib_port *p,
            struct port_attribute *unused, char * buf){
    switch (rdma_port_get_link_layer(
                p->ibdev, p->port_num)) {
        case IB_LINK_LAYER_INFINIBAND:
            return sprintf(buf, "%s\n", "InfiniBand");
        case IB_LINK_LAYER_ETHERNET:
            return sprintf(buf, "%s\n", "Ethernet");
        default:
            return sprintf(buf, "%s\n", "Unknown");
    }
}
```

(a) Vulnerable code example in Draper [204] dataset correctly predicted by Russel *et al.*'s token-based method.

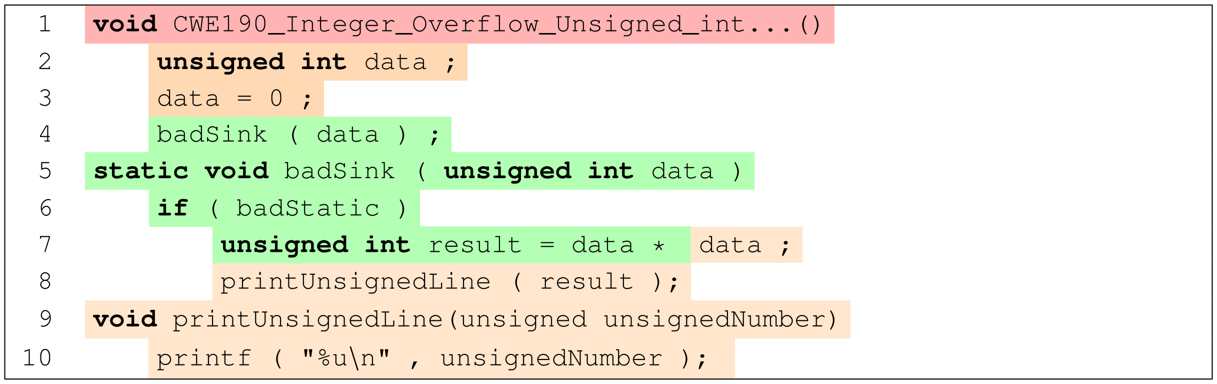


```
static int mov_read_dvc1(MOVContext *c,
        AVIOContext *pb, MOVAtom atom) {
    AVStream *st;
    uint8_t profile_level;
    if (c->fc->nb_streams < 1)
        return 0;
    st = c->fc->streams[c->fc->nb_streams-1];
    if (atom.size >= (1<<28) || atom.size < 7)
        return AVERROR_INVALIDDATA;
    profile_level = avio_r8(pb);
    if ((profile_level & 0xf0) != 0xc0)
        return 0;
    ...
    st->codec->extradata_size = atom.size - 7;
    avio_seek(pb, 6, SEEK_CUR);
    avio_read(
        pb, st->codec->extradata,
            st->codec->extradata_size);
    return 0;
}
```

(b) Vulnerable example from FFMpeg+Qemu [250] dataset correctly predicted by graph model. Other method could not predict the vulnerability in this example.

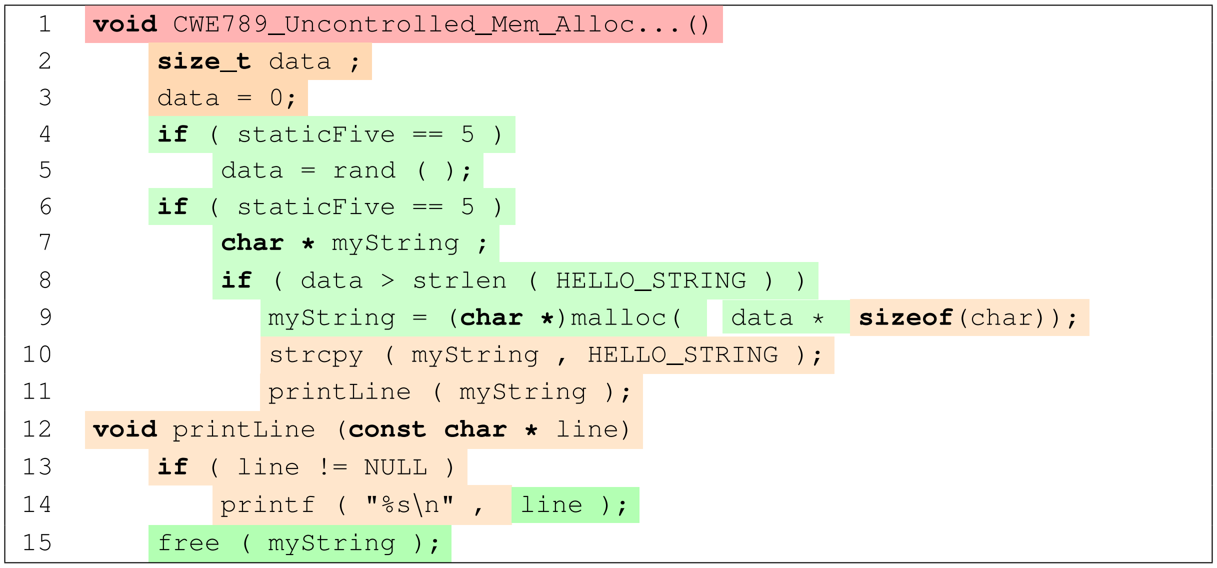
Figure 4.3: Contribution of different code component in correct classification of vulnerability by different model. Red-shaded code elements are most contributing, Green-shaded are the least. **Red** colored code are the source of vulnerabilities.

Further we qualitatively investigate, why does a graph-based model perform better than its token-based counterpart? Thus, we ask,



```
void CWE190_Integer_Overflow_Unsigned_int...()
    unsigned int data ;
    data = 0 ;
    badSink ( data ) ;
static void badSink ( unsigned int data )
    if ( badStatic )
        unsigned int result = data * data ;
        printUnsignedLine ( result );
void printUnsignedLine(unsigned unsignedNumber)
    printf ( "%u\n" , unsignedNumber );
```

Figure 4.4: Integer Overflow vulnerability (CWE-190).



```
void CWE789_Uncontrolled_Mem_Alloc...()
    size_t data ;
    data = 0;
    if ( staticFive == 5 )
        data = rand ( );
    if ( staticFive == 5 )
        char * myString ;
        if ( data > strlen ( HELLO_STRING ) )
           myString = (char *)malloc( data * sizeof(char));
           strcpy ( myString , HELLO_STRING );
           printLine ( myString );
void printLine (const char * line)
    if ( line != NULL )
        printf ( "%s\n" , line );
    free ( myString );
```

Figure 4.5: Uncontrolled Memory vulnerability (CWE-789).

RQ-4.2. Does a model learn irrelevant feature with improper choice of model?

In order to choose a good DL model for Vulnerability Detection, it is important to understand what features the model uses to make its predictions. A good model should assign greater importance to the vulnerability related code features.

Experimental Setup. To understand what features a model uses for its prediction, we find the feature importance assigned to the predicted code by the existing approaches. For token-based models such as VulDeePecker, SySeVR, and Russell *et al.*, we use Lemna to identify feature importance [79]. Lemna assigns each token in the input with a value ω_i^t, representing the contribution of that token for prediction. A higher value of ω_i^t indicates a larger contribution of token towards the prediction and vice versa. For graph-based models, such as Devign, Lemna is not applicable [79]. In this case, we use the activation value of each vertex in the graph to obtain the feature importance. The larger the activation, the more critical the vertex is.

Results. To visualize the feature importances, we use a heatmap to highlight the most to least important segments of the code. Figure 4.3 shows two examples of correct predictions. Figure 4.3(a) shows an instance where Russell *et al.*'s token-based method accurately predicted a vulnerability. But, the features that were considered most important for the prediction (lines 2 and 3) are not related to the actual vulnerability that appears in buggy `sprintf` lines (lines 6, 8, and 10). We observe similar behavior in other token based methods.

In contrast, Figure 4.3(b) shows an example that was misclassified as non-vulnerable by token-based methods, but graph-based models accurately predict them as vulnerable. Here we note that the vulnerability is on line 20, and graph-based models use lines 3, 7, 19 to make the prediction, *i.e.* mark the corresponding function as vulnerable. We observe that each of these lines shares a data dependency with line 20 (through `pb` and `st`). Since graph-based models learn the semantic dependencies between each of the vertices in the graph through the code property graph, a series of connected vertices, each with high feature importance, causes the graph-based model to make the accurate prediction. Token-based models lack the requisite semantic information and therefore fail to make accurate predictions. In addition to this case study, Figure 4.4, and Figure 4.5 shows two other examples of integer overflow and memory overflow, respectively. VulDeePecker [132] and SySeVR [131] correctly classified these as vulnerable. However, the heatmap shows the those models are just picking up mostly unrelated features from the code.

Result 4.2: *Form the case study, we conjecture that when appropriate model is not deployed for code understanding task, models tend to pick up irrelevant features from the dataset.*

4.5 Concluding Remarks

In this chapter, we systematically study different aspects of Deep Learning based Vulnerability Detection to effectively find real world vulnerabilities. We empirically show different shortcomings of existing models that potentially limits the usability of those techniques in practice. Our investigation found that existing modeling techniques do not completely address code semantics and data imbalance in vulnerability detection. Following these empirical findings, we propose a framework for collecting real world vulnerability dataset. We propose ReVeal as a configurable vulnerability prediction tool that addresses the concerns we discovered in existing systems and demonstrate its potential towards a better vulnerability prediction tool.

Part II
Implicit Encoding

Chapter 5: Unified Pre-training for Program Understanding and Generation

5.1 Motivation

In the previous two chapters, we have studied design of models for source code processing with explicit encoding of PL constructs. Such encoding require specifically crafted models to cater to specific need for PL properties. In this chapter (and next two chapters), we will study general purpose models with implicit encoding. In particular, we study the design and training of models from large corpus of source code and other metadata with minimal human supervision. Such training endows models with the knowledge about the properties of source code implicitly.

Engineers and developers write software programs in a programming language (PL) like Java, Python, etc., and often use natural language (NL) to communicate with each other. Use of Natural Languages such as English in software engineering ranges from writing documentation, commit messages, bug reports to seeking help in different forums (*e.g.,* Stack Overflow), etc. Automating different software engineering applications, such as source code summarization, generation, and translation, heavily rely on the understanding of PL and NL—we collectively refer them as PLUG (stands for, Program and Language Understanding and Generation) applications or tasks. Note that the use of NL in software development is quite different than colloquially written and spoken language. For example, NL in software development often contains domain-specific jargon, *e.g.,* when software engineers use *Code Smell*[1], it means a potential problem in code (something other than *Smell* in regular English language).

In this work, our goal is to develop a general-purpose model that can be used in various PLUG applications. Recent advancements in deep learning and the availability of large-scale PL and de-

[1]https://en.wikipedia.org/wiki/Code_smell

Program snippet in Python

```
def sort_list(uns):
        return sorted(uns, key=lambda x:x[0])
```

Program snippet in Java

```
static Tuple[] sortArray(Tuple[] uns){
        return Arrays.sort(
          uns, new Comparator<Tuple>() {
                public int compare(
                 Tuple o1, Tuple o2) {
                        return o1.get(0) == o2.get(0);
                 }
        });
}
```

Summary: sort a list of tuples by first element.

Figure 5.1: Example motivating the need to understand the association of program and natural languages for code summarization, generation, and translation.

velopers' NL data ushered in the automation of PLUG applications. One important aspect of PLUG applications is that they demand a profound understanding of program syntax and semantics and mutual dependencies between PL and NL. For example, Figure 5.1 shows two implementations of the same algorithm (sorting) in two PL and corresponding NL summary. An automatic translation tool must understand that function `sorted` in Python acts similar to `Arrays.sort` in Java and the `lambda` operation in Python is equivalent to instantiating a `Comparator` object in Java. Similarly, a tool that summarizes either of these code must understand that `x[0]` in Python or `Tuple.get(0)` in Java refers to the first element in the tuple list.

Most of the available data in PL and NL are unlabeled and cannot be trivially used to acquire PLUG task-specific supervision. However, PLUG tasks have a common prerequisite — understanding PL and NL syntax and semantics. Leveraging unlabelled data to pre-train a model to learn PL and NL representation can be transferred across PLUG tasks. This approach reduces the requirement of having large-scale annotations for task-specific fine-tuning. In recent years we have seen a colossal effort to pre-train models on a massive amount of unlabeled data (*e.g.,* text, images, videos) [55, 138, 48, 49, 129, 213] to transfer representation encoders across a wide variety of applications. There are a few research effort in learning general purpose PL-NL representation encoders, such as CodeBERT [63] and GraphCodeBERT [78] that are pre-trained on a *small-scale* bimodal data (code-text pairs). Such models have been found effective for PLUG tasks, including code search, code completion, etc.

Language generation tasks such as code summarization is modeled as sequence-to-sequence learning, where an encoder learns to encode the input code and a decoder generates the target summary. Despite the effectiveness of existing methods, they do not have a pre-trained decoder for language generation. Therefore, they still require a large amount of parallel data to train the decoder. To overcome this limitation, Lewis *et al.* [127] proposed denoising sequence-to-sequence pre-training where a Transformer [228] learns to reconstruct an original text that is corrupted using an arbitrary noise function. Very recently, Lachaux *et al.* [122] studied denoising pre-training using a large-scale source code collection aiming at unsupervised program translation and found the approach useful. This raises a natural question, *can we unify pre-training for programming and natural language?* Presumably, to facilitate such pre-training, we need unlabeled NL text that is relevant to software development. Note that unlike other bimodal scenarios (*e.g.,* vision and language), PL and associated NL text share the same alphabet or uses anchor tokens (*e.g.,* "sort", "list", "tuple" as shown in Figure 5.1) that can help to learn alignment between semantic spaces across languages.

We introduce PLBART (Program and Language BART), a bidirectional and autoregressive transformer pre-trained on unlabeled data across PL and NL to learn multilingual representations applicable to a broad spectrum of PLUG applications. We evaluate PLBART on code summarization, generation, translation, program repair, clone detection, and vulnerability detection tasks. Experiment results show that PLBART outperforms or rivals state-of-the-art methods, *e.g.,* CodeBERT and GraphCodeBERT, demonstrating its promise on program understanding and generation. We perform a thorough analysis to demonstrate that PLBART learns program syntax, logical data flow that is indispensable to program semantics, and excels even when limited annotations are available.

5.2 Methodology

PLBART uses denoising sequence-to-sequence pre-training to utilize unlabeled data in PL and NL. Such pre-training lets PLBART reason about language syntax and semantics. At the same time, PLBART learns to generate language coherently.

Table 5.1: Statistics of the data used to pre-train PLBART. "Nb of documents" refers to the number of functions in Java and Python collected from Github and the number of posts (questions and answers) in the natural language (English) from StackOverflow.

	Java	**Python**	**NL**
All Size	352 GB	224 GB	79 GB
All - Nb of tokens	36.4 B	28 B	6.7 B
All - Nb of documents	470 M	210 M	47 M

Table 5.2: Example encoder inputs and decoder outputs during denoising pre-training of PLBART. We use three noising strategies: token masking, token deletion, and token infilling (shown in the three examples, respectively).

PLBART Encoder Input	**PLBART Decoder Output**
`Is 0 the` **`[MASK]`** `Fibonacci` **`[MASK]`** `?` **`<En>`**	**`<En>`** `Is 0 the` **`first`** `Fibonacci` **`number`** `?`
`public static main ( String args [ ] ) { date = Date ( ) ; System . out . ( String . format ( " Current Date : % tc " , ) ) ; }` **`<java>`**	**`<java>`** `public static` **`void`** `main ( String args [ ] ) {` **`Date`** `date =` **`new`** `Date ( ) ; System . out .` **`printf`** `( String . format ( " Current Date : % tc " ,` **`date`** `) ) ; }`
`def addThreeNumbers ( x , y , z ) : NEW_LINE INDENT return` **`[MASK]`** **`<python>`**	**`<python>`** `def addThreeNumbers ( x , y , z ) : NEW_LINE INDENT return` **`x + y + z`**

5.2.1 Denoising Pre-training

Data & pre-processing We pre-train PLBART on a large-collection of Java and Python functions and natural language descriptions from GitHub and StackOverflow, respectively. We download all the GitHub repositories associated with Java and Python languages available on Google

BigQuery.[2] We extract the Java and Python functions following the pre-processing pipeline from Lachaux *et al.* [122]. We collect the StackOverflow posts (include both questions and answers, exclude code snippets) by downloading the data dump (date: 7th September 2020) from stackexchange.[3] Statistics of the pre-training dataset are presented in Table 5.1. We tokenize all the data with a sentencepiece model [120] learned on 1/5'th of the pre-training data. We train sentencepiece to learn 50,000 subword tokens.

One key challenge to aggregate data from different modalities is that some modalities may have more data, such as we have 14 times more data in PL than NL. Therefore, we mix and up/down sample the data following [48] to alleviate the bias towards PL. We sample instances for pre-training according to a multinomial distribution with probabilities $(q_1, q_2, \ldots, q_N)$:

$$q_i = \frac{1}{p_i} \cdot \frac{p_i^\alpha}{\sum_{j=1}^N p_j^\alpha}, p_i = \frac{n_i}{\sum_{j=1}^N n_j},$$

where N is the total number of languages and n_i is the total number of instances in language i. We set the smoothing parameter α to 0.3.

Table 5.3: Example inputs to the encoder and decoder for fine-tuning PLBART on sequence generation tasks: source code summarization (S), generation (G), and translation (T).

	PLBART Encoder Input	**PLBART Decoder Output**
S	`def maximum (a , b , c) : NEW_LINE INDENT return max ( [ a , b , c ] ) <python>`	`<En> Find the maximum of three numbers`
G	`Find the maximum of three numbers <En>`	`<java> public int maximum ( int a , int b , int c ) { return Math . max ( a , Math . max ( b , c ) ) }`
T	`public int maximum ( int a , int b , int c ) { return Math . max ( a , Math . max ( b , c ) ) } <java>`	`<python> def maximum (a , b , c) : NEW_LINE INDENT return max ( [ a , b , c ] )`

[2]https://console.cloud.google.com/ marketplace/details/github/github-repos
[3]https://archive.org/download/stackexchange

Architecture PLBART uses the same architecture as BART$_{base}$ [127], it uses the sequence-to-sequence Transformer architecture [228], with 6 layers of encoder and 6 layers of decoder with model dimension of 768 and 12 heads (~140M parameters). The only exception is, we include an additional layer-normalization layer on top of both the encoder and decoder following Liu *et al.* [137], which is found to stabilize training with FP16 precision.

Noise function, f In denoising autoencoding, a model learns to reconstruct an input text that is corrupted by a noise function. Reconstruction of the original input requires the model to learn language syntax and semantics. In this work, we use three noising strategies: token masking, token deletion, and token infilling [127]. According to the first two strategies, random tokens are sampled and replaced with a mask token or deleted from the input sequence. In token infilling, a number of text spans are sampled and replaced with a *single* mask token. The span lengths are drawn from a Poisson distribution ($\lambda = 3.5$). We mask 35% of the tokens in each instance.

Input/Output Format The input to the encoder is a noisy text sequence, while the input to the decoder is the original text with one position offset. A language id symbol (e.g., <java>, <python>) is appended and prepended to the encoder and decoder inputs, respectively. We provide a few examples in Table 5.2. The input instances are truncated if they exceed a maximum sequence length of 512.

Learning PLBART is pre-trained on N languages (in our case, N=3), where each language N_i has a collection of unlabeled instances $\mathcal{D}_i = \{x_1, \ldots, x_{n_i}\}$. Each instance is corrupted using the noise function f and we train PLBART to predict the original instance x from $f(x)$. Formally, PLBART is trained to maximize $\mathcal{L}_\theta$:

$$\mathcal{L}_\theta = \sum_{i=1}^{N} \sum_{j=1}^{m_i} \log P(x_j | f(x_j); \theta)$$

where m_i is the number of sampled instances in language i and the likelihood P is estimated following the standard sequence-to-sequence decoding.

Optimization We train PLBART on 8 Nvidia GeForce RTX 2080 Ti GPUs for 100K steps. The effective batch size is maintained at 2048 instances. We use Adam (ϵ = 1e-6, $\beta_2 = 0.98$) with a linear learning rate decay schedule for optimization. We started the training with dropout 0.1 and reduced it to 0.05 at 50K steps and 0 at 80K steps. This is done to help the model better fit the data [137]. The total training time was approximately 276 hours (11.5 days). All experiments are done using the Fairseq library [171].

5.2.2 Fine-tuning PLBART

We fine-tune PLBART for two broad categories of downstream applications.

Sequence Generation PLBART has an encoder-decoder architecture where the decoder is capable of generating target sequences autoregressively. Therefore, we can directly fine-tune PLBART on sequence generation tasks, such as code summarization, generation, and translation. Unlike denoising pre-training, the source sequence is given as input to the encoder during fine-tuning, and the decoder generates the target sequence. The source and target sequence can be a piece of code or text sequence. Table 5.3 shows a few examples of input and output to and for PLBART for different generation tasks. Note that PLBART prepends a language id to the decoded sequence; it enables fine-tuning PLBART in a multilingual setting (*e.g.,* code generation in multiple languages).

Sequence Classification We fine-tune PLBART on sequence classification tasks following Lewis *et al.* [127]. The input sequence is fed into both the encoder and decoder. For a pair of inputs, we concatenate them but insert a special token ("`</s>`") between them. A special token is added at the end of the input sequence. This last token's representation from the final decoder layer is fed into a linear classifier for prediction.

Optimization We fine-tune PLBART for a maximum of 100K steps on all the downstream tasks with 2500 warm-up steps. We set the maximum learning rate, effective batch size, and dropout rate to 3e-5, 32 and 0.1, respectively. The final models are selected based on the validation BLEU (in generation task) or accuracy (in classification tasks). Fine-tuning PLBART is carried out in one Nvidia GeForce RTX 2080 Ti GPU.

Table 5.4: Details of the hyper-parameters used during fine-tuning for sequence generation tasks. * indicates pre-trained from scratch using source code-text pairs.

Hyper-parameter	**RoBERTa***	**CodeGPT-2**	**CodeBERT**	**GraphCodeBERT**	**PLBART**
vocab size	50,265	50,001	50,265	-	50,004
n_positions	514	1024	514	256	1024
model size	768	768	768	768	768
# layers	12	12	12	12	6
# heads	12	12	12	12	12
d_{ff}	3072	3072	3072	-	3072
dropout	0.1	0.1	0.1	-	0.1
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	5e-5	5e-5	5e-5	1e-4	5e-5
batch size	32	32	32	64	32

5.3 Experiment Design

To understand PLBART's performance in a broader context, we evaluate PLBART on several tasks. In particular, we investigate the following research questions,

- **RQ-5.1.** Does PLBART's pre-training help the model understand source code better?
- **RQ-5.2.** Does PLBART's pre-training teach model to generate correct code?
- **RQ-5.3.** How effective is PLBARTin learning automated code editing?

Our evaluation focuses on assessing PLBART's ability to capture rich semantics in source code and associated natural language text.

Table 5.5: Statistics of the downstream benchmark datasets.

Task	**Dataset**	**Language**	**Train**	**Valid**	**Test**
Summarizaion	Husain *et al.* [93]	Ruby	24,927	1,400	1,261
		Javascript	58,025	3,885	3,291
		Go	167,288	7,325	8,122
		Python	251,820	13,914	14,918
		Java	164,923	5,183	10,955
		PHP	241,241	12,982	14,014
Generation	Iyer *et al.* [95]	NL to Java	100,000	2,000	2,000
Translation	Code-Code [142]	Java to C#	10,300	500	1,000
		C# to Java	10,300	500	1,000
	Code Editing [225]	Java$_{small}$	46,680	5,835	5,835
		Java$_{medium}$	52,364	6,545	6,545
Classification	Vulnerability Detection [250]	C/C++	21,854	2,732	2,732
	Clone Detection [230]	Java	100,000	10,000	415,416

5.3.1 Evaluation Tasks

We divide the evaluation tasks into four categories. The evaluation task datasets are summarized in Table 5.5. We use CodeXGLUE [142] provided public dataset and corresponding train-validation-test splits for all the tasks.

Code Summarization refers to the task of generating a natural language (English) summary from a piece of code. We fine-tune PLBART on summarizing source code written in six different programming languages, namely, Ruby, Javascript, Go, Python, Java, and PHP.

Code Generation is exactly the opposite of code summarization. It refers to the task of generating a code (in a target PL) from its NL description. We fine-tune PLBART on the Concode dataset [95], where the input is a text describing class member functions in Java and class environment, the output is the target function.

Code Translation requires a model to generate an equivalent code in the target PL from the input code written in the source PL. Note that the source and target PL can be the same. Hence, we consider two types of tasks in this category. The first task is a typical PL translation task, translating a code *i.e.* from Java code to C#, and vice versa. In this task, the semantic meaning of the translated code should exactly match the input code. Thus, this task evaluates PLBART's understanding of program semantics and syntax across PL. The second task we consider is automated code editing. In this task, the input is a buggy code, and the output is a modified version of the same code which fixes the bug. This task helps us understand PLBART's ability to understand code semantics and apply semantic changes in the code.

Code Classification aims at predicting the target label given a single or a pair of source code. We evaluate PLBART on two classification tasks. The first task is clone detection, where given a pair of code, the goal is to determine whether they are clone of each other (similar to paraphrasing in NLP).. The second task is detecting whether a piece of code is vulnerable. This task help us gauging PLBART's effectiveness in program understanding in an unseen PL since the code examples in this task are written in C/C++.

5.3.2 Evaluation Metrics

BLEU computes the n-gram overlap between a generated sequence and a collection of references. We use corpus level BLEU [174] score for all the generation tasks, except code summarization where we use smoothed BLEU-4 score [134] following Feng *et al.* [63].

CodeBLEU is a metric for measuring the quality of the synthesized code [197]. Unlike BLEU, CodeBLEU also considers grammatical and logical correctness based on the abstract syntax tree and the data-flow structure.

Exact Match (EM) evaluates if a generated sequence exactly matches the reference.

5.3.3 Baseline Methods

We compare PLBART with several state-of-the-art models and broadly divide them into two categories. First, the models that are trained on the evaluation tasks from scratch, and second, the models that are pre-trained on unlabeled corpora and then fine-tuned on the evaluation tasks. Table 5.4 shows the hyperparameter details of PLBARTalong with other baseline models.

Training from Scratch

Seq2Seq [143] is an LSTM based Seq2Seq model with attention mechanism. Vocabulary is constructed using byte-pair encoding.

Transformer [228] is the base architecture of PLBART and other pre-trained models. Transformer baseline has the same number of parameters as PLBART. Hence, a comparison with this baseline demonstrates the direct usefulness of pre-training PLBART.

Pre-trained Models

As described in section 6.3, PLBART consists of an encoder and autoregressive decoder. We compare PLBART on two categories of pre-trained models. First, the encoder-only models (*e.g.,* RoBERTa, CodeBERT, and GraphCodeBERT) that are combined with a randomly initialized decoder for task-specific fine-tuning. The second category of baselines include decoder-only models (CodeGPT) that can perform generation autoregressively.

RoBERTa, RoBERTa (code) are RoBERTa [138] model variants. While RoBERTa is pre-trained on natural language, RoBERTa (code) is pre-trained on source code from CodeSearchNet [93].

CodeBERT [63] combines masked language modeling (MLM) [55] with replaced token detection objective [46] to pre-train a Transformer encoder.

GraphCodeBERT [78] is a concurrent work with this research which improved CodeBERT by modeling the data flow edges between code tokens. We report GraphCodeBERT's performance directly from the paper since their implementation is not publicly available yet.

GPT-2, CodeGPT-2, and CodeGPT-adapted are GPT-style models. While GPT-2 [187] is pre-trained on NL corpora, CodeGPT-2 and CodeGPT-adapted are pre-trained on CodeSearchNet [142]. Note that, CodeGPT-adapted starts from the GPT-2 checkpoint for pre-training.

5.4 Research Findings

We designed PLBARTto jointly understand and generate source code and apply it to different downstream tasks. To understand PLBART's capability better in both code understanding, we first ask,

RQ-5.1. Does PLBART's pre-training help the model understand source code better?

Experimental Setup. To answer this question, we compare PLBARTwith other tools *w.r.t.* three different tasks – Code summarization, Vulnerability Detection, and Clone detection. All these tasks require models' profound understanding of source code. Among these tasks, Code summarization is a generative task, where the model needs to generate an NL summary of a given source code. On the other hand, Vulnerability Detection and Clone detection are Code Classification tasks. Given a code (or pair of codes), the model classifies the input into positive and negative classes.

Results. Table 5.6 shows the result of code summarization. PLBART outperforms the baseline methods in five out of the six programming languages with an overall average improvement of 0.49 BLEU-4 over CodeBERT.

The highest improvement (~16%) is in the Ruby language, which has the smallest amount of training examples. Unlike CodeBERT, PLBART is not pre-trained on the Ruby language; however, the significant performance improvement indicates that PLBART learns better generic program semantics. In contrast, PLBART performs poorly in the PHP language. The potential reason is syntax mismatch between the pre-trained languages and PHP. Surprisingly, RoBERTa performs

Table 5.6: Results on source code summarization, evaluated with smoothed BLEU-4 score. The baseline results are reported from Feng *et al.* [63].

Methods	Ruby	Javascript	Go	Python	Java	PHP	Overall
Seq2Seq	9.64	10.21	13.98	15.93	15.09	21.08	14.32
Transformer	11.18	11.59	16.38	15.81	16.26	22.12	15.56
RoBERTa	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT	12.16	14.90	18.07	19.06	17.65	**25.16**	17.83
PLBART	**14.11**	**15.56**	**18.91**	**19.30**	**18.45**	23.58	**18.32**

better than PLBART on the PHP language. We suspect that since RoBERTa is pre-trained on natural language only, it does not suffer from the syntax mismatch issue. Overall in comparison to the Transformer baseline, PLBART improves with an average of 2.76 BLEU-4, and we credit this improvement to the pre-training step.

In both clone detection and the vulnerability detection tasks, PLBART outperforms CodeBERT. We present the results in Table 5.7. In the vulnerability detection task, code semantics is the most critical feature [250, 38]. Since PLBART is not pre-trained on C/C++ language, its improved performance compared to the Transformer baseline is the testament that PLBART can identify semantics beyond the language syntax's specifics. Moreover, PLBART's improved performances over CodeBERT and GraphCodeBERT confirms its effectiveness in program understanding in addition to its generation ability.

Table 5.7: Results on the vulnerable code detection (accuracy) and clone detection (F1 score) tasks.

Tasks	Vulnerability Detection	Clone Detection
Transformer	61.64	-
CodeBERT	62.08	96.5
GraphCodeBERT	-	97.1
PLBART	**63.18**	**97.2**

We acknowledge that neither PLBART nor CodeBERT is state-of-the-art in vulnerability detection, as graph-based models perform best in this task. VulBerta [81], and CoTexT [181] were the SOTA for this task at the time of this project, and while writing this dissertation, respectively. Nevertheless, our goal in this evaluation is to study how well PLBART understands program semantics in an unseen language for a different type of task (other than the generation, *i.e.* classification).

Result 5.1: PLBART*demonstrates exemplary performance in tasks that require source code understanding. For source code summarization,* PLBART*outperforms existing pre-trained models in five of the six programming languages. For vulnerability detection and close detection,* PLBART*performs comparably or slightly better than the model explicitly pre-trained to understand code* – e.g., *CodeBERT.*

Next, we evaluate PLBART's performance in different tasks that requires model's capacity to generate source code.

RQ-5.2. Does PLBART's pre-training teach model to generate correct code?

Experimental Setup. In this research question, we evaluate PLBARTin two different source code generative tasks – NL to Code Generation and Code to Code Translation. The output of these tasks is source code; thus, these experiments will stress-test PLBART's ability in source code generation.

Results.

NL to Code Generation Table 5.8 shows the evaluation result on code generation from NL description. PLBART outperforms all the baselines in terms of BLEU and CodeBLEU. While CodeGPT-adapted [142] achieves the best Exact Match (EM) score, PLBART outperforms CodeGPT-adapted by a large margin in terms of CodeBLEU. This result implies that PLBART generates *significantly more* syntactically and logically correct code than all the baselines.

Table 5.9 shows an example of code generated by PLBART. The difference between the reference code and the generated code is in line 6 onward. In the reference code, `loc0` is returned, however same `loc0` is returned in an `else` block in the generated code. If we look closely, in

Table 5.8: Results on text-to-code generation task using the CONCODE dataset [95].

Methods	Exact Match Accuracy	BLEU	CodeBLEU
Seq2Seq	3.05	21.31	26.39
Guo *et al.* [77]	10.05	24.40	29.46
Iyer *et al.* [94]	12.20	26.60	-
GPT-2	17.35	25.37	29.69
CodeGPT-2	18.25	28.69	32.71
CodeGPT-adapted	**20.10**	32.79	35.98
PLBART	18.75	**36.69**	**38.52**
PLBART$_{10K}$	17.25	31.40	33.32
PLBART$_{20K}$	18.45	34.00	35.75
PLBART$_{30K}$	18.65	34.84	37.08
PLBART$_{50K}$	17.70	35.02	37.11
CodeBART$_{80K}$	18.45	36.76	39.01

Table 5.9: An example of generated code by PLBART that is syntactically and semantically valid, but does not match the reference.

Input text	`returns the count to which the specified key is mapped in this frequency counter , or 0 if the map contains no mapping for this key .`

Reference Code	`Integer function (T arg0) {`
	`    Integer loc0 = counter.get(arg0);`
	`    if (loc0 == null) {`
	`            return 0 ;`
	`    }`
	`    return loc0;`
	`}`

Generated Code	`int function (T arg0) {`
	`    Integer loc0 = counter.get(arg0);`
	`    if (loc0 == null) {`
	`            return 0 ;`
	`    }`
	`    else {`
	`            return loc0;`
	`    }`
	`}`

the reference code, line 6 will be executed only if the condition in line 3 (*i.e.*`loc0 == null`) is `false`. In the generated code, `loc0` will be returned only if the condition in line 3 is `false`, making the generated code semantically equivalent to the reference code.

To study whether PLBART learns code syntax and logical flow during pre-training or fine-tuning, we perform an ablation study where we use subset of the training examples (10K, 20K, and 50K) to finetune PLBART in this task.As table 5.8 shows, with only 10K examples, PLBART outperforms all baselines in terms of CodeBLUE. This ablation shows that PLBART learns program syntax and data flow during pre-training, resulting in effective performance on downstream tasks even when finetuned on small number of examples.

As we have shown in CODIT(§3), generating syntactically and logically correct code has been a big challenge in program generation. While CODIT's model is explicitly designed to ensure syntactic correctness, We conjecture that PLBART's large-scale denoising sequence-to-sequence pre-training implicitly enables the model understand program syntax and logical flow; therefore enables PLBART to generate syntactically and logically valid code.

Table 5.10: Results on source code translation using Java and C# language dataset introduced in [142]. PBSMT refers to phrase-based statistical machine translation where the default settings of Moses decoder [119] is used. The training data is tokenized using the RoBERTa [138] tokenizer.

Methods	**Java to C#**			**C# to Java**		
	BLEU	**EM**	**CodeBLEU**	**BLEU**	**EM**	**CodeBLEU**
Naive Copy	18.54	0	34.20	18.69	0	43.04
PBSMT	43.53	12.50	42.71	40.06	16.10	43.48
Transformer	55.84	33.00	63.74	50.47	37.90	61.59
RoBERTa (code)	77.46	56.10	83.07	71.99	57.90	80.18
CodeBERT	79.92	59.00	85.10	72.14	58.80	79.41
GraphCodeBERT	80.58	59.40	-	72.64	58.80	-
PLBART	**83.02**	**64.60**	**87.92**	**78.35**	**65.00**	**85.27**

Code to Code Translation. Table 5.10 presents the evaluation results on code translation. PLBART outperforms all the baselines *w.r.t.* EM, BLEU, and CodeBLEU. PLBART improves over CodeBERT by 9.5% and 10.5% when translating from Java to C# and C# to Java, respectively. Although PLBART is not pre-trained on C# language, there is a significant syntactic and semantic similarity between Java and C#. Thus PLBART understands C# language syntax and semantics. However, such similarities are non-trivial, making the Naive copy and PBSMT perform very poorly in both the translation tasks.

Table 5.11: Example C# code generated by PLBARTthat does not exactly match the reference code.

Reference Code : C#	
	```
	public bool find(int start_1){
	    findPos = start_1;
	    ...
	    else{
	            if (findPos >= _regionEnd){
	                    matchFound = false;
	                    return false;
	            }
	    }
	    ...
	}
	```

Generated Code : C#	
	```
	public bool find(int start){
	    findPos = start;
	    ...
	    else if (findPos >= _regionEnd){
	            matchFound = false;
	            return false;
	    }
	    ...
	}
	```

Table 5.11 shows an example where PLBART's generated C# code does not exactly match the reference; however, they are semantically equivalent. In the reference, the `else` block (line 4-9) is equivalent to the `else if` block (line 4-7) in the generated code. In addition, `start` is generated as function parameter and used in the function body, equivalent to `start_1` in the reference code. This further corroborates the syntactic understanding of PLBART and its ability to reason about the data flow in source code.

Result 5.2: PLBART*'s pre-training teaches the model to syntactically and semantically correct code. When finetuned on a fraction of training data,* PLBART*demonstrates significantly better performance in NL to code generation and code to code translation by generating syntactically and semantically correct code.*

Finally, we evaluate PLBART in the context of this dissertation. In particular, we evaluate PLBART's performance for automatic program editing.

RQ-5.3. How effective is PLBARTin learning automated code editing?

Table 5.12: Results on program repair (Abstract Code).

Methods	**Small**		**Medium**	
	Exact Match Accuracy	**BLEU**	**Exact Match Accuracy**	**BLEU**
Naive Copy	0	78.06	0	90.91
Seq2Seq	10.00	76.76	2.50	72.08
Transformer	14.70	77.21	3.70	89.25
CodeBERT	16.40	77.42	5.16	**91.07**
GraphCodeBERT	17.30	**80.58**	**9.10**	72.64
PLBART	**19.21**	77.02	8.98	88.50

Experimental Setup. We evaluate PLBART's effectiveness in automated code editing in two different versions (abstract and concrete) of the Bugfix dataset proposed by Tufano *et al.* [223]. While in the Concrete version, the identifier names appearing in the original code are retained, the abstract version replaces such concrete identifier names with abstract names such as `VAT_1`, `METHOD_1` , etc. In this task, both the input and the output are in the same language. While the input is a buggy code, the output should be the target bug-free code. Thus in this task, the exact match is the critical metric.

Results. Table 5.12 shows the result of automated code editing in the abstract dataset. PLBART can generate 17.13% and 74.03% more correct bug fixes than CodeBERT in Java$_{small}$ and Java$_{medium}$ datasets, respectively. Unlike CODIT, in this evaluation, we are generating the whole method after the edit pattern is applied since we pre-trained PLBARTon method granularity.

Table 5.13 shows the results on the concrete version of the code edits. Here we compare two different granularities of the edited code. First, we compare when the model only generates the

Table 5.13: Evaluation of PLBARTin generating concrete edits.

Method	Mode of Edited Code	Exact Match Accuracy	
		Small	Medium
CODIT	Patch Only	6.53	4.79
Transformer (PLBART without pretraining)	Patch Only	20.65	7.87
	Full Method	14.85	4.97
PLBART	Patch Only	**26.67**	**19.79**
	Full Method	20.35	8.35

patched code. This experiment allows us to compare our earlier method, CODIT directly. Also, we compare the whole method generation on the concrete dataset. When the whole edited method is being generated, PLBART outperforms the Transformer model [4] by 37% in the small dataset and 68% in the medium dataset. Such an improvement is wholly attributed to the PLBART's pre-training. PLBART achieves *significantly higher* performance than CODIT when we train PLBART to generate the patched code only. In contrast to CODIT's exact match accuracy of 6.53% and 4.79% exact in small and medium datasets, respectively, PLBARTachieves 26.67% and 19.79% in those datasets. While we designed CODIT to explicitly encode the syntax property of source code into the model, empirically, we show that PLBART can implicitly learn the syntax and semantic properties of source code through pre-training.

Result 5.3: PLBART*'s pre-training enables the model to jointly understand and generate source code, which is crucial for automated code editing. Results show that pre-training from a large corpus of data teach the model to generate syntactically and semantically correct code for automated code editing.*

5.5 Concluding Remarks

This chapter presents PLBART, a sizeable pre-trained sequence-to-sequence model that can perform program and language understanding and generation tasks. PLBART achieves state-of-

[4] This transformer model follows the same architecture as PLBART. The only difference is that this model is trained from scratch.

the-art performance on various downstream software engineering tasks, including code summarization, code generation, and code translation. Furthermore, experiments on discriminative tasks establish PLBART's effectiveness on program understanding. We also show that PLBART learns crucial program characteristics due to pre-training, such as syntax, identifier naming conventions, data flow. While in this chapter, we empirically establish that large-scale pre-training teaches the model through implicit encoding, in the next chapter, we will empirically investigate usage of such model in automated code editing. In particular, we will experimentally evaluate different input components useful for automated code editing in light of the implicitly encoded (pre-trained) model.

Chapter 6: On Multi-Modal Learning of Editing Source Code

6.1 Motivation

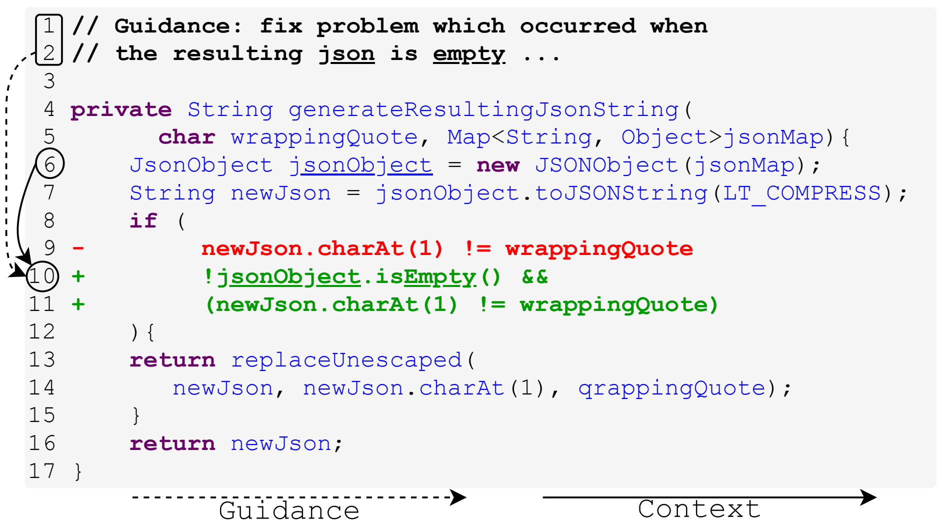
In the last chapter, we introduce PLBART, a bidirectional and autoregressive transformer pre-trained on unlabeled data across PL and NL to learn multilingual representations applicable to a broad spectrum of PLUG applications.

```
//Guidance:  use LinkedList and fix sublist problem ...
public void addPicture (String picture){
    if ((pictures) == null) {
-       pictures = new ArrayList<>();
+       pictures = new LinkedList<>(); //correct patch
+       pictures = new HashSet<>(); //plausible  patch
    }
    pictures.add(picture);
}
```

Figure 6.1: Example of an identical code (marked in red) changed in two different ways (**green** and blue) in two different contexts, where both can be correct patches. However, based on developers' guidance (top line) to fix a list related problem, **green** is the correct patch in this context.

While PLBART showed initial promise of using pre-trained model in automated code editing, learning such generic code changes is challenging. A programmer may change an identical piece of code in different ways in two different contexts, both can potentially be correct patches (see Figure 6.1).

For example, an identical code fragment `pictures = new ArrayList <> ()` was changed in two different ways:
`pictures = new HashSet<>();` and `pictures = new LinkedList<>()` in two different code contexts. Without knowing the developers' intention and the edit context, the automated code editing tools have no way to predict the most intended patches. For instance, in the above example, `LinkedList` was used to fix a sublist-related problem. Once such an intention



```
 1   // Guidance: fix problem which occurred when
 2   // the resulting json is empty ...
 3
 4   private String generateResultingJsonString(
 5          char wrappingQuote, Map<String, Object>jsonMap){
 6       JsonObject jsonObject = new JSONObject(jsonMap);
 7       String newJson = jsonObject.toJSONString(LT_COMPRESS);
 8       if (
 9 -          newJson.charAt(1) != wrappingQuote
10 +          !jsonObject.isEmpty() &&
11 +          (newJson.charAt(1) != wrappingQuote)
12       ){
13       return replaceUnescaped(
14          newJson, newJson.charAt(1), qrappingQuote);
15       }
16       return newJson;
17   }
```

Figure 6.2: A motivating example. The guidance provides a brief summary of what needs to be changes. The underlined tokens are directly copied from guidance and context into the patched code.

is known, it is easy to choose a LinkedList-related patch from the alternate options. Thus, such an additional modality of information can reinforce the performance of automated code-editing tools.

In fact, given just a piece of code without any additional information, it is perhaps unlikely that even a human developer can comprehend how to change it. Consider another real-life example shown in Figure 6.2. If a programmer *only* considers the edited expression in line 9, it is difficult to decide how to modify it. However, with additional information modalities – *i.e.* the guidance (line 1,2) and the context (the whole method before the patch), the correct patch often becomes evident to the programmer since the guidance effectively summarises how to change the code and the context provides necessary ingredients for generating a concretely patched code. We hypothesize that such multi-modal information could be beneficial to an automated code-editing tool. To that end, we design MODIT, a multi-modal code editing engine that is based on three information modalities: (i) the code fragment that needs to be edited, (ii) developers' intention written in natural language, and (iii) explicitly given edit context.

In particular, MODIT is based on a transformer-based [228] NMT model. As input, MODIT takes the code that needs to be edited (*e.g.,* the lines that need to be patched), additional guidance describing developers' intent, and the context of the edits that are explicitly identified by the developer (*e.g.,* the surrounding method body, or surrounding lines of code, etc.). Note that previous works [37, 44] also provided context and the edit location while generating edits; however, they are fed together to the model as a unified code element. Thus, the model had the burden of identifying the edit location and then generating the patch. In contrast, isolating the context from the edit location and feeding them to the model as different modalities provides MODIT with additional information about the edits.

Curating developers' intent for a large number of edits that can train the model is non-trivial. As a proof of concept, we leverage the commit messages associated with the edits to simulate developers' intent automatically. We acknowledge that commit messages could be noisy and may not always reflect the change summary [139]. Nonetheless, our extensive empirical result shows that, even with such noisy guidance, MODIT performs better in generating correctly edited code.

Being a model that encodes and generates source code, MODIT needs to both clearly understand and correctly generate programming languages (PL). While several previous approaches [37] designed sophisticated tree/grammar-based models to embed the knowledge of PL into the model, the most recent transformer-based approaches [63, 78, 5] showed considerable promise with pre-training with a large volume of source code. Since these models are pre-trained with billions of source code written by actual developers, and transformers are known to learn distant dependencies between the nodes, these models can learn about code structures during the pre-training step. Among such pre-trained models, PLBART [5] learns jointly to understand and generate source code and showed much promise in generative tasks. Thus, we chose PLBART as the starting point to train MODIT, i.e., we initialize MODIT's model with learned parameters from PLBART.

We evaluate MODIT on two different datasets ($B2F_s$, and $B2F_m$) proposed by Tufano *et al.* [225] consisting of an extensive collection of bug-fix commits from GitHub. Our empirical investigation shows that a summary of the change written in natural language as additional guidance from the developer improves MODIT's performance by narrowing down the search space for change patterns. The code-edit context, presented as a separate information modality, helps MODIT to generate edited code correctly by providing necessary code ingredients (*e.g.,* variable names, method names, *etc*). MODIT generates ~3.5 times more correct patches than CODIT showing that MODIT is robust enough to learn PL syntax implicitly. Furthermore, MODIT generates two times as many correct patches as a large transformer model could generate.

Additionally, our empirical investigation reveals that when we use one encoder to encode all information modalities rather than learning from individual modalities separately, the model learns representation based on inter-modality reasoning. In contrast, a dedicated encoder for each individual modality only learns intra-modality reasoning. Our experiment shows that a multi-modal/single-encoder model outperforms multi-modal/multi-encoder model by up to 46.5%.

We summarize our main contributions in this paper as follows.

- We propose MODIT– a novel multi-modal NMT-based tool for automatic code editing. Our extensive empirical evaluation shows that Automatic Code Editing can be vastly improved with additional information modalities like code context and developer guidance.

- We empirically investigate different design choices for MODIT. We provide a summary of the lessons that we learned in our experiments. We believe such lessons are valuable for guiding future research.

6.2 Methodology

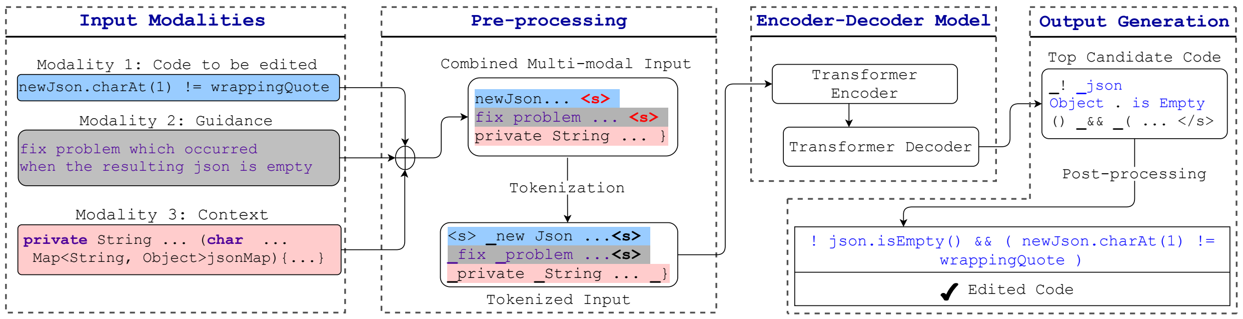


Figure 6.3: Overview of MODIT pipeline

Figure 6.3 shows an overview of MODIT's working procedure. MODIT is a multi-layer encoder-decoder based model consisting of a Transformer-based encoder and a Transformer-based decoder.

Both the encoder and decoder consist of 6 layers. MODIT works on three different modalities of information: (i) Code that needs to be edited (e_p), (ii) natural language guidance from the developer ($\mathcal{G}$), and (iii) the context code where the patch is applied (C). We acknowledge that e_p is essentially a substring of C. However, by explicitly extracting and presenting e_p to MODIT, we provide MODIT with additional information about the change location. Thus, despite being a part of the context, we consider e_p a separate modality. Nevertheless, MODIT consists of three steps. First, the pre-processing step processes and tokenizes these input modalities (§6.2.1). Then the encoder in MODIT encodes the processed input, and the decoder sequentially generates the patched code as a sequence of tokens (§6.2.2). At final step, MODIT post-processes the decoder generated output and prepares the edited code (§6.2.3).

6.2.1 Pre-processing

Input Consolidation. In the pre-processing step, MODIT generates consolidated multi-modal input (X) from the three input modalities (*i.e.* e_p, $\mathcal{G}$, and C). MODIT combines these input modalities as a sequence separated by a special `<s>` token *i.e.* $\boxed{X = e_p \texttt{ <s> } \mathcal{G} \texttt{ <s> } C}$. In the example shown in Figure 6.2, e_p is `newJson.charAt(1))!= wrappingQuote`, $\mathcal{G}$ is `fix problem which occurred when the resulting json is empty`, and C is the whole function before the edit (see *Input Modalities* in figure 6.3). MODIT generates a consolidates multi-modal input sequence as `newJson.charAt(1)) ... <s> fix problem which occurred ... <s> private String ... }`.

Tokenization. MODIT uses sentence-piece tokenizer [120]. Sentence-piece tokenizer divides every token into sequence of subtokens. Such subword tokenization is similar to previously used byte-pair encoding in automatic code editing literature [109, 101]. We use PLBART [5]'s sentence-piece tokenizer which is trained on billions of code from GitHub. After tokenizing the consolidated input X from figure 6.2, we get `_new Json . char At ( 1 ) ... <s> _fix _problem _which _oc cur red ... <s> _private _String ... _}`.

6.2.2 Encoder-Decoder Model

The input to MODIT's encoder-decoder model is a sequence of subtokens generated in the previous step.

Transformer Encoder. Given an input sequence $X = x_1, x_1, ..., x_n$, the encoder learns the representation of every token at layer l as $R_l^e(x_i)$ using self-attention computed as

$$R_l^e(x_i) = \sum_{j=i}^{n} a_{i,j} * R_{l-1}^e(x_j) \tag{6.1}$$

Where $R_{l-1}^e(x_j)$ is the representation of subtoken x_j as generated by layer $l-1$, and $a_{i,j}$ is the attention weight of subtoken x_i to x_j. Such attention weights are learned by multi-head attention [228]. Final layer generated representation (*i.e.* $R_6^e(x_i)$) is the final representation for every subtoken x_i in the input. Note that, the encoder learns the representation of Equation (6.1) of a subtoken, using all subtokens in the sequence. Thus the learned representation of every subtoken contains information about the whole input sequence. Since we encode all the information modalities in one sequence, the learned representation of every subtoken encodes information about other modalities.

Transformer Decoder. The decoder in MODIT is a transformer-based sequential left-to-right decoder consisting of 6 layers. It sequentially generates one subtoken at a time using previously generated subtokens and the final representation ($R_l^e(x_i)$) from the encoder. The decoder contains two modules – (i) self-attention, and (ii) cross-attention. The self-attention layer work similar to the self-attention in the encoder. First, with self attention, decoder generates representation $R^d l(y_i)$ of last generated token y_i with self attention on all previously generated tokens $(y_1, y_2, ..., y_i)$. This self attention follows same mechanism described in Equation (6.1). After learning decoder representation by self attention, decoder applies attention of encoder generated input representation using the following equation,

$$\mathcal{D}_l(y_i) = \sum_{j=i}^{n} \alpha_{i,j}^l * R_6^e(x_j) \tag{6.2}$$

Where $\alpha_{i,j}^l = softmax\left(dot\left(R_6^e\left(x_j\right), R^d l\left(y_i\right)\right)\right)$ is the attention weight between output subtoken

y_i to input subtoken x_j. The softmax generates an attention probability distribution over the length of input tokens. Finally the decoder learned representation, $D_l(y_i)$ is projected to the vocabulary to predict maximally likely subtoken from the vocabulary as next token.

In summary, the encoder learns representation of every subtokens in the input using all input subtoken, essentially encoding the whole input information in every input subtoken representation. The decoder's self-attention mechanism allows the decoder to attend to all previously generated subtokens allowing the decoder decide on generating correct token at correct place. The cross-attention allows the decoder to attend to encoded representation - implicitly letting the model decide where to copy from the input where to choose from new tokens in the vocabulary. We initialize the end-to-end encoder-decoder in MODIT using pre-trained weights of PLBART [5].

6.2.3 Output Generation

The decoder in MODIT continue predicting subtoken until it predicts the end of sequence `</s>` token. During inference, MODIT uses beam search to generate sequence of subtokens. Once the decoder finishes, MODIT post-processes the top ranked sequence in the beam search. First, MODIT removes the end of sequence `</s>` token. It then detokenizes the subtokens sequence to code token sequence. In this step, MODIT merges generated subtokens that are fragments of a code token into one code token. For the example shown in figure 6.2, MODIT generates the subtoken sequence `_! _json . is Empty () _&& _( _new Json . char At ( 1 ) _!= _wrap ping Quote _) </s>`. After detokenization, MODIT generates `! json.isEmpty()&& ( newJson.charAt(1)!= wrappingQuote ) .`

6.3 Experimental Design

6.3.1 Dataset

To prove our concept of MODIT, we experiment on two different datasets (*i.e.* $B2F_s$, and $B2F_m$) proposed by Tufano *et al.* [225]. In these two datasets, they collected large collections

Table 6.1: Statistics of the datasets studied.

Dataset	Avg. Tokens	Avg. Change Size*	Avg. tokens in Guidance	# examples		
				Train	Valid	Test
$B2F_s$	32.27	7.39	11.55	46628	5828	5831
$B2F_m$	74.65	8.83	11.48	53324	6542	6538

* Change size measured as token edit distance.

of bug-fix code changes along with commit messages from Java projects in GitHub. Each example in these datasets contains the java method before the change (C_p), the method after the change (C_n), and the commit message for the change. There are some examples ($<$ 100) with corrupted bytes in the commit message, which we could not process. We excluded such examples from the dataset. Table 6.1 shows statistics of the two datasets we used in this paper. $B2F_s$ contains smaller methods with maximum token length 50, and $B2F_m$ contains bigger methods with up to 100 tokens in length. The average size of the change (edit distance) is 7.39, and 8.83 respectively, in $B2F_s$ and $B2F_m$.

6.3.2 Data Preparation

For the datasets described in section 6.3.1, we extract the input modalities and the expected output to train MODIT. For every method pair (*i.e.* before edit - C_p, after edit - C_n) in those dataset, we use GumTree [62] to extract a sequence of tree edit locations. We identify the root of the smallest subtree of C_p's AST that encompasses all the edit operations. We call the code fragment corresponding to that subtree as *code to be edited*(e_p) and used as MODIT's first modality. Similarly, we extract the code corresponding to the smallest subtree encompassing all the edit operations from C_n and use that as *code after edit*(e_n). We use the commit message associated with the function pair as MODIT's second modality, guidance($\mathcal{G}$). Finally, we use the full method before edit (C_p) as MODIT's third modality, context(C).

6.3.3 Training

After combining every example in the datasets in MODIT's input (e_p, $\mathcal{G}$, C) and expected output (e_n), we use this combined dataset to train MODIT. For training MODIT, we use Label Smoothed Cross Entropy as loss function. We use Adam optimizer, with a learning rate of $5e^{-5}$. We train MODIT for 30 epochs, after every epoch, we run beam search inference on the validation dataset. We stop training if the validation performance does not improve for five consecutive validations.

6.3.4 Evaluation Metric

We use the top-1 accuracy as the evaluation metric throughout the paper. For proof-of-concept, we evaluate all techniques with beam size 5. When the generated patched code matches *exactly* with the expected patched code e_n, it is correct, incorrect otherwise. Note that this is the most stringent metric for evaluation. Previous approaches [43, 146] talked about filtering out infeasible patches from a ranked list of top k patches using test cases. However, we conjecture that such test cases may not always be available for general purpose code edits. Thus, we only compare top-1 accuracy.

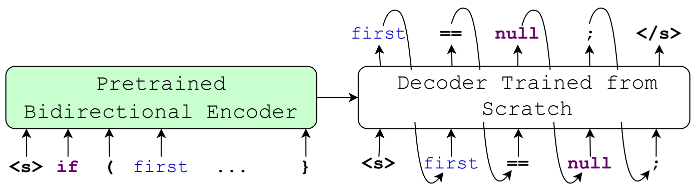
6.4 Research Findings

MODIT contains several design components: (i) use of multimodal information, (ii) use of transformer and initializing it with the pre-trained model, and (iii) use of end-to-end encoder-decoder (using PLBART) to generate patches instead of separately using pre-trained encoder or pre-trained decoder, as used by previous tools. First, we are interested in evaluating MODIT *w.r.t.* state-of-the-art methods. In particular, we evaluate how these three design choices effect MODIT's performance. So, we start with investigating,

RQ-6.1. How accurately does MODIT generate edited code *w.r.t.* other techniques?

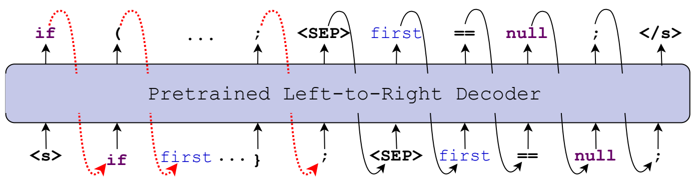
Experimental Setup. We carefully chose the baselines to understand the contribution from different design choices of MODIT. We evaluated our model in two experimental settings. First, we train different baseline models where the full model is trained from scratch. In this setting, the first baseline we consider is an LSTM with attention [23] NMT model. Various existing code patching approaches [225, 43] used such settings. Second baseline is Transformer [228] based Seq2Seq model. We consider two different-sized transformers. This enables us to contrast effect of model size in code-editing performance. The *Transformer-base* model consists of six encoder layers and six decoder layers. The *Transformer-base* model's architecture is the same as MODIT's architecture. Furthermore, we consider another transformer with a much larger architecture. *Transformer-large* contains twelve encoder layers and twelve decoder layers with three times as many learnable parameters as the *Transformer-base* model. The final baseline in this group is CODIT, which is a tree-based model. Comparison *w.r.t.* CODIT allows us to contrast externally given syntax information (in the form of CFG) and learned syntax by transformers (*i.e.* MODIT). We use all three input modalities (see Figure 6.3 for example) as input to the LSTM and Transformer. Using auxiliary modalities is non-trivial with CODIT since the input to CODIT must be a syntax-tree. Thus, we use uni-modal input (e_p) with CODIT.

In the second setting, we consider different pre-trained models, which we used to fine-tune for patch generation. Figure 6.4 shows schematic diagrams of the pre-trained models we compared in this evaluation. First two models we considered are CodeBERT [63], and GraphCodeBERT [78]. Both of these models are pre-trained encoders primarily trained to understand code. To use these for the patching task, we add a six-layered transformer-based decoder along with the encoder. The decoder is trained from scratch (see figure 6.4(a)). Another pre-trained baseline is CodeGPT [142]. GPT is a single left-to-right decoder model primarily pre-trained to generate code. For the code editing task, a special token `<SEP>` combines the input and the output as a sequence separated. Jiang *et al.* [101] showed the effectiveness of GPT for the source code patching task (see figure 6.4(b)). In contrast to these pre-trained models, MODIT uses PLBART, an end-to-end encoder-decoder model trained to understand and generate code simultaneously (see figure 6.4(c)). To



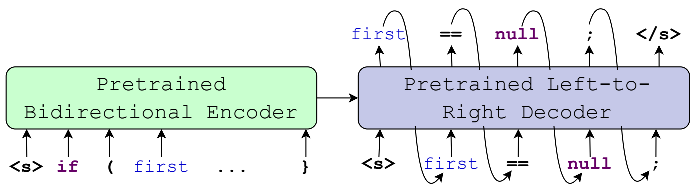
Pretrained Bidirectional Encoder
Decoder Trained from Scratch
<s> if (first ... }
<s> first == null ;
first == null ; </s>

(a) CodeBERT — Consist of bidirectional pretrained encoder and a decoder trained from scratch.



Pretrained Left-to-Right Decoder
if (... ; <SEP> first == null ; </s>
<s> if first ... } ; <SEP> first == null ;

(b) CodeGPT — One pretrained single decoder processes the input and output sequentially from left to right.



Pretrained Bidirectional Encoder
Pretrained Left-to-Right Decoder
<s> if (first ... }
<s> first == null ;
first == null ; </s>

(c) PLBART — Consist of pretrained bidirectional encoder and pretrained left to right decoder.

Figure 6.4: Schematic diagram of the three types of pre-trained models. used to evaluate MODIT.

compare from a fairground, we evaluate these pre-trained models with uni-modal input (e_p), and multi-modal input (e_p<s> $\mathcal{G}$<s> C), separately.

Results. Table 6.2 shows the accuracy in top 1 predicted patch by MODIT along with different baselines. LSTM based Seq2Seq model predicted 6.14% and 1.04% correct patches in $B2F_s$ and $B2F_m$ respectively. The *Transformer-base* model achieves 11.18% and 6.61% top-1 accuracy in those datasets, which improves further to 13.40% and 8.63% with the *Transformer-large* model. CODIT predicts 6.53% and 4.79% correct patches in $B2F_s$ and $B2F_m$, respectively. Note that CODIT takes the external information in the form of CFG; thus, the patches CODIT generate are syntactically correct. Nevertheless, the transformers, even the smaller model, perform better to predict the correct patch. We conjecture that the transformer model can implicitly learn the code

Table 6.2: Top-1 accuracies of different models *w.r.t.* their training type, model sizes, input modality.

Training Type	Model Name	# of params (M)	Multi-Modal	Accuracy (%)	
				$B2F_s$	$B2F_m$
From Scratch	LSTM	82.89	✓	6.14	1.04
	Transformer-base	139.22	✓	11.18	6.61
	Transformer-large	406.03	✓	13.40	8.63
	CODIT	105.43	✗	6.53	4.79
Fine-tuned	CodeBERT	172.50	✗	24.28	16.76
			✓	26.05	17.13
	GraphCodeBERT	172.50	✗	24.44	16.85
			✓	25.67	18.31
	CodeGPT	124.44	✗	28.13	16.35
			✓	28.43	17.64
	Vanilla PLBART†	139.22	N/A	20.35	8.35
	MODIT	139.22	✗	26.67	19.79
			✓	**29.99**	**23.02**

† - vanilla PLBART generates the whole method after the edit, while MODIT generates just the patched code.

syntax without direct supervision.

In contrast to the models trained from scratch, when we fine-tune a pre-trained model, it generates *significantly more* correct patches than models trained from scratch. For instance, MODIT (initialized with pre-trained PLBART) generates *168%* and *248%* more correct patches than the *Transformer-base* model (with randomly initialized parameters), despite both of these models having the same architecture and the same number of parameters. In fact, the smallest fine-tuned model (CodeGPT) performs much better than the larger model trained from scratch (*Transformer-large*).

All the fine-tuned models exhibit better performance when the input data are multi-modal with various degrees of improvement. With all three input modalities, CodeBERT [63] generates 7% and 2.2% more correct patches in $B2F_s$ and $B2F_m$, respectively, compared to a unimodal CodeBERT model. In case of MODIT, such improvement is 11.07% in $B2F_s$ and 16.23% in $B2F_m$. The $\mathcal{G}$ in the multi-modal data often contains explicit hints about how to change the code. For instance, consider the example shown in Figure 6.2, the guidance explicitly says there is a problem with the `json` when it is `empty`. Furthermore, with the presence of C in the input, the model can identify different variables, methods used in the method and potentially copy something from the context. We conjecture that such additional information from these two additional input modalities (i) reduce the search space for change patterns, (ii) help models copy relevant identifiers from the context.

Among the fine-tuned models multi-modalities, MODIT generates 15.12% more correct patches than CodeBERT, 16.82% than GraphCodeBERT, and 5.49% than CodeGPT in $B2F_s$. In the case of $B2F_m$ dataset, MODIT's improvement in performance is 34.38%, 25.72%, 30.50% higher than CodeBERT, GraphCodeBERT, and CodeGPT, respectively. To understand these results better, let us look at some of the examples.

```
//Guidance:  merging of items that aren't actually equal
public static boolean equals(
                ItemStack one, ItemStack two) {
-   return one.isSimilar(two) &&
             (one.toString().equals(two.toString()));
+   return one.isSimilar(two); //MODIT generated
    /* CodeGPT generated */
+   return one.toString().equals(two.toString());
}
```

Figure 6.5: Example patch where MODIT was able to generate correct patch, but CodeGPT could not. MODIT's patch is shown in **green**, and CodeGPT generated patch is shown in blue.

Figure 6.5 shows an example patch where MODIT correctly generated the expected patch but CodeGPT could not. If we look closely, we can see that the code to be changed (e_p) is a boolean expression where the two clauses are combines with `&&`. While only the first clause, `one.isSimilar(two)` is the expected output, CodeGPT chooses the second clause, `one.toString().equals(two.toString())` from the original. Recall from figure 6.4(b), CodeGPT processes the combined input and output sequence (separated by special `<SEP>` token) in *left-to-right* fashion. Thus, encodes representation of the input tokens do not contain information about the whole input sequence. In contrast, the MODIT uses a pre-trained *bi-direction* encoder which helps MODIT to understand the input fully. Based on the examples we have seen and the empirical result, we conjecture that, for code-editing tasks, the model must fully understand the

```
// Guidance:  ...  code refactoring ...
public boolean isEmpty() {
-     if((first) == null){ return true;}
-     return false;
+     return (first) == null; //MODIT predicts
    /* CodeBERT generated */
+     return ((first) == null) || (first.get()) == null;
}
```

Figure 6.6: Correctly predicted patch by MODIT. CodeBERT could not understand and reason about the textual hint to predict the correct patch.

input in a bi-directional fashion.

Figure 6.6 shows an example where MODIT generated correct patch, CodeBERT could not. Note that the guidance text explicitly asks about *code refactoring*, implying that the patched code should be semantically similar to the original code. Similar to the original code, patched could should return `true` when `first == null`, otherwise it should return `false`. An automated code change tool should not add additional code features when doing the refactoring. However, CodeBERT generated patch which introduced an additional clause `first.get()== null` in the return expression, which make CodeBERT's generate code semantically different from the original. MODIT was able to generate the correct patch for this example.

Finally, we summarize the empirical lessons we learned in this research question as

- Multi-modal input improves Code-Editing capability, irrespective of the underlying model used. The guidance often narrows the edit pattern search space, and the context narrows down the token generation search space.

- Transformer models (especially larger ones) are robust enough to learn the code's syntax information without direct supervision. When a pre-trained model is used to initialize transformer parameters, the improvement is *notably higher*.

- For code-editing task, both *understanding the input* and *correctly generated* output are important. While a pre-trained encoder understands the code and a pre-trained decoder generates correct code, an end-to-end pre-trained encoder-decoder model (*e.g.*, PLBART) the best choice to fine-tune for this task.

Result 6.1: MODIT *generates 29.99%, and 23.02% correct patches in top-1 position for two different datasets outperforming CodeBERT by up to 25.72%, GraphCodeBERT by up to 34.38%, and CodeGPT by up to 30.50%. Pre-trained models tend to be more effective than models trained from scratch for code editing—*MODIT *improves the performance by 167% than the best model trained from the scratch.*

MODIT uses three input modalities. Our next evaluation target is how these individual modalities effect MODIT's performance? Thus we ask,

RQ-6.2. What are the contribution of different input modalities in MODIT 's performance?

Experimental Setup. In this experiment, we investigate the contribution of different input modalities in MODIT's performance. Recall from section 6.2.1 that we use three inputs in MODIT (*i.e.* e_p, C, $\mathcal{G}$). Here, we investigate different combinations of such input modalities. More precisely, we investigate the influence of three information sources: (i) code that needs to be changed (e_p), (ii) context (C), and (iii) guidance ($\mathcal{G}$). Note that, by presenting e_p as a separate information modality, we are essentially providing MODIT with the information about the location of the change. To study the effect of such presentation, we study another alternative experimental setup, where we annotate the change location inside the context with two unique tokens `<START>` and `<END>`.

Results. Table 6.3 shows MODIT's performance with different combination of input modalities. When we present only the context to MODIT, it predicts 13.05% correct patches in $B2F_s$ and 4.50% in the $B2F_m$, which improves further to 17.89%, and 4.51% in those two datasets respectively when we add $\mathcal{G}$. Note that in these two scenarios, the model does not explicitly know which portion of the code needs to be edited; it sees the whole method and predicts (only) the patched code (e_n). In addition to learning how to patch, the model implicitly learns where to apply the patch in this setup. To test whether the identification of such location is the performance bottleneck, we

Table 6.3: Contribution of different input modalities in MODIT's performance. ✓ indicates that corresponding input modality is used as encoder input, ✗ indicates otherwise. We report top-1 accuracy as performance measure. Exp. ID is used later to refer to corresponding experiment result. Exp. ID Φ_* denotes an experiment with $*$ as input modalities.

Exp. ID	**Inputs**			**Accuracy (%)**	
	e_p	C	$\mathcal{G}$	$B2F_s$	$B2F_m$
Φ_c	✗	✓	✗	13.05	4.50
Φ_{cg}	✗	✓	✓	**17.89**	**4.51**
$\Phi_c^\dagger$	✗	✓†	✗	13.03	4.53
$\Phi_{cg}^\dagger$	✗	✓†	✓	**17.90**	**4.60**
Φ_e	✓	✗	✗	26.67	19.79
Φ_{eg}	✓	✗	✓	**28.76**	**21.63**
Φ_{ec}	✓	✓	✗	29.79	21.40
Φ_{ecg}	✓	✓	✓	**29.99**	**23.02**

† e_p is surrounded by two special tokens **`<START>`** and **`<END>`** inside the context.

surround the code that needs to be patched with two special tokens `<START>` and `<END>`. SequenceR [44] also proposed such annotation of buggy code. Surprisingly, such annotation resulted in comparable (slightly worse in one case) performance by MODIT.

In the next set of experiments, we extract the code that needs to be edited (e_p) and present it as a separate input modality. First, we only present the e_p without the other two modalities. When we only present the e_p and generate the edited code (e_n), it results in 26.67% top-1 accuracy in the $B2F_s$ and 19.79% in the $B2F_m$. Ding *et al.* [58] attributed such improvement to the reduced search space due to shorter input. Our result corroborates their empirical findings. Nevertheless, when we add the $\mathcal{G}$ modality with the e_p, MODIT's performance improves to 28.76% and 21.63% in $B2F_s$ and $B2F_m$, respectively.

In our final set of experiments in this research question, we augment e_p with the C. In this evaluation setup, MODIT predicts 29.79% correct patches in the $B2F_s$ and 21.40% in the $B2F_m$, which is improved further to 29.99%, and 23.02% correct patches in those two datasets when we add $\mathcal{G}$.

Figure 6.7 shows an example where MODIT with all modalities could successfully generate

```
// Guidance:  fixed some bugs in type checking
// improved performance by caching types of expressions
private TypeCheckInfo getType(SadlUnionType expression){
    ...
    return new TypeCheckInfo(
-         declarationConceptName, declarationConceptName
          /* MODIT generated patch with guidance */
+         declarationConceptName, declarationConceptName,
+         this, expression
          /* MODIT generated patch without guidance */
+         this.declarationConceptName,
+         this.declarationConceptName
    );
}
```

Figure 6.7: Example showing the effect of textual guidance in MODIT's performance. MODIT produced the **correct patch** with guidance, without guidance as input MODIT's produced **patch** is essentially refactored version of original input.

correct patch. The text guidance ($\mathcal{G}$) provides hint that variable `expression` should somehow associate with the construction of `TypeCheckInfo` in the **patched code**. However, without this guidance MODIT generated a **wrong patch** by accessing existing parameters from **`this`** object. Essentially, without the guidance, MODIT refactored the input code.

```
// Guidance:  Fix bug of sending wrong message
public void setPredecessor (model.Message m) {
    this.predecessor = Integer.valueOf(m.Content);
    model.Message sent = new model.Message();
    sent.To = m.Origin;
-   sendMessage(m);
    /* MODIT generates with the context. */
+   sendMessage(sent);
    /* MODIT generates without context as input. */
+   sendMessage(m.toString());
}
```

Figure 6.8: Example showing the necessity of context information in predicting the correct patch. MODIT's generated **correct patch** with the context as input. Without context, MODIT received `sendMessage(m)` and the guidance as input, did not know the variable `sent` could be the parameter of the function `sendMessage`, and predicted a wrong patch.

Figure 6.8 shows the effect of context as input modality to MODIT. The before edit version of the code(e_p) passed the wrong parameter (`m`) to `sendMessage` function. When the context

(C) is presented to MODIT, it saw another variable (`sent`) in the context. In contrast, without context(C), MODIT indeed changed the parameter; but sent `m.toString()` — resulting in a wrong patch.

When we extract the buggy code and present the buggy code along with the context, we see a big performance improvement (see the difference between Φ_c, and Φ_{ec} in table 6.3).

We hypothesize that, when only context (*i.e.* full code) is presented (Φ_c), the model gets confused to identify which portion from the context needs to be edited since any portion of the code is a likely candidate for patching. However, when we extract the exact code that needs to be edited and present as a separate input modality to MODIT, it can focus on patching just that code using other modalities (including the context) as a supporting source of information. In a recent study, Ding *et al.* [58] pointed out the need for effective ways to include context in the NMT based code editors. Our empirical results show that MODIT's way of including context as a separate modality is a potential solution to that problem.

In summary, each of the modalities contribute to the overall performances of MODIT. Lessons learned in these experiments are:

- Additional textual guidance helps the patch generation. Such guidance can provide important clue about how to modify the code and sometimes provide ingredients necessary for the change.
- Adding context explicitly in the input enables the model to select appropriate identifiers for patching.
- Isolating buggy code help the model put proper focus on the necessary part of the code while leveraging auxiliary information from other modalities.

Result 6.2: *All three modalities (code to be edited, context, and guidance) are essential for* MODIT *to perform the best. Without either one of those, performance decreases.* MODIT*'s per-*

formance improves up to 37.37% when additional textual guidance is used as an input modality. Context modality improves MODIT*'s performance up to 6.4%.*

Recall from Section 6.2.1, MODIT proposes to encode all the input modalities as a sequence and use one encoder for the consolidated multi-modal input. An alternative to this encoding mechanism is to encode individual input modality with dedicated input encoder. Our next evaluation aims at finding out the best strategy to encode input modalities. Hence, we investigate,

RQ-6.3. What is the best strategy to encode multiple input modalities?

Experimental setup. To validate MODIT's design choice of appending all input modalities into one sequence, we test alternative ways to combine input modalities. In particular, we follow the design choice proposed by Lutellier *et al.* [146], where they used multiple encoders to encode the e_p and the C. Tufano *et al.* [226] also leverages a similar idea to encode input code and code review messages. Nevertheless, we use a multi-encoder model shown in figure 6.9. In a multi-

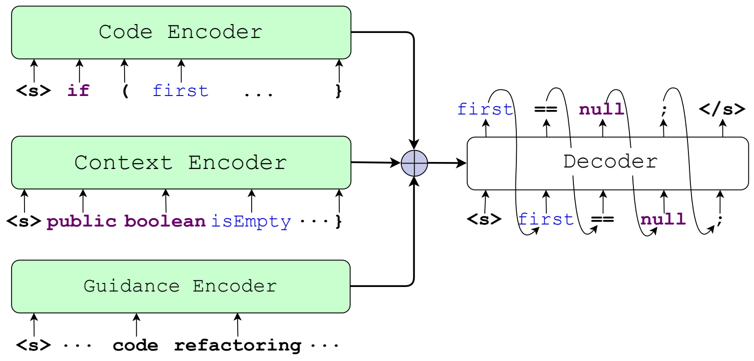


Figure 6.9: An alternative architecture of code editing with multi-encoder model. We initialize each of the encoders with pre-trained Encoder model.

encoder setting, we first encode each input modality with a corresponding dedicated encoder. After the encoder finishes encoding, we concatenate the encoded representations and pass those to the decoder for generating patched code. To retain maximum effectiveness, we initialize each individual encoder with pre-trained weights from CodeBERT [63]. We consider a single-encoder model (also initialized with CodeBERT) as a baseline to compare on the fairground. While presenting the inputs to the single encoder model, we concatenate input modalities with a unique separator token

<s>. Finally, to test the robustness of our empirical finding, we propose two different experimental settings. In the first evaluation setup, we use all three input modalities. We compare a tri-encoder model with a single-encoder model. Next, we consider bimodal input data – e_p and $\mathcal{G}$. We use a dual-encoder model and compare it with a single-encoder model in this setup.

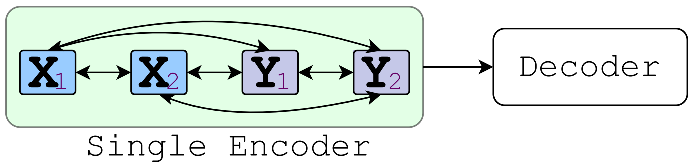
Table 6.4: Comparison of multi encoder model.

# of Modalities	**# of Encoders**	**Accuracy (%)**	
		$B2F_s$	$B2F_m$
3 (e_p, $\mathcal{G}$, C)	3	20.63	11.69
	1	**26.05**	**17.13**
2 (e_p, $\mathcal{G}$)	2	23.12	15.49
	1	**23.81**	**17.46**

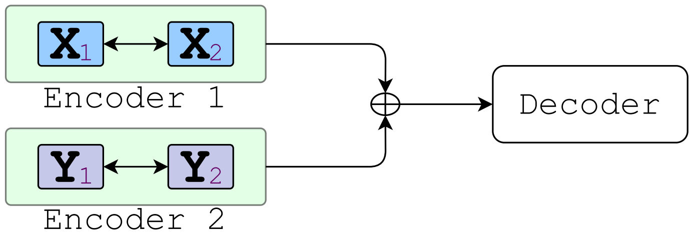
Result. Table 6.4 shows the result of multi-encoder models. For tri-modal input data, if we use three different encoders, the model can predict 20.63% correct patches in the $B2F_s$ and 11.69% correct patches in the $B2F_m$. In contrast, if we use a single encoder, the model's predictive performance increases to 26.05% and 17.13% top-1 accuracy in the $B2F_s$ and the $B2F_m$, respectively.

In the bimodal dataset (where the input modalities are e_p and $\mathcal{G}$), the dual-encoder model predicts 23.12% correct patches in the top-1 position for the $B2F_s$ and 15.49% correct for the $B2F_m$. The single encoder counterpart, in this setup, predicts 23.81% correct patches for the $B2F_s$ and 17.46% for the $B2F_m$. The empirical results show that the single-encoder model performs better in both the experimental setup than the multi-encoder setup. We find similar results with GraphCodeBERT [78].

To explain why single-encoder is performing better than multi-encoder, let us look at the encoders' working procedure. Figure 6.10 depicts how the encoder generates representation for input tokens. Note that the encoders we used in this research question are transformer-based generating representation for an input token by learning its dependency on all other tokens in the sequence. When we present all the input modalities to a single encoder, it generates input representation for



(a) Single encoder for encoding multiple-modalities. Encoder can learn representation *w.r.t.* all modalities.



(b) Dual-encoder for encoding individual modalities separately. Representation of tokens from a particular modality is learned *w.r.t.* (only) other tokens from the same modality.

Figure 6.10: Input token representation generation in single encoder and multiple encoder.

those tokens *w.r.t.* and other tokens in the same modality and tokens from other modalities. For instance, in figure 6.10(a), the encoder generates X_2's representation considering X_1, Y_1, and Y_2. In contrast, in figure 6.10, X_2's representation is learned only *w.r.t.* X_1, since encoder1 does not see the input modality Y. Thus, when we present all the input modalities to one single encoder, we conjecture that learned representations are more robust than that of learning with multi-encoder.

Finally, we summarize the lessons we learned in this research question as

- In multi-modal translation, using single encoder results in better performance than using a separate encoder for each modality.
- Single-encoder generates input representation by inter-modality reasoning (attention), hence learns more robust representation than that of multi-encoder.

Result 6.3: *Encoding all the input modalities by a single encoder is the best way to learn in a multi-modal setting. A single encoder improves code-editing performance by up to 46.5% than the corresponding multi-encoder setting.*

Both in CODIT (§3), and MODIT (this work), we assumed that the location of the code change is given. This assumption may pose a threat to the usefulness of MODIT in a real development scenario. To study the impact, we ask

RQ-6.4. What is the impact of localization in automated code editing

To answer to the research question, we perform an experiment where we pass the whole function as input to MODIT and expect the whole edited function to be generated. Table 6.5 shows the

Table 6.5: Performance of MODIT when the input in the full code and the output is patched full code.

Inputs		**Accuracy (%)**	
Full Code	**Guidance**	$B2F_s$	$B2F_m$
✓	✗	20.35	8.35
✓	✓	**21.57**	**13.18**

top-1 accuracy in the $B2F_s$ and the $B2F_m$. MODIT generates correctly patched full code in 20.35% cases for the $B2F_s$ and 8.35% cases for the $B2F_m$. With additional textual guidance, the performance is further improved to 21.57% and 13.18% in the $B2F_s$ and $B2F_m$, respectively. While textual guidance helps in this experimental setup, we notice a big drop in performance than the results shown in table 6.3. This is because the benchmark datasets we used contain small edits (see table 6.1). Thus, while generating the full code, the model wastes a large amount of effort trying to generate things that did not change. Nevertheless, our hypothesis *external guidance improves code editing* holds even when the model generates full code.

Result 6.4: *Knowing the precise location to edit code significantly improves the performance of automated code editing. When the model knows the precise location, it (i.) precisely knows which portion of the code to put most emphasis on, and (ii.) does not waste resources trying to generate code which did not change.*

6.4.1 Discussion & Threats to Validity

Tokenization for Source Code Processing

Table 6.6: Comparison between concrete tokenization and abstract tokenization alongside pre-trained models. Results are shown as top-1 accuracy of full code generation in $B2F_s$/ $B2F_m$ datasets.

Token type	CodeBERT	GraphCodeBERT	PLBART
Abstract	16.4 / 5.16	**17.30 / 9.10**	19.21 / **8.98**
Concrete	**17.3 / 8.38**	16.65 / 8.64	**20.35** / 8.35

The possible number of source code can be virtually infinite. Vocabulary explosion has been a big challenge while processing source code with Machine Learning technique [224, 110]. Previous research efforts have addressed this problem using several different heuristics. For instance, Tufano *et al.* [224, 225] identifiers abstraction, which drastically reduces the vocabulary size considered making it easier to learn patterns by the model. Recent studies [58, 146, 101, 110] found that Byte-Pair Encoding [210] partially solves the open-vocabulary problem by sub-dividing rare words into relatively less rare sub-words. Such sub-division is also learned from large corpora of data. All the pre-trained models used in this paper used sub-word tokenization techniques. CodeBERT and GraphCodeBERT used RoBERTa tokenizer [138], CodeGPT used GPT tokenizer [187], and PLBART used sentence-piece tokenizer [120]. The use of such tokenizers strips away the burden of identifier abstraction. Our investigation shows that, in some cases, pre-trained models perform better with concrete tokens than abstract tokens (see table 6.6 for detailed result). Thus, we champion using input and outputs with concrete tokens when a pre-trained model is used.

External Validity

Bias in the dataset. Both $B2F_s$, and $B2F_m$ are collection of bug-fix commits, and thus there is a threat that these dataset may exhibit specific bias towards bug-fix patches. While the commits in these datasets are filtered and classified as bug fix commits, these changes are made by real developers as part of development life cycle. Unlike other bugfix datasets [104], $B2F_s$ and $B2F_m$

do not isolate the bug. Thus, we conjecture that possibility of existence of any such bias is minimal.

Noise in commit message. We used commit message as a guidance for code editing. While previous research efforts [250, 251] showed that commit messages are very useful to summarize the changes in a commit, other research efforts [68, 114] also elucidated noises present in the commit message. To mitigate this threat, we carefully chose the dataset we tested MODIT on. The original authors [225] of the the dataset reported that they carefully investigated the dataset and after manual investigation, they reported that 97.6% of the commits in their datasets are true positive. Despite this threat, MODIT's performance seems to improve with commit message as additional input.

Construct Validity

In general, developers write commit message after they edited the code, in theory, summarizing the edits they made. In this paper, we assumed an experimental setup where developer would write the summary before editing the code. Such assumption may pose a threat to the applicability of MODIT in real world, since in some cases, the developer may not know what edits they are going to make prior to the actual editing. Regardless, we consider MODIT as a proof-of-concept, where empirically we show that, if a developer had the idea of change in mind, that could help an automated code editor.

Internal Validity

All Deep Learning based techniques are sensitive to hyper-parameters. Thus using a sub-optimal hyper-parameter can pose a threat to the validity of MODIT, especially while comparing with other baselines. As we compared with other pre-trained models, we cannot really modify the architecture and dimensions of other pre-trained models. As for other hyper-parameters (*i.e.* learning rate, batch size, *etc*), we use the exact same hyper-parameters described by respective paper. Nevertheless, we open source out code and data for broader dissemination.

6.5 Concluding Remarks

In this chapter, we highlight that an automatic code edit tool should possess knowledge about the underlying programming language, in general. Also, it can benefit from additional information such as edit context and developers' intention expressed in natural language. To that end, we design, present, and evaluate MODIT– a multi-modal NMT-based automated code editor. Our in-depth evaluation shows that MODIT improves code-editing by leveraging knowledge about programming language through pre-training. In addition, we showed that leveraging additional modalities of information could benefit the source code editor. Our empirical evaluation reveals some critical lessons about the design choices of building an automated code editor that we believe will guide future research in automatic code editing.

Chapter 7: Generative pre-training by “Naturalizing” source code

7.1 Motivation

Statistical models of the “naturalness" of code [89] have proven useful for a range of Software Engineering tasks [10, 185], including code generation [18], repair [37, 225], summarization [136], retrieval [175], and clone detection [236, 57]. The earlier work in this area trained models directly on tasks, including the early work on type recovery [84, 11], de-obfuscation [195, 227], repair [80], and summarization [97, 4]. Training on-task requires a lot of labeled data. While labeled data is abundant for tasks like code completion (where the corpus inherently provides supervision), other tasks like code generation, translation, summarization, repair, etc., require well-curated, high-quality data. Simply grabbing data from Github might yield poor-quality [75], highly-duplicated data [7]. With increasing model capacity (hundreds of millions, even billions of parameters, are pretty common; larger models tend to perform better [41, 232]), this unacceptable disparity between vast model capacity and the limited availability of well-curated, high-quality, labeled data has increased and will likely worsen.

This shortage of high-quality labeled data for on-task training is not unique to Software Engineering (SE), although it is complicated here by the increased, specialized skill required for labeling SE data. To address the issue of training large models in the presence of data scarcity, such models are often pre-trained on some generic tasks, which relate to actual downstream tasks. For example, consider two SE tasks: code generation and code translation. Both tasks require ML models to learn how to generate natural, syntactically, and semantically correct code. This commonality across tasks motivates a quest for better pre-trained models, using a self- (or un-) supervised task which transfers well to other downstream tasks. Such pre-trained models can also learn a generic representation of the input data, which, in turn, transfers to diverse downstream tasks.

A popular approach for dealing with this problem involves derivatives of BERT style models [54], *e.g.,* CodeBERT [63], GraphCodeBERT [78], etc. These models are good at capturing generic code representations. For code generation tasks, GPT-3 or BART-style models (*e.g.,* Codex, CodeT5, PLBART, SPTCode, etc. [41, 5, 232, 170]) are popular. The important insight here is that independent of final tasks, when *very* high capacity models are trained with huge code corpora to learn simple, self-supervised, "busy work", they still *learn general syntactic and semantic constraints of writing code*. Different approaches adopt different techniques to train the model to write code. For instance, GPT-style models (*e.g.,* Codex) learn to generate code sequentially, mimicking the left-to-right language model. CodeT5 masks out some tokens and asks the model to generate *only* those masked tokens. On the other hand, PLBART and SPT-Code present the model with erroneous code (with deleted or masked tokens) and ask the model to generate the corrected, complete code. The models' ability to generate code depends mainly on the pre-training objective that the model is optimized for.

We propose a novel pre-training task: we ask the model to "naturalize" code, *i.e.* take "weird", synthetic code as input and output semantic equivalent, "natural" code that a human developer would have written. This is a very demanding pre-training task—the model has to learn both code naturalness *and* code semantics. We were inspired by noting the work of human Editors (of books, journals, newspapers): they digest imperfectly written but mostly correct text, understand the intent, and then produce more perfect text with pretty much the same meaning. Editing is *hard*: a skilled Editor has to have very high levels of language comprehension, to understand given, potentially badly-written text, *and then* deploy very high-level writing skills to generate well-formed text. If Editing could be used as an at-scale pre-training task, the learned model would presumably have excellent language comprehension and also generate excellent text. However, it's not obvious how to generate at-scale training data for this "Editing" task, say, for English.

But our concern here is code, not natural language. We start with the argument that, because of the bimodal, dual-channel nature of code [33], it is indeed possible to generate at-scale training

a. Natural Code

```
Scanner sc = new Scanner(...);
while (sc.hasNext()) {
    String ln = sc.next();
    ...
}
...
```

b. Un-natural code

```
Scanner sc = new Scanner(...);
for(; sc.hasNext() ; ) {
    String ln = sc.next();
    ...
}
...
```

Figure 7.1: Example of a natural code fragment written by developers and its 'un-naturally' transformed counterpart. If the `initialization` and `update` part of the `for` loop were to left empty, developers would write the `while` loop.

data for the Editing task (a.k.a. refactoring in Software Engineering terminology). Code has a formal channel, with well-defined semantics; because of this, it's possible to transform code into endless forms, all *meaning-equivalent*. Essentially, we can deploy a set of meaning preserving transformations to *rewrite* existing code from widely-used GitHub projects (which presumably have good-quality code that has passed human code review). These rewrites, (*e.g.,* Figure 7.1), preserve meaning but will make the code into an artificial, often unnatural form[1].

Nevertheless, after rewriting code with de-naturalizing transformation, we now have a matched pair of two semantically equivalent forms of code: a "de-naturalized" form and the original "natural" form. Furthermore, we can produce these pairs at-scale, and then pre-train on a code "Naturalization" task. By analogy with human Editors as described above, such pre-training forces the model to learn two hard things: 1) capture the meaning of the input code, and 2) generate an output that more closely resembles human-written code. We hypothesize that the resulting model will both learn better meaning representations, *and also* generate better code.

To this end, we pre-trained our NetGen model, using "Code Naturalizing" task. NetGen is based on a transformer-based sequence-to-sequence model, and learns to "naturalize" artificially generated "de-naturalized" code back into the form originally written by developers. We *emphasize* that NetGen learns to generate the whole code; this learned skill transfers to downstream fine-tuning tasks that require code generation. We show that our pre-training objective helps model generate more natural code (complete code, with high syntactic and semantic similarity with the

[1]Studies, with human-subjects [35, 34] suggest that humans find such rewritten but semantically identical forms harder to read and understand.

original human-written code). With proper fine-tuning, NetGen achieves state-of-the-art performance in various downstream fine-tuning tasks, including code generation, code translation, bug fix, that demand code generation. We also show that NetGen is specially effective when labelled data is scarce. We summarize our main contributions.

1. We introduce the idea of "Code naturalization" as a pre-training task.
2. Using code from Github, and custom tooling, we have generated and released a large dataset for pre-training models on the Naturalization task.
3. We have built and released a large Sequence-to-Sequence model pre-trained on Naturalization.
4. We show that (when appropriately fine-tuned) NetGen outperforms SOTA on several settings.

7.2 Background & Problem Formulation

This section presents the relevant technical background that leads to this work and an overview of the main research questions.

7.2.1 The Dual Channels of Code

Humans can read and write both natural languages and code. However, unlike natural language, source code involves *two* channels of information: formal & natural [35]. The formal channel, unique to code, affords precise, formal semantics; interpreters, compilers, etc., use this channel. On the other hand, the natural channel (perhaps more probabilistic and noisy) relies on variable names, comments, etc., and is commonly used by humans for code comprehension and communication [34, 35]. The formal channel's precision enables semantic preserving code transformation, which supports static analysis, optimization, obfuscation, *etc*. For instance, major refactoring of a source code may drastically change the syntactic structure while preserving the semantics [67, 57]. However, not all the semantically equivalent code is “natural" [88]—the usual way developers write code and thus, amenable to statistical models [88]. In fact, deviation from such “naturalness" may lead to unintended bugs [190], and increase difficulty of human comprehension [34, 35].

We leverage the natural/formal duality for our pre-training objective in this work. We keep the formal channel constant (not changing the meaning) for a given code and modify the syntax by creating “unnatural” code. Then we train the model to take the “unnatural" code as input and do what a human Editor does with natural language text: understand the “unnatural" code and generate more natural code that a developer would write. Thus, the model simultaneously learns to both comprehend code, *and* generate “natural” code.

7.2.2 “Naturalizing" *vs.* De-noising

Naturalizing pre-training essentially follows in the tradition of *denoising pre-training*, although, arguably, the former is more subtle and challenging. Denoising pre-training [126, 122, 5] is a well-established pre-training strategy for encoder-decoder models: the encoder is presented with a noised-up input, and the decoder is asked to generate the original, noise-free input. By training the model to identify & remove “noise” in a noisy output, (in theory) one teaches it to reason about and correctly generate text. Exactly what a model learns largely depends on the noise types. For instance, PLBART [5] uses syntactic noise[2](*i.e.* token masking, token deletion, etc.). Thus, denoising pre-training enables PLBART to learn both about the syntax of input source code, *and* learn to generate syntactically correct code. Naturalizing pre-training, on the other hand, begins with syntactically correct but artificially-created *unnatural* source code and forces the model to generate correct *semantically equivalent natural* code that is just what a human originally wrote. Such pre-training requires more subtle changes to the code. We hypothesize that this provides a more demanding pre-training setting, which will lead to better on-task code generation performance.

[2]Noise that breaks the syntax structure of code

7.2.3 Research Questions

Our hypothesis is that our *naturalizing* task (see section 7.3.1) endows our pre-trained model with the ability to generate syntactically and semantically correct, *and* natural code. This leads to several RQs.

RQ-7.1. Does "Naturalization" help to improve code generation?

In contrast to existing de-noising techniques [5] that help the model learn lexical & syntactic structure, the naturalizing task, which is arguably more demanding than de-noising, forces NetGen generating better code with higher syntactic and semantic correctness.

The pre-training data we use (in NetGen) challenges the model to naturalize code that was "de-naturalized" in several ways, such as dead-code inserted, variable renamed, etc. We investigate the relative performance under different naturalization challenges.

RQ-7.2. How do different components in NetGen contribute to code generation?

We evaluate the performance under different challenges on a held-out validation dataset. This dataset is sampled with the same distribution of de-naturalizing transforms as the training dataset ($\mathcal{D}_t$); on this set, the model to reconstruct the original code. Our exploratory investigation reveals that Variable Renaming is the hardest transformation to undo: the model reconstructs original code with only 40% accuracy. Dead Code, on the other hand, is the easiest with 99% accuracy.

We further investigate NetGen's performance for downstream source code generation tasks.

RQ-7.3. How effective is NetGen when fine-tuned for different generative tasks in source code?

We fine-tune the pre-trained NetGen on task-specific training dataset for a certain time budget and evaluate the fine-tuned model on the benchmark testing dataset for corresponding task. These tasks include source code (java) generation from text, code translation (from Java to C# and C# to Java), and Bug fixing. After fine-tuning, NetGen achieves the state-of-the-art performance in all these tasks. In addition, we also discover that, code generated by NetGen are syntactically and semantically more closer to the expected code.

We observe that training a model for a complex task requires sufficient labeled data. However, for most software engineering tasks, finding labeled data is a significant challenge [6]. We investigate potential scenario where size of the training data is extremely small.

RQ-7.4. How well does NetGen's pre-training help in tasks where labelled data is scarce?

We simulate training data scarcity in two different ways – *Zero-shot learning*, and *Few-shot learning*. For "Zero-shot" learning, we evaluate the pre-trained NetGen in different tasks *without* any task specific fine-tuning. For "few-shot" setting, we simulate training data scarcity by subsampling the benchmark training datasets. We fine-tune the pre-trained NetGen on these limited training examples and measure the performance. We observe that NetGen is very efficient in low-data training. Since NetGen learns to generate syntactically and semantically correct code as part of pre-training, it faces less burden while learning in low-data training.

7.3 Methodology

Our approach comprises three steps: (i) "De-Naturalize" source code to accumulate pre-training data for NetGen (§7.3.1); (ii) pre-train NetGen using this data for naturalization task (§7.3.2); (iii) Fine-tune pre-trained NetGen with task specific dataset (§7.3.3).

7.3.1 De-Naturalizing Source Code

For the first step above, we use six rules to transform a natural code into its unnatural counterpart. These transformations are semantic-preserving but rewrite an original, natural, (human-) written code to an artificial form. Given a natural code element, we deploy an appropriate transformation, based on its AST structure and rewrite the code to "de-naturalize" it.

```
int search(int[] arr, int key, int low, int high){
    while (low <= high) {
        int mid = low  + ((high - low) / 2);
        if(arr[mid] == key)  { return mid; }
        else { high = mid + 1; }
    }
    return -1;
}
```

Listing 7.1: Original Code

```
int search(int[] arr, int key, int low, int high){
    for (;low <= high;) {
        int mid = low  + ((high - low) / 2);
        if(arr[mid] == key) { return mid; }
        else { high = mid + 1; }
    }
    return -1;
}
```

Listing 7.2: Loop Transformation

```
int search(int[] arr, int key, int low, int high){
    while (low <= high) {
        int mid = low  + ((high - low) / 2);
        while ( i< i ) {
            high = mid + 1;
        }
        // ... Rest of the Code
    }
    return -1;
}
```

Listing 7.3: DeadCode Insertion

```
int search(int[] arr, int key, int low, int high){
    while (high >= low) {
        int mid = low  + ((high - low) / 2);
        if(arr[mid] != key)  {
            high = mid + 1;
        }
        else{ return mid; }
    }
    return -1;
}
```

Listing 7.4: Block and Operand Swap

Figure 7.2: Semantic preserving transformation used to prepare the pre-training data for NetGen (Part-1).

```
int search(int[] arr, int key, int low, int high){
    while (low <= high) {
        int mid = low  + ((high - low) / 2);
        if(arr[mid] == key)  { return mid; }
        else { high = mid + 1; }
    }
    return -1;
}
```

Listing 7.5: Original Code

```
int search(int[] arr, int key, int low, int high){
    while (low <= high) {
        int mid = low  + ((high - low) / 2);
        if(arr[mid] == key) { return mid; }
        else {
            high = mid++;
        }
    }
    return -1;
}
```

Listing 7.6: Inserting confusing code element

```
int search(int[] var_1, int key, int low, int var_2){
    while (low <= var_2) {
        int mid = low  + ((var_2 - low) / 2);
        if(var_1[mid] == key) { return mid; }
        else { var_2 = mid + 1; }
    }
    return -1;
}
```

Listing 7.7: Variable Renaming

Figure 7.3: Semantic preserving transformation used to prepare the pre-training data for NetGen (Part-2).

Designing Transformation Rules.

We use six classes of de-naturalizing transformations. These transformations are motivated by prior work on functional reasoning about source code [57, 74, 73] and semantic bug-seeding [177]. Figure 7.2 and figure 7.3 show the details.

Loop Transformation (Listing 7.2). This rule modifies `for` loops into equivalent `while` loop and vice-versa. We rewrite a `while` loop of the form `while (` $\boxed{\texttt{condition}}$ `) { loop-body }` into a `for` loop as `for (` $\boxed{\texttt{;}}$ $\boxed{\texttt{condition}}$ $\boxed{\texttt{;}}$ `) { loop-body }`. Likewise, to transform a `for` loop into a `while` loop, we move the initializer of the `for` (if any) before the loop, and the update *expression* (if any) of the `for` loop as the last *statement* in the loop. We also add this update statement before any loop breaking statement (*i.e.* `break`, `continue`). For example, we transform “`for(`$\boxed{\texttt{int i = 0;}}$ `i < 10;` $\boxed{\texttt{i++}}$`){ if(i){ foo(); continue;} bar(); }`” as “$\boxed{\texttt{int i = 0;}}$ `while(i < 10){ if(i){ foo();` $\boxed{\texttt{i++;}}$ `continue;} bar();` $\boxed{\texttt{i++;}}$`}`”.

Dead Code Injection (Listing 7.3). We inject blocks of dead code at random positions in the original code. By “dead code" we mean code that appears in the source but is never executed. In Listing 7.3, we inject the code block `high = mid + 1;` at line 4 of the original code (listing 7.1). To add challenge to the model, we transplant these inserted statements from the same input code. To ensure the "death" of inserted code, we put the inserted statements in a block headed by either a loop or a branch, guarded by a unsatisfiable condition so that the code inside the block will never execute. In Listing 7.3, the condition $\boxed{\texttt{i < i}}$ is always **`false`** ; and the code in line 5 is quite dead.

Block Swap (Listing 7.4). Here we swap the “then" block of a chosen `if` statement with the corresponding `else` block. To preserve semantic equivalence, we negate the original branching condition. For instance, listing 7.4 replaces the `if` block (line 4 in Listing 7.1) with the `else` block (line 5 in listing 7.1). We negate the original condition (`arr[mid] == key`) as (`arr[mid] != key`).

Operand Swap (Listing 7.4). Here, we swap the operands of binary logical operations. For instance, we change the expression `low <= high` with `high >= low` in line 2 in Listing 7.4. When swapping the operands of a logical operator, we change the operator to make sure the modified expression is the logical equivalent to the one before modification. In case of asymmetric inequality operators (>, <, >=, <=), we change the direction – keep as is for symmetric operators (*i.e.* ==, ! =).

Confusing Code Insertion (Listing 7.6). We introduce confusing code patterns in the code as outlined by [74, 73]. In particular, we introduce two forms of confusing code. First, we modify the of the form `{i = j; j += 1;}` to `i = j++;` . Second, we introduce ternary operator as applicable. For example, we transform the code `if (x != 0){y = p;} else {y = q;}` to `y = (x != 0)? p : q;` .

Variable Renaming (Listing 7.7). We rename some variables to `VAR_i`. While renaming a variable, we analyze the dataflow of that variable and rename all occurrences of that variable in the entire code. From all the variables used in the code, we change just a certain percentage. For instance, in Listing 7.7, we renamed variable `arr` to `var_1` , and variable `high` to `var_2` , leaving all other variables unchanged. Note that, unlike other transformations, variable renaming does not create AST of Dataflow graph difference. However, this challenging task [12] forces the model to learn to generate natural variable names. This resembles the de-obfuscation pre-training task of [203].

Applying Transformation.

Assume a set of transformation rules $\Phi = \{\phi_1, \phi_2, \phi_3, ...\}$. Given original code c_i, $\phi_j(c_i)$ transforms the code, changing the structure while preserving semantics. Figure 7.4 shows how to apply such transformation to c_i. It works in three steps:

- *Find Transformation Location.* Given a piece of source code (c_i), we first use tree-sitter[3] to parse out the AST (T_{c_i}). From the AST, we extract potential locations for de-naturalization.

[3]`https://tree-sitter.github.io/tree-sitter/`

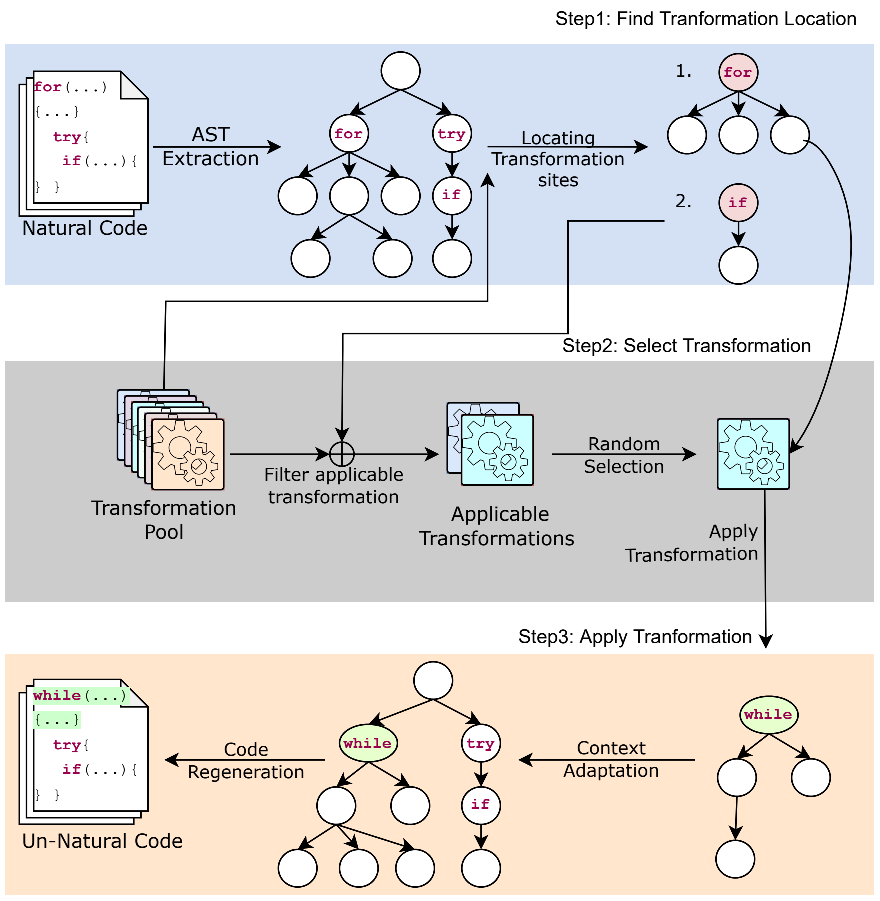


Figure 7.4: "De-Naturalization" workflow in NetGen.

These locations are nodes (n_k) in T_{c_i}. While choosing location n_k from T_{c_i}, we consult Φ – we extract the nodes where at least one of $\phi_j \in \Phi$ is applicable.

- *Select Transformation Rule.* Once we have a set of such nodes, we filter out the transformation rules that cannot be applied to any node of in T_{c_i}. After such a filtration, we have a set of transformations $\Phi_a \subseteq \Phi$. At this stage, we randomly select one transformation pattern $\phi_j \in \Phi_a$ to apply at an application location (AST node) n_k.

- *Apply Transformation.* We apply ϕ_j to n_k to get the transformed node n'_k. We then structurally match n'_k with the original AST T_{c_i}, specifically n_k. We adapt the context of n_k to the transformed node's (n'_k) context. In that way, we get the transformed AST (T'_{c_i}), which we then translate to get the transformed code c'_i.

We designed the transformation function ϕ_j and subsequent context adaptation in such a way that preserves the meaning or functionality of the original code. We use AST analysis and (approximated) data flow analysis on code AST.

7.3.2 Pre-training

Once we have a pool of "unnatural" code using the transformation in Section 7.3.1 (*i.e.* transform code c_i as 'un-natural' code $\phi_j(c_i)$), we use a neural sequence-to-sequence translation model ($\mathcal{M}$) to reconstruct c_i from $\phi(c_i)$, *i.e.* we want $\mathcal{M}(\phi_j(c_i))$ to approximate c_i . In particular, given a training dataset $\mathcal{D}_t = \{c_1, c_2, ...\}$ consisting of developers written code, set of "de-naturalizing" transformations $\Phi = \{\phi_1, \phi_2, \phi_3, ...\}$, we optimize the following function to learn $\mathcal{M}$'s optimal parameter Θ.

$$\Theta = \arg\min_{\theta} \sum_{c_i \in \mathcal{D}_t} CrossEntropy\left(\mathcal{M}\left(\phi_j\left(c_i\right)\right), c_i\right) \tag{7.1}$$

7.3.3 Fine-tuning

The objective of our pre-training is to learn to both comprehend and generate general-purpose source code. However, different tasks related to source code generation (*e.g.,* text to code generation, code to code translation, bug fixing) call for task-specific training of the pre-trained model. This training phase on a pre-trained model is known as fine-tuning [54]. We consider the fine-tuning in NetGen as a translation task and follow the standard transformer based-machine translation procedure [228]. First, the encoder generates the encoded representation $R(X)$ given the input $X = [x_1, x_2, ..., x_n]$. The decoder then sequentially generates the output $Y = [y_1, y_2, ..., y_m]$. While encoding an input token x_k, the encoder learns the attention matrix *w.r.t.* every token in the input, including x_k. Such attention matrix is known as *self-attention*. While generating an output token y_m, the decoder learns the attention matrix with all previously generated tokens $[y_1, y_2, ..., y_{m-1}]$ through *self-attention* and the encoder generated representation $R(X)$ through *cross-attention*. We refer to Vaswani *et al.* [228] for more detail about transformer-based translation.

7.4 Experimental Design

This section details the experimental design of NetGen.

Pre-training data. Following prior works [63, 78, 232], we primarily use CodeSearchNet [93] dataset for the pre-training purpose. CodeSerachNet is a publicly available dataset with six languages: Java, Python, Go, JavaScript, Ruby, and PHP. In addition to CodeSearchNet, CodeT5 uses additional data for C and C#. We also use 1M functions each for C and C#. For these two additional languages, we collected 5000 active projects from GitHub and randomly selected 1M functions considering the maximum sequence length of the model.

Table 7.1: Statistics of fine-tuning datasets.

	Task	**Dataset**	**Train#**	**Dev#**	**Test#**
Text$\rightarrow$Code	Generation [96]	Concode	100000	2000	2000
Code$\rightarrow$Code	Translation [142]	CodeXGLUE	10300	500	1000
Text+code$\rightarrow$Code	BugFix [223]	Small	46628	5828	5831
		Medium	53324	6542	6538

Fine-tuning data. We evaluate different variations of three benchmark tasks related to source code generation. The first task is *Text to Code generation*, where the input is an NL description of a Java method, and the output is the code. The second task is *Code Translation* between *Java to C#* and *C# to Java*. For this task, we evaluate Java-C# parallel dataset proposed by CodeXGLUE [142]. The third and final task is *Bug Fix*, where the given a buggy code and a summary of the fix model generates the fixed code. For this task, we used the two different versions of the dataset (small, with less than 50 tokens and medium with up to 100 tokens) proposed by Tufano *et al.* [223]. Note that, similar to MODIT [39], we evaluate on *concrete* [4] version of the refinement datasets.Table 7.1 shows the datasets and their statistics. For Text to Code Generation

[4]In contrast to the original 'Abstract' edit dataset proposed by Tufano *et al.* [223], we continue using the 'concrete' edit dataset with concrete identifier names. See out comparison between 'concrete' and 'abstract' edits in §6.4.1

and Code Translation, we reuse the same split from CodeXGLUE [142], and for Bug Fix, we reuse the same split as MODIT.

Pre-training Model Configurations. We use 12 layer transformers with 12 attention heads on both encoder and decoder following the CodeT5 [232] architecture. As discussed in Section 6.3, we use de-naturalization generative objectives for pre-training. We initialize our model with CodeT5's [232] released parameters. In particular, we initialize NetGen with "CodeT5-base" model. We pre-train NetGen on 2 Nvidia GeForce RTX 3090 GPUs for 25K steps, maintaining the effective batch size at 1080 with learning rate 5e-5. We train NetGen for approximately 168 hours.

Evaluation Metric. Throughout the experiments in this work, we evaluate accuracies w.r.t. exact match (EM), Syntax match (SM), Dataflow match (DM), and CodeBLEU (CB) [197]. SM is the proportion of matching subtrees between output code and tadget code's ASTs *w.r.t.* number of all possible subtrees in the target code's AST. DM is the percentage of matched (with target code) anonymized dataflow edge (def-use edge) of output code *w.r.t.* all dataflow edges in the target code. Note that, both the SM and DM are components of CB. We explicitly evaluate these for understanding the syntactic and semantic correctness of generated code. We reuse Microsoft CodeXGLUE tool [155] to compute SM, DM, and CB.

Baselines. While comparing the evaluation results for different tasks, we compare with large scale pre-trained models, including GPT-2 [187], CodeGPT [142], PLBART [5], SPT-Code [170] and CodeT5 [232]. Most of our fine-tuning evaluation is on benchmarked dataset; thus, we report the available results from CodeXGLUE leaderboard [156]. There are some task specific baselines, which we discuss while describing corresponding task.

7.5 Research Findings

We evaluate NetGen on (i) pre-training and (ii) three fine-tuning tasks. We also check NetGen's effectiveness in zero-shot and few-shot settings.

7.5.1 NetGen's Effectiveness on pre-training

RQ-7.1. Does "Naturalization" help to improve code generation?

Motivation. We investigate whether pre-training on naturalizing task helps the model generate correct and natural code (code that is syntactically and semantically similar to the original code).

Experimental Setup. We compare three large scale pre-trained models: (i) CodeT5 [232], (ii) PLBART [5], and (iii) NetGen. Note that, since PLBART is only pre-trained on Java and Python, we compare PLBART only for those languages, with the corresponding results of other models. We ask each of these models to reconstruct developers' written code from its de-naturalized (but semantically identical, see §7.3.1 & §7.3.1) variants. We use the held-out validation data from our training procedure for this evaluation. We evaluate the models for generating the Exact Match (EM), Syntax Match (SM) and Dataflow Match (DM).

Table 7.2: Evaluation of NetGen for code generation task. CS is the percentage of examples where output is directly copied from source, and ED is the median edit distance between input code and output code.

Eval Data	Model	EM	SM	DM	CB	CS	ED
Full	CodeT5	0	13.93	19.86	9.74	0%	60
	NetGen	**70.39**	**98.78**	**97.69**	**97.31**	0.01%	8
Java & Py	CodeT5	0	13.83	23.67	10.87	0%	65
	PLBART	0	73.17	75.95	74.56	7.05%	3
	NetGen	64.13	98.16	96.85	96.82	0.01%	10

Results. Table 7.2 shows the evaluation results.

• *Syntax Match.* We find that the code generated by PLBART and NetGen are mostly syntactically correct. However, CodeT5's does not always generate syntactically valid code, suggesting an advantage for naturalization pre-training. For instance, Figure 7.5 shows code generated by different models from the given input. As we can see, CodeT5 generates a syntactically erroneous fragment. In contrast, PLBART made a minor edit on the input code, just removing the `protected` keyword. Both PLBART and NetGen are pre-trained to generate complete code rather than fragments (which is the case of CodeT5 [188]); thus, the former two generally do better at generating syntactically correct code.

Example : 1

1. Input

```
protected SDV iam(SDV in,...){
    if(i < i){
        return new IAM(...);
    }
    return new IAM(...);
}
```

2. PLBART output

```
SDV iam(SDV in, ...){
    if(i < i){
        return new IAM(...);
    }
    return new IAM(...);
}
```

3. NetGen output

```
protected SDV iam(SDV in,...){
    return new IAM(...);
}
```

4. CodeT5 output

```
if (in) {
     return
}  }
```

Figure 7.5: Example of input and generated code by different pretrained models.

• *Semantic Match.* NetGen is effective at recovering developers' written code from its de-naturalized semantic variants—around 70% of the generated code (CodeBlue = 97%) *exactly matches* the original code. PLBART, which deploys syntactic denoising, is at the second position in terms of CodeBlue.

NetGen also dominates the other two models in generating syntactically (SM) & semantically (DM) valid code. While PLBART appears to generate syntactically correct code, it mostly copies code from the input—median edit distance from PLBART's input and the generated code is 3 (see Table 7.2). In fact, in 7.05% of cases, PLBART just copies the input! By contrast, NetGen learns to generate *variants* of the input code, with only 0.01% direct copy and a median edit distance of

10. Since PLBART is trained to remove syntax errors from the input, we conjecture that it does not inherently learn the semantic variation of the code. By contrast, we expose NetGen to semantic code variations, forcing it to learn to generate code that is both more natural <u>*and*</u> semantically equivalent.

Example : 2

1. Input

```
int maxVal ( int value1 , int VAR_2
    , int target ) {
     if ( value1 > VAR_2 ) {
          for ( int i = value1 ; i <
              value1 ; i ++ ) {
               return value1 > target ?
                    value1 : target;
          }
          return value1 ;
     }
     else
          return VAR_2 > target ?
              VAR_2 : target ;
}
```

2. PLBART output

```
maxVal ( int value1 , int VAR_2 ,
    int target ) {
     if ( value1 > VAR_2 ) {
          for ( int i = value1 ; i <
              value1 ; i ++ ) {
               return value1 > target ?
                    value1 : target;
          }
          return value1 ;
     }
     else
          return VAR_2 > target ?
              VAR_2 : target ;
}
```

3. NetGen output

```
int maxVal ( int value1 , int
    value2 , int target ) {
     if ( value1 > value2 ) {
          for ( int i = value1 ; i <
              value1 ; i ++ ) {
               return value1 > target ?
                    value1 : target;
          }
          return value1 ;
     }
     else return value2 > target ?
         value2 : target ;
}
```

4. CodeT5 output

```
if ( value1 > VAR_2
     value1 = VAR_2 ;
     value1 = VAR_2 ;
     value1 ;
     }
     return value1 ;
  } }
```

Figure 7.6: Example of input generated code by different pre-trained models.

• *Closer look into CodeT5.* Unlike NetGen and PLBART, CodeT5 is not explicitly trained to generate complete code. During pre-training, CodeT5 learned to "unmask" masked token se-

Example : 3

1. Input

```
int maxVal ( int value1 , int value2
     , int target ) {
     if ( value1 > value ) {
          for ( int i = value1 ; i <
              value1 ; i ++ ) {
               return value1 > target ?
                    value1 : target;
          }
          return value1 ;
     }
     else
          return value2 > target ?
              value2 : target ;
}
```

2. PLBART output

```
maxVal ( int value1 , int value2 ,
    int target ) {
     if ( value1 > value ) {
          for ( int i = value1 ; i <
              value1 ; i ++ ) {
               return value1 > target ?
                    value1 : target;
          }
          return value1 ;
     }
     else
          return value2 > target ?
              value2 : target ;
}
```

3. NetGen output

```
int maxVal  ( int value1 , int
    value2 , int target ) {
     if ( value1 > value ) {
          for ( int i = value1 ; i <
              value1 ; i ++ ) {
               if ( value1 > target ) {
                    return value1 ;
               }
               else {
                    return target ;
               }
               return value1 ;
          }
     }
     else if ( value2 > target ) {
          return value2 ;
     }
     else {
          return target ;
     }
}
```

4. CodeT5 output

```
int int i = value2 ; i ++ ) {
     return value2 ;
}
```

} }

value1 value2 ; value1 ; } value2 ; } }

Figure 7.7: Example of input generated code by different pre-trained models (contd.)

quences. Thus, to better measure CodeT5's generation capacity, we conduct another experiment where we replaced all occurrences of some of the variable names in code with a special `MASK1`, `MASK2` tokens and asked CodeT5 to generate. This is one of the objectives (masked identifiers

Example : 4

1. Input

```
int maxVal ( int value1 , int value2
     , int target ) {
     if ( value1 > value ) {
          return value1 ;
     }
     else
          return value2 > target ?
              value2 : target ;
}
```

2. PLBART output

```
maxVal ( int value1 , int value2 ,
    int target ) {
     if ( value1 > value ) {
          return value1 ;
     }
     else
          return value2 > target ?
              value2 : target ;
}
```

3. NetGen output

```
int maxVal ( int value1 , int value2
     , int target ) {      if (
    value1 > value ) {
          return value1 ;
     }
     else if ( value2 > target ) {
          return value2 ;
     }
     else {
          return target ;
     }
}
```

4. CodeT5 output

```
private int } int ( ) {
     return value ;
} } ( ) {
     return
} } }
```

Figure 7.8: Example of input generated code by different pre-trained models (contd.)

prediction) CodeT5 is pre-trained to optimize. We take the CodeT5's output and identify all potential identifiers [5]. Surprisingly, in *only* 0.27% of the cases, could CodeT5 generate *all* the variables, and in 0.61% of cases *half* of the masked variables., while NetGen successfully translates 40.45% of those examples back to its original code, including correctly predicting the replaced variable names. In addition, CodeT5's generated token sequence contained a lot of other tokens than the variable names (figure 7.5.4, for example). More example code generation are shown in Figures 7.6, 7.7, 7.8, 7.9.

Result 7.1: *Naturalization enables NetGen to reason about code semantics and thus help generate more natural code variants than existing pre-training models and pre-training objectives.*

We also did an ablation study evaluating the effect of NetGen's different components on the

[5]we use regex `"[A-Za-z_]+[A-Za-z0-9_]*"` to find identifiers.

Example : 5

1. Input

```
int maxVal ( int value1 , int value2
    , int target ) {
    if ( value1 > value ) {
        return value1 ;
        for ( int i = value1 ; i <
            value1 ; i ++ ) {
            return value1 ;
        }
    }
    else return value2 ;
}
```

2. PLBART output

```
maxVal ( int value1 , int value2 ,
    int target ) {
    if ( value1 > value ) {
        return value1 ;
        for ( int i = value1 ; i <
            value1 ; i ++ ) {
            return value1 ;
        }
    }
    else return value2 ;
}
```

3. NetGen output

```
int maxVal ( int value1 , int value2
    , int target ) {
    if ( value1 > value ) {
        return value1 ;
        int i = value1 ;
        while ( i < value1 ) {
            return value1 ;
            i ++ ;
        }
    }
    else
        return value2 ;
}
```

4. CodeT5 output

```
int for ( int i = value2 ; ivalue2 ;
    } } intvalue1 ; value1 ; value2
    ; value1 ; value2 ; } }
```

Figure 7.9: Example of input generated code by different pre-trained models (contd.)

results.

RQ-7.2. How do different components in NetGen contribute to code generation?

Motivation. In this RQ, we study how different transformation rules (see §7.3.1)contribute to learn generating natural code from different semantic variants . We also evaluate how well NetGen learns that in different programming languages over training time.

Experimental Setup. While pre-training, we checkpoint the NetGen model every 1k training steps, for a full run of 25k steps. At each checkpoint, we evaluate the naturalization task performance. Before training, we held out 0.1% of the total data as validation data. Note that, since our goal in this experiment is to understand NetGen's pre-training better, we "de-naturalized" the validation data using the same training data distribution. This setting gives us a controlled environment for experimentation.

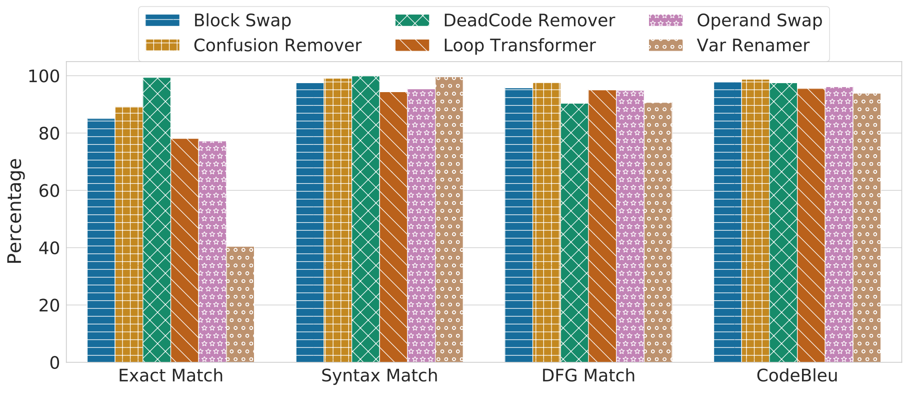
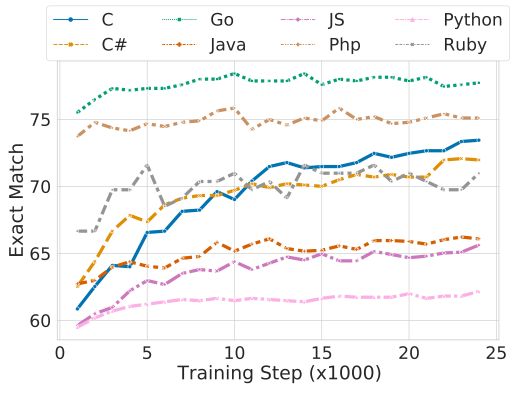


Figure 7.10: Performance of NetGen pre-trained model under different code transformations.

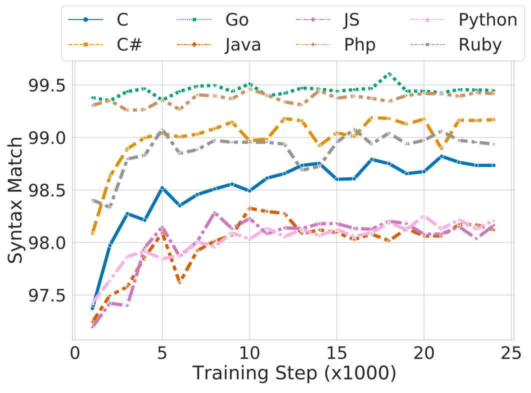
<u>**Results.**</u> Figure 7.10 shows NetGen's performance under different types of semantic variants. Results show that NetGen has most trouble recreating the original code (just 40% Exact Match) with the variable renaming task. Variable renaming is challenging even for human developers [9]—different developers may propose different names for the same object. Nevertheless, on this task, NetGen achieves good syntax and dataflow match (99% and 92% respectively), indicating that NetGen preserves syntax or semantics in most cases while generating code with renamed variables.

On the other hand, NetGen can eliminate Dead Code with 99% accuracy. This result may be an artifact of our specific implementation of this transformation. Our dead-code insertion rule is simple, and formulaic; so the NetGen quickly learns to identify and remove such dead code. A more complex pattern of dead code may challenge the model more, and help make it more robust; we leave this for future work. For naturalizing other transformations, NetGen achieves more than 80% exact match accuracy for Block swap and Confusion removing, and more than 75% exact match accuracy for the rest. In all cases, syntax match, dataflow match, and CodeBLEU are well above 90%.

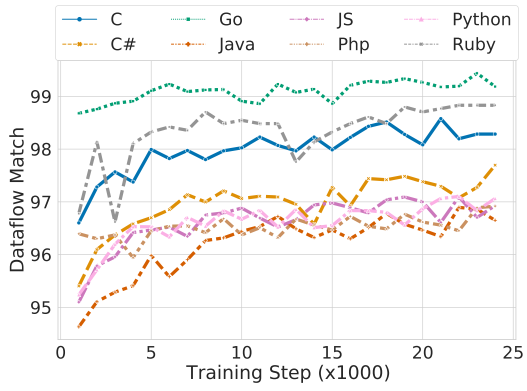
7.5.2 Pre-training progression for all metrics



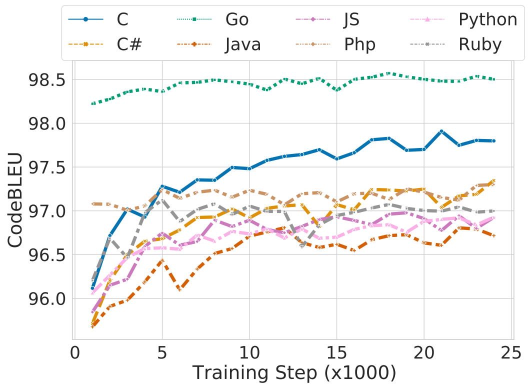
(a) Exact Match Accuracy



(b) Syntax Match Accuracy



(c) Dataflow Match Accuracy



(d) CodeBLEU

Figure 7.11: Progression of Different metrics of different language in Validation dataset over number pre-training steps.

Figure 7.11 shows how validation performance improves for different languages, with more training steps. Across all the languages the performance rapidly increases over the first few thousand training steps. In fact, at the beginning of (step 0) of NetGen's pre-training, the overall exact match is 0, syntax match is 13.93%, dataflow match is 19.86% and CodeBLEU is 9.74% (see Table 7.2 for details[6]). However, after just 1000 steps of training, the exact match rises to 61%, syntax match to 97%, dataflow match to 94%, and CodeBLEU to 95%. These metrics continue improving

[6]NetGen's pre-training start from CodeT5-base. Thus, CodeT5-base is NetGen's checkpoint at step 0.

as training progresses. These results confirm that across all the languages NetGen gradually learns to generate more natural code from semantic variants.

Result 7.2: *Pre-training performance depends on the types of semantic variants—while variable renaming seems the most difficult (~40% accuracy), dead-code elimination appears to be an easier task (~99% accuracy) to learn.*

7.5.3 NetGen's Effectiveness on Fine-tuning Tasks

This section evaluates NetGen's performance on three benchmark source code generative tasks.

RQ-7.3. How effective is NetGen when fine-tuned for different generative tasks in source code?

Table 7.3: Results of Text to Code Generation. '-' implies that those results are not reported by corresponding approaches. $\mathcal{M}_{last}$ is the model after completing the fintuning, and $\mathcal{M}_{best}$ is the intermediate model with best validation performance.

Approach		EM	SM	DM	CB
Seq2Seq		3.05	-	-	26.39
Guo *et al.* [77]		10.05	-	-	29.46
Iyer *et al.* [96]		12.20	-	-	-
GPT-2		17.30	-	-	29.69
CodeGPT		20.10	-	-	35.98
PLBART		18.75	-	-	38.52
CodeT5-base (reported)		**22.30**	-	-	43.20
CodeT5*	$\mathcal{M}_{last}$	21.85	44.34	44.52	41.75
	$\mathcal{M}_{best}$	21.55	41.08	43.71	38.30
NetGen	$\mathcal{M}_{last}$	22.25	**45.59**	**46.87**	**43.73**
	$\mathcal{M}_{best}$	**22.30**	44.38	45.64	42.44

* Our reproduced result using CodeT5's publicly available pre-trained model.

Baselines. In addition to the baselines discussed in Section 7.4, for the *Text to Java Code generation* task, we compare with a group of baselines with no pre-training involved. These baselines include LSTM based Sequence to sequence models, Guo *et al.* [77]'s, and Iyer *et al.* [96]'s proposed techniques. We also report our reproduced version of CodeT5 results in different tasks, slightly different from what they reported. For both the *Bug Fix* task, we compare with the reported results of MODIT [39] and our reproduced CodeT5 result.

Results.

Text to Code Generation. Table 7.3 shows evaluation results for text to code generation. We trained for 30 epochs. We stopped the training is the validation performance does not increase for more than three(3) consecutive epochs. For both CodeT5 and NetGen, we report the performance of final model after the fine-tuning terminated ($\mathcal{M}_{last}$) and the performance of the model with best validation perfomance ($\mathcal{M}_{best}$). Interestingly, for both CodeT5 and NetGen, the $\mathcal{M}_{last}$ model performs better than the corresponding $\mathcal{M}_{best}$ model. The result shows that NetGen's generated code are more syntactically and semantically closer to the target code. The $\mathcal{M}_{last}$ model of NetGen outperforms CodeT5's $\mathcal{M}_{last}$ model by 2.8% in SM, 5.28% in DM and 4.74% in CB. Such improvements are statistically significant with more than 99.99% confidence (see Table 7.5). We conjecture that NetGen's pre-training with "naturalization" help generate more natural code.

Table 7.4: Code Translation results. '-' implies that those results are not reported by corresponding approaches.

Approach	Java → C#				C# → Java			
	EM	SM	DM	CB	EM	SM	DM	CB
PBSTM	12.5	-	-	42.7	16.1	-	-	43.5
CodeBERT	59.0	-	-	85.1	58.8	-	-	79.4
SPT-Code	64.1	-	-	-	60.2	-	-	-
PLBART	64.6	-	-	87.9	65.0	-	-	**85.3**
CodeT5 (reported)	65.9	-	-	-	66.9	-	-	-
CodeT5*	65.9	90.4	91.9	87.8	66.0	90.4	88.9	84.4
NetGen	**66.2**	**91.0**	**92.0**	**88.1**	**67.3**	**91.0**	**89.8**	85.2

* Our reproduced result using CodeT5's publicly available pre-trained model.

Code Translation. Table 7.4 shows the results of NetGen and different baselines for Code Trans-

Table 7.5: One sided Wilcoxon sign rank test statistics result comparison between NetGen and CodeT5 across different tasks and metrics. Highlighted results are statistically significant with >95% confidence.

Task	**Metric**	**P-Value**
Code generation (Text to Java)	Syntax Match	<0.001
	Dataflow Match	<0.001
	CodeBLEU	<0.001
Translation (C# to Java)	Syntax Match	0.093
	Dataflow Match	0.147
	CodeBLEU	0.016
Translation (Java to C#)	Syntax Match	0.166
	Dataflow Match	0.462
	CodeBLEU	0.021

lation. For Java to C# translation, NetGen achieves exact match accuracy of 66.2% while CodeT5's accuracy is 65.9%. In C# to Java translation, NetGen achieves 67.3% exact match accuracy, which CodeT5 achieves 66.0%. In addition, the syntactic match (SM), Dataflow match (DM), and CodeBLEU are also higher than that of CodeT5. While the improvement in SM and DM are not statistically significant, the improvement in CodeBLEU is statistically significant with more than 97% confidence (Table 7.5). Since Java and C# are very similar languages, we believe translating across PL *w.r.t.* these datasets are not very challenging, which is also corroborates by high performance of other baselines. Nevertheless, the statistically significant improvement on CodeBLEU suggest that NetGen is overall learning slightly better translation than CodeT5.

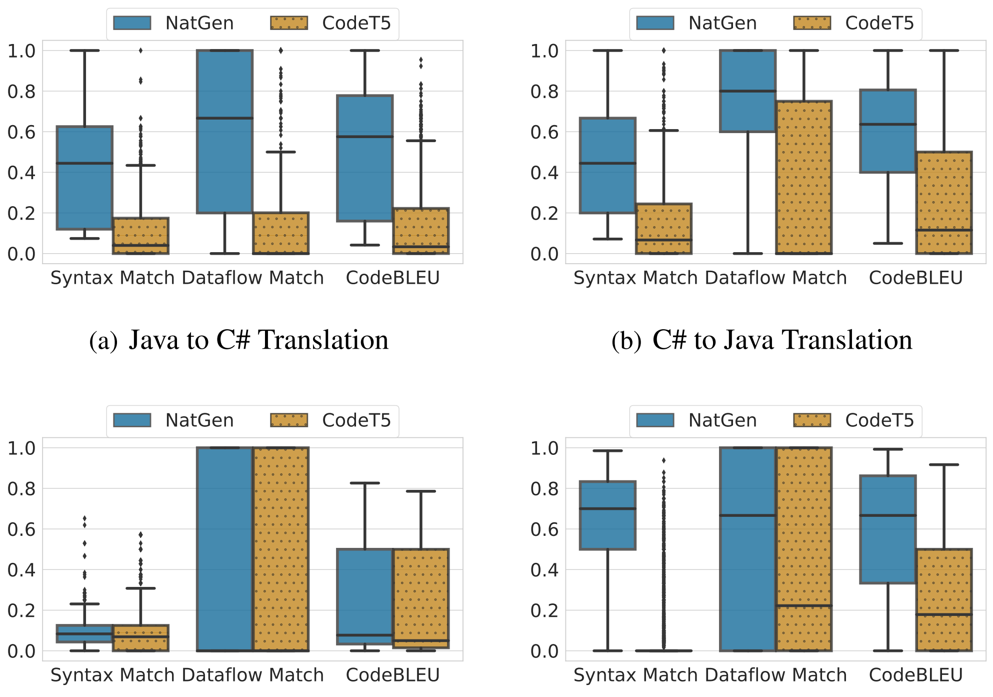
Table 7.6: Result of Bug fix (Top 1 fix accuracy).

Approach	**BugFix$_{small}$**		**BugFix$_{medium}$**	
	Unimodal	**Multimodal**	**Unimodal**	**Multimodal**
MODIT	20.35	21.57	8.35	13.18
CodeT5	21.79	22.97	12.59	**14.94**
NetGen	**22.26**	**23.43**	**13.32**	14.93

Bug Fix. Similar to MODIT, we evaluate the top-1 accuracy of the generated fixed code. We also evaluate uni-modal settings, where the fix description is unavailable, and multi-modal settings, where we have access to the fix description. Table 7.6 shows the results of Bug Fix. For the BugFix$_{small}$ dataset, NetGen outperforms both CodeT5 and MODIT in both unimodal and multi-modal settings. For For the BugFix$_{medium}$ dataset, NetGen performs better than CodeT5 and MODIT in unimodal setting and slightly worse than CodeT5 in the multi-modal setting.

Result 7.3: *NetGen performs better than most of the existing baselines. NetGen's improvement in Syntax match and Dataflow match signifies NetGen's ability to generate code syntactically and semantically closer to target code.*

RQ-7.4. How well does NetGen's pre-training help in tasks where labelled data is scarce?



(a) Java to C# Translation

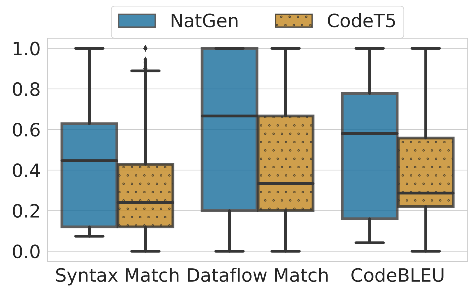
(b) C# to Java Translation

(c) Text to Code Generation

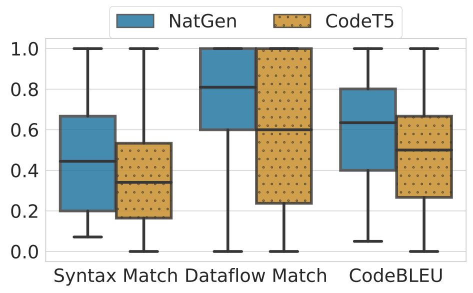
(d) Bug Fix (small, multimodal)

Figure 7.12: Zero-shot transfer learning capability of NetGen in for different tasks.

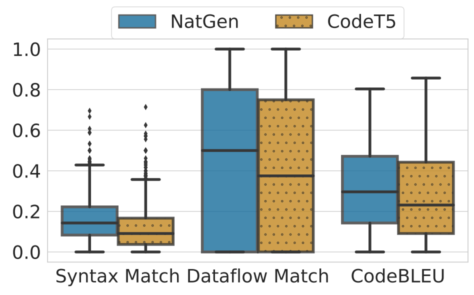
Motivation. Learning to generate code usually requires a large amount of annotated training data. A lot of time and effort goes into curating high-quality training data [6, 122]. Unsupervised pre-training endows machine learning models with necessary domain knowledge about the



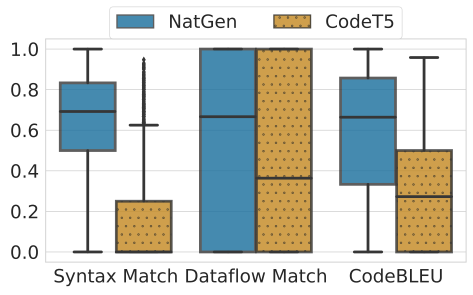
(a) Java to C# Translation



(b) C# to Java Translation



(c) Text to Code Generation

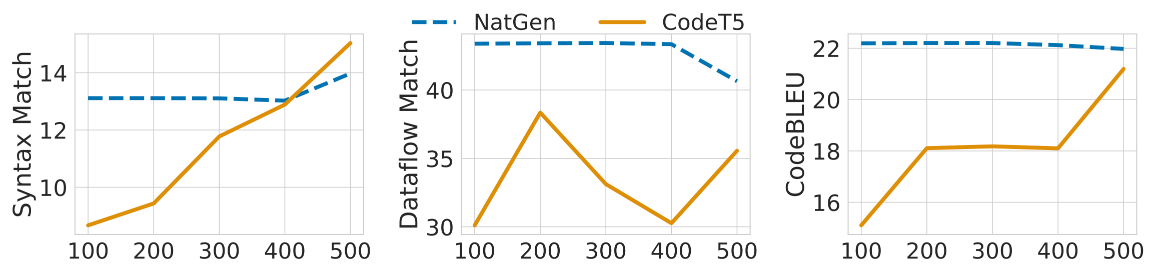


(d) Bug Fix (small, multimodal).

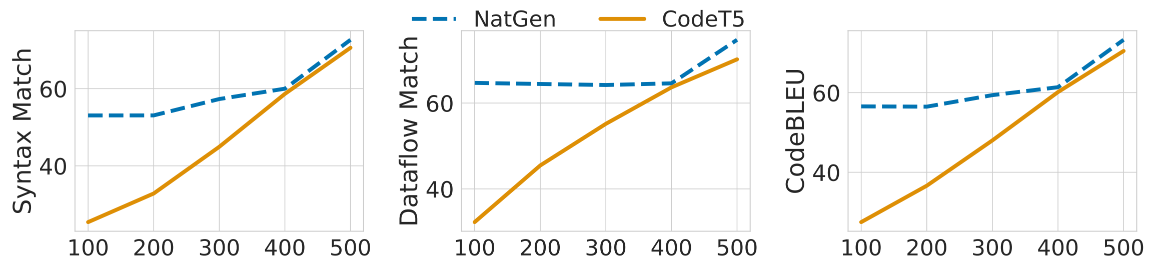
Figure 7.13: Few shot Learning evaluation of NetGen. In each case, the pre-trained model is fine-tuned on 200 training examples for 10 epoch and the result is on the full test set.

task [61]. In practice, this knowledge appears to transfer across multiple tasks. Such pre-training reduces the effort to learn each different task. We therefore study the effectiveness of NetGen's domain knowledge about source code syntax and semantics. In particular, we *stress test* whether the knowledge NetGen learned during pre-training is useful for downstream tasks, by limiting available task-specific training data.

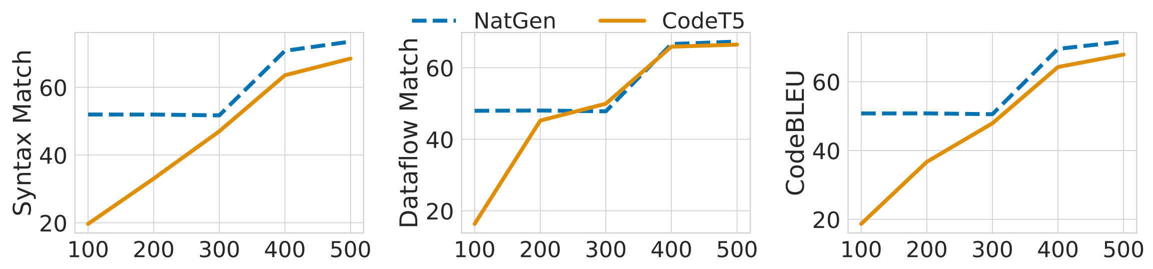
Experimental Setup. We evaluate NetGen's over several data-limited tasks: *Text to Code generation*, *Code Translation*, and *Bug Fix*. We consider two different settings. First, we consider zero-shot [202, 237] evaluation. Here we evaluate different pre-trained models *without* any task-specific training. Naturally, we don't see good performance in this setting. Nevertheless, this stress-test measures the code generation ability of models. Second, we try few-shot learning [231, 189, 214]. We randomly choose a few training examples for each task and fine-tune the pre-trained models on those examples, and evaluate their performance. We gradually increase the number of



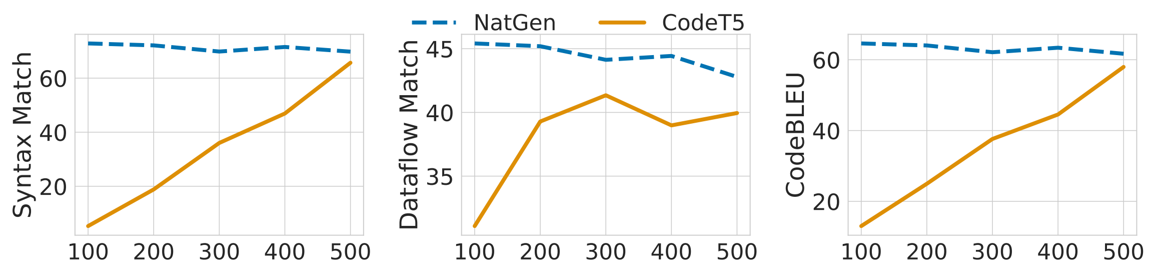
(a) Text to Code Generation.



(b) C# to Java Translation.



(c) Java to C# Translation.



(d) Bug Fix (small, multimodal).

Figure 7.14: NetGen's results on different tasks with Few shot settings. X-axis shows number of training examples.

training examples over several few-shot settings.

Results. Figure 7.12 shows the NetGen's and CodeT5's zero-shot performance. Lacking task-specific training, we can see here how much transferable knowledge each model learned just during pre-training. There are large differences in all the tasks between NetGen and CodeT5 across Syntax Match and Dataflow Match. It signifies NetGen learns to generate both syntactically and semantically correct code during pre-training, which CodeT5 rarely can do. Figure 7.13 shows the performance of NetGen and CodeT5 when trained on 200 training examples. NetGen also has an advantage over CodeT5 here.

We note a larger performance gap in the Translation tasks (Figure 7.12(a) & 7.12(b)) and Bug Fix (Figure 7.12(d)) tasks, compared to Text to Code Generation task (Figure 7.12(c)) in both the zero-shot and the few shot (Figure 7.13) experiments. We conjecture that such discrepancy is the artifact of the nature of the tasks. The cross-lingual alignment between NL and Java code is the key factor in generating text to code. In contrast, both the input and output are the programming language in the translation and bug fix task. Thus, we hypothesize that NetGen leverages its shared knowledge across different programming languages learned during the pre-training.

We further stress test NetGen's with few-shot learning; we gradually increased the number of training examples and trained both CodeT5 and NetGen. Figure 7.14 shows the performance progress as the number of training examples increase. For all four tasks, NetGen significantly improves over CodeT5 when the number of training examples is minimal. With increasing training examples, the performance gap gradually decreases. Arguably, with enough *labeled* data and enough resources, all high-capacity models will get better at generating source code. Nevertheless, we learn two critical lessons from NetGen's better performance in zero-shot and few-shot learning. First, NetGen's better performance across all tasks suggests that that the coding knowledge it learns from the naturalization task is more generic and transferable. Second, for any pre-trained model to be effective in code generation, especially in a limited training data scenario, the pre-training should explicitly teach the model how to write code. Otherwise, we hypothesize that a big chunk of fine-tuning resources will be spent on the models' learning to write code.

Result 7.4: *NetGen is very effective in source code generative tasks when minimal training resource is available. Since NetGen explicitly learns to generate code during pre-training, it can avoid learning such during fine-tuning saving fine-tuning resource.*

7.5.4 Limitations & Threats

Bias introduced by 'de-naturalizing' transformations. In Section 7.3.1, we described our six transformations to "de-naturalize" source code. The NetGen model learns to revert one transformation at a time. In fact, we found empirically that, when given code with more than one 'de-naturalization' transformation applied, the model reverses only one of them. There is thus a threat our limited application of de-naturalization limits the ability of our NetGen. Regardless, we consider NetGen as a proof-of-concept and the first work towards teaching a model to write natural code.

Knowledge retention from CodeT5. As mentioned in Section 7.4, we start NetGen's pre-training from CodeT5-base model [232]. Starting further pre-training from an existing pre-trained checkpoint is very common in large-scale pre-training. For instance, GraphCodeBERT [78] is pre-trained based on CodeBERT [63] model, which was pre-trained based on RoBERTa [138] model. Both the Open AI-CodeX [41] and Github Copilot [72] models are further pre-trained in OpenAI-GPT3 [32]. Nevertheless, when we further train a pre-trained model on different tasks, it is subject to "*catastrophic forgetting*" [115] of the knowledge learned in the base model. In order to test whether NetGen is forgetting CodeT5's knowledge about natural language generation, we also evaluate NetGen for Code summarization. Here the input is source code, and the output is Natural language. After fine-tuning NetGen's overall BLEU in 19.547 while CodeT5's was 19.551, suggesting that NetGen mostly retains CodeT5's capacity to generate NL.

Fair Comparison with CodeT5. We initialize NetGen with pre-trained checkpoint from CodeT5 (already pre-trained 75K steps with their objective) and train NetGen for 25K steps with 'natural-code' writing objective. A skeptic reader would want to know what happens when we pre-train CodeT5 for 25K more steps with their training objective. We argue that since the pre-training objective does not explicitly account for generating code (See section 3.2 of CodeT5's original paper), further training with the CodeT5 objective does not necessarily increase its code generation capacity. We do acknowledge CodeT5's ability to understand and reason about *input*. Since the pre-training large model is extremely expensive (§7.4)[7]; we leverage such knowledge by initializing NetGen from CodeT5's publicly available pre-trained model. Moreover, CodeT5 release neither their code for pre-training (only for fine-tuning), nor any earlier or later checkpoints for us to carry out further investigation.

"Naturalization" with program-analysis. NetGen is a prototype of a generative pre-trained model with "Naturalization" task, trained to revert six classes of de-naturalization transformations (see Figure 7.2). However, perfect performance *w.r.t.* these transformation is **not** the main objective of this research. Tools to accomplish "naturalization" could surely be built using traditional refactoring methods; however, our goal is to train NetGen so that it learns to generate natural code with the help of this "Naturalization" task.

NetGen as "Code-Refactoring" tool. NetGen suggests the promise of neural transformers to build meaning-preserving code-refactoring tools. However, to realize a more accurate and powerful neural re-factoring tool, more training data, with a larger variety of transformations, would be required.

7.6 Concluding Remarks

In this chapter, we introduce the "Code-Naturalization" pre-training objective for generative models of code. As proof-of-concept we pre-trained our NetGen to write 'natural' source code from 'un-natural' counterpart. With this pre-training, NetGen learns to write code syntactically

[7]CodeT5 was pre-trained on 16 NVIDIA A100s, with 40G memory each, for 12 days! One might reasonably assume it was already well-trained on the original objective

and semantically closer to developers' written code. We "de-naturalize" existing developers' code, using six kinds of "semantic-preserving" transformations. We further fine-tune the NetGen on different variations of three downstream tasks that require code generation. NetGen achieves state-of-the-art performance in these downstream tasks, and NetGen's generated code are syntactically and semantically closer to the target code. Our pre-training on the 'naturalizing' task is especially effective in resource-constrained setting *i.e.* zero-shot, and few-shot transfer learning.

Chapter 8: Conclusion

8.1 Dissertation Summary

This dissertation approaches solving automated code editing problem. We formulate the problem of editing an existing code with guidance from developers. We aim to develop tools to generate edited code, given the code before the edit and other guidance from the developer. We deconstruct the problem as a combination of source code understanding and source code generation problem and approach to solving the problem with encoder-decoder-based Deep Learning models. The main challenge in designing such a system is designing effective ways to embed the knowledge about Programming Language into the model. As a systematic solution strategy to this challenge, we propose two different directions for building a model.

Our first solution approach is explicitly encoding PL properties into the model (Part I). We can build models based on the PL properties, which are very strictly defined (*e.g.,* syntax). As an example of such a model, we built CODIT (Chapter 3), a tree-based neural machine translation model. We built CODIT based on the Context-Free Grammar of Programming Language. In contrast to generating edited source code directly, CODIT first generates the tree structure of the code, guaranteeing syntactic correctness. CODIT has shown promise in learning frequent code edit patterns followed by developers, which has the potential for automated program repair. We also performed empirical study of different explicitly encoded model for source code understanding (Chapter 4).

In another orthogonal approach, we implicitly encode these PL properties into the model (Part II). The idea of implicit encoding hinges on the fact that PL properties such as syntax, semantics, and coding convention are independent of the task the model is aiming at. Researchers can re-use a model endowed with such PL knowledge in many SE tasks. With these motivations, we pre-trained a large-scale model for programming languages, PLBART(Chapter 5) on billions of source code collected from GitHub. PLBART is the first pre-trained encoder-decoder model for Programming Language, where we showed that with large scale-pre-training, models learn to generate syntactically and semantically correct code. PLBART shows great promise in many different SE tasks. In a subsequent empirical study (MODIT), we have shown the effectiveness of PLBART in automated code editing (Chapter 6).

We pre-trained another large-scale model, NatGen(Chapter 7), to encode the naturalness of the source code into the model. We designed a new pre-training objective, "Naturalizing" of source code, exploiting the code's bimodal, dual-channel (formal & natural channels) nature. Unlike natural language, code's bimodal, dual-channel nature allows us to generate semantically equivalent code at scale. We introduce six classes of semantic preserving transformations to introduce unnatural forms of code and then force our model to produce more natural original programs written by developers. We show that NatGen is incredibly competitive at zero-shot and few-shot learning and better at learning code properties (e.g., syntax, data flow).

Finally, this dissertation lays a foundational contribution to solving Automated Code Editing, which I hope will guide future research on this problem. While solving this problem, we designed and trained three different models – CODIT, PLBART, NatGen, and two distinct training objectives/procedures – Tree Translation (Section 3.2.1) and Naturalization (Section 7.3). I hope these designs will inspire the future design of models for other SE tasks. In addition to solving the Automated Code Editing problem, we extensively evaluated other problems in source code understanding and source code generation. I hope the techniques we invented in this dissertation and the empirical lessons we learned will help advance the frontier of Artificial Intelligence for Software Engineering (AI4SE). I also hope this dissertation will lay the foundation of *Programming Language Processing (PLP)* as a new research field.

8.2 Future Work

Program Generation/Synthesis. Automated program generation/synthesis can help developers to a great extent to increase productivity. The ever-changing landscape of APIs and libraries in modern API-driven software development demands a steep learning curve for programmers. To each such burden, an automated program synthesis/generation tool may help programmers. Such an automated tool can generate programs from a set of I/O examples, a description of the task, etc. The current SOTA for machine generation of programs is far from perfect. On the one hand, there is precise program synthesis with provable correctness. However, these programs are often tailored towards the problem domain and do not scale well for general-purpose programs. On the other hand, most recently, large-scale models such as CodeX [41], GitHub Copilot [72], Amazon CodeWhisperer [22] have been tremendously helping developers with code suggestions, interactive pair-programming, etc., However, such general-purpose source code generation often generates syntactically and semantically incorrect codes, hindering their integration in automated programming. In particular, program synthesizer should be built by leveraging both ends of the spectrum. While there are some initial works [98] in this research direction, further investigation is needed to build effective program synthesizer for general purpose program with syntactic and semantic correctness guarantee.

Learning Source Code Functionality Most recent works in developing large-scale models for source code are aimed at learning syntax and semantics of source code. However, human perception of a piece of code depends on the functionality or execution behavior of the code. Thus, for a model to achieve near human level understanding about code, it is of utmost importance to reason about the functionality. Ding *et al.* have performed some initial work on statically reasoning about functionality [57]. Pei *et al.* [180, 179, 178] had done some initial work on dynamic reasoning about functionality through execution trace modeling. A joint modeling of static and dynamic functionality reasoning would pave the way of proper understanding of source code.

Ensuring security and trustworthiness of software systems. Digitization and dependence on automation make the human race susceptible to security/privacy and trust violation. The distributed, independent, and layered architecture of modern-day systems may raise vulnerable emergent behavior across the system as a whole, even when each component in the system is independently tested for their respective service level agreements(SLA). For instance, consider an IoT system automated home; the sensors, actuators, routers, servers are all independent layers of a system. However, when such systems use shared resources (e.g., memory, network access), they may create security/privacy attack backdoors due to the inconsistencies between layers. The multi-vendor nature of different layers leaves room for confusion and makes it difficult to reach a consensus on the SLA. Such vulnerable emergent behaviors are often not anticipated layers assembler and often identified in post-deployment. A challenging open research problem is to precisely localize the bug reasoning about the execution trace and program flow graphs. The inter-layer communication and associated uncertainty (if any) makes it specially difficult to reason about any fault, much so for a Machine Learning based system. Solving this problem requires in depth analysis an modeling of the volatility along with models for all the components.

References

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

[2] A. Agrawal and T. Menzies, "Is "better data" better than "better data miners"?" *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1050–1061, 2018.

[3] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online: Association for Computational Linguistics, Jul. 2020, pp. 4998–5007.

[4] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

[5] ——, "Unified pre-training for program understanding and generation," *2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2021.

[6] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, *Avatar: A parallel corpus for java-python program translation*, 2021. arXiv: `2108.11590 [cs.SE]`.

[7] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 143–153, 2019.

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

[9] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 38–49, 2015.

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

[11] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, “Typilus: Neural type hints,” *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pp. 91–105, 2020.

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

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

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

[15] U. Alon, S. Brody, O. Levy, and E. Yahav, “Code2seq: Generating sequences from structured representations of code,” *arXiv preprint arXiv:1808.01400*, 2018.

[16] ——, “Code2seq: Generating sequences from structured representations of code,” *International Conference on Learning Representations*, 2019.

[17] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[18] M. Amodio, S. Chaudhuri, and T. W. Reps, “Neural attribute machines for program generation,” *arXiv preprint arXiv:1705.09231*, 2017.

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

[20] A. Arcuri and L. Briand, “A practical guide for using statistical tests to assess randomized algorithms in software engineering,” *Proceedings of the 33rd International Conference on Software Engineering*, ICSE ’11, 1–10, 2011.

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

[22] A. AWS, *Code whisperer (`https://aws.amazon.com/codewhisperer/`)*. 2022.

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

[24] M. Barenkamp, J. Rebstadt, and O. Thomas, "Applications of ai in classical software engineering," *AI Perspectives*, vol. 2, no. 1, pp. 1–15, 2020.

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

[26] S. Barua, M. M. Islam, and K. Murase, "Prowsyn: Proximity weighted synthetic oversampling technique for imbalanced data set learning," *Advances in Knowledge Discovery and Data Mining*, J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, Eds., pp. 317–328, 2013.

[27] S. Barua, M. M. Islam, X. Yao, and K. Murase, "Mwmote–majority weighted minority oversampling technique for imbalanced data set learning," *IEEE Transactions on knowledge and data engineering*, vol. 26, no. 2, pp. 405–425, 2012.

[28] H. D. Beale, H. B. Demuth, and M. Hagan, "Neural network design," *Pws, Boston*, 1996.

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

[30] J. S. Bridle, "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in Springer, 1990, pp. 227–236.

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

[32] T. B. Brown *et al.*, *Language models are few-shot learners*, 2020. arXiv: `2005.14165 [cs.CL]`.

[33] C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, "A theory of dual channel constraints," *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 25–28, 2020.

[34] C. Casalnuovo, K. Lee, H. Wang, P. Devanbu, and E. Morgan, "Do programmers prefer predictable expressions in code?" *Cognitive science*, vol. 44, no. 12, e12921, 2020.

[35] C. Casalnuovo, E Morgan, and P Devanbu, "Does surprisal predict code comprehension difficulty," *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*, 2020.

[36] S. Chakraborty, T. Ahmed, Y. Ding, P. Devanbu, and B. Ray, "Natgen: Generative pre-training by" naturalizing" source code," *arXiv preprint arXiv:2206.07585*, 2022.

[37] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, 2020.

[38] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[39] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 443–455, 2021.

[40] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[41] M. Chen *et al.*, *Evaluating large language models trained on code*, 2021. arXiv: `2107.03374 [cs.LG]`.

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

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

[44] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

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

[46] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," *International Conference on Learning Representations*, 2020.

[47] C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "PyMT5: Multi-mode translation of natural language and python code with transformers," *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 9052–9065, Nov. 2020.

[48] A. Conneau and G. Lample, "Cross-lingual language model pretraining," in H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 7059–7069.

[49] A. Conneau *et al.*, "Unsupervised cross-lingual representation learning at scale," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online: Association for Computational Linguistics, Jul. 2020, pp. 8440–8451.

[50] *Cwe-761 (`https://cwe.mitre.org/data/definitions/761.html`)*, 2020.

[51] H. K. Dam, "Artificial intelligence for software engineering," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 25, no. 3, pp. 34–37, 2019.

[52] C. Developers, *Cppcheck: A tool for static c/c++ code analysis*, 2019.

[53] L. Developers, *Clang (`clang.llvm.org`)*, 2019.

[54] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Jun. 2019.

[55] ——, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[56] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," *International Conference on Learning Representations*, 2019.

[57] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, *Contrastive learning for source code with structural and functional properties*, 2021. arXiv: `2110.03868 [cs.PL]`.

[58] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as translation: The data and the metaphor," *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 275–286, 2020.

[59] L. Dong *et al.*, "Unified language model pre-training for natural language understanding and generation," *arXiv preprint arXiv:1905.03197*, 2019.

[60] M. Drissi *et al.*, "Program language translation using a grammar-driven tree-to-tree model," *ICML Workshop on Neural Abstract Machines & Program Induction (NAMPI v2)*, 2018.

[61] D. Erhan, A. Courville, Y. Bengio, and P. Vincent, "Why does unsupervised pre-training help deep learning?" *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 201–208, 2010.

[62] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 313–324, 2014.

[63] Z. Feng *et al.*, “CodeBERT: A pre-trained model for programming and natural languages,” *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Nov. 2020.

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

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

[66] E. Foundation, *Eclipse IDE (`https://www.eclipse.org`)*, 2018.

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

[68] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, “Noise and heterogeneity in historical build data: An empirical study of travis ci,” *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 87–97, 2018.

[69] X. Ge, Q. L. DuBose, and E. Murphy-Hill, “Reconciling manual and automatic refactoring,” *Proceedings of the 34th International Conference on Software Engineering*, pp. 211–221, 2012.

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

[71] F. Girosi, M. Jones, and T. Poggio, “Regularization theory and neural networks architectures,” *Neural computation*, vol. 7, no. 2, pp. 219–269, 1995.

[72] GitHub, *Github copilot (`https://copilot.github.com/`)*. 2022.

[73] D. Gopstein, A.-L. Fayard, S. Apel, and J. Cappos, “Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion,” *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 605–616, 2020.

[74] D. Gopstein, H. H. Zhou, P. Frankl, and J. Cappos, “Prevalence of confusing code in software projects: Atoms of confusion in the wild,” *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 281–291, 2018.

[75] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, “Code to comment “translation”: Data, metrics, baselining & evaluation,” *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 746–757, 2020.

[76] C. A. Gunter, *Semantics of programming languages: structures and techniques*. MIT press, 1992.

[77] D. Guo, D. Tang, N. Duan, M. Zhou, and J. Yin, "Coupling retrieval and meta-learning for context-dependent semantic parsing," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 855–866.

[78] D. Guo *et al.*, "Graphcodebert: Pre-training code representations with data flow," *International Conference on Learning Representations*, 2021.

[79] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 364–379, 2018.

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

[81] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," *arXiv preprint arXiv:2205.12424*, 2022.

[82] J. Harer, C. Reale, and P. Chin, "Tree-transformer: A transformer-based method for correction of tree-structured data," *arXiv preprint arXiv:1908.00449*, 2019.

[83] M. Harman, "The role of artificial intelligence in software engineering," *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pp. 1–6, 2012.

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

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

[86] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum, "Developing a natural language interface to complex data," *ACM Transactions on Database Systems (TODS)*, vol. 3, no. 2, pp. 105–147, 1978.

[87] M. R. Hess and J. D. Kromrey, "Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances," *annual meeting of the American Educational Research Association*, pp. 1–30, 2004.

[88] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

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

[90] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," *International Workshop on Similarity-Based Pattern Recognition*, pp. 84–92, 2015.

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

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

[93] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[94] S. Iyer, A. Cheung, and L. Zettlemoyer, "Learning programmatic idioms for scalable semantic parsing," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5426–5435.

[95] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 1643–1652.

[96] ——, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[97] ——, "Summarizing source code using a neural attention model," *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083, 2016.

[98] N. Jain *et al.*, "Jigsaw: Large language models meet program synthesis," *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 1219–1231, 2022.

[99] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "Scottknott: A package for performing the scott-knott clustering algorithm in r," *TEMA (São Carlos)*, vol. 15, no. 1, pp. 3–17, 2014.

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

[101] N. Jiang, T. Lutellier, and L. Tan, “Cure: Code-aware neural machine translation for automatic program repair,” *arXiv preprint arXiv:2103.00073*, 2021.

[102] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” *Proceedings of the 2013 International Conference on Software Engineering*, pp. 672–681, 2013.

[103] R. W. Johnson, “An introduction to the bootstrap,” *Teaching Statistics*, vol. 23, no. 2, pp. 49–54, 2001.

[104] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for java programs,” *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

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

[106] A. Kamath and R. Das, “A survey on semantic parsing,” *arXiv preprint arXiv:1812.00978*, 2018.

[107] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Pre-trained contextual embedding of source code,” *arXiv preprint arXiv:2001.00059*, 2019.

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

[109] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, “Big code!= big vocabulary: Open-vocabulary models for source code,” *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1073–1085, 2020.

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

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

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

[113] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," *arXiv preprint arXiv:2003.13848*, 2020.

[114] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 481–490, 2011.

[115] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[116] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-Source Toolkit for Neural Machine Translation," *ArXiv e-prints*, eprint: `1701.02810`.

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

[118] P. Koehn, "Statistical significance tests for machine translation evaluation," *Proceedings of the 2004 conference on empirical methods in natural language processing*, pp. 388–395, 2004.

[119] P. Koehn *et al.*, "Moses: Open source toolkit for statistical machine translation," in *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, Prague, Czech Republic: Association for Computational Linguistics, Jun. 2007, pp. 177–180.

[120] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 66–71.

[121] P. Kumar and S. Bedathur, "A survey on semantic parsing from the perspective of compositionality," *arXiv preprint arXiv:2009.14116*, 2020.

[122] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 601–20 611, 2020.

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

[124] 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," *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 3–13, 2012.

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

[126] M. Lewis *et al.*, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” *arXiv preprint arXiv:1910.13461*, 2019.

[127] M. Lewis *et al.*, “BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online: Association for Computational Linguistics, Jul. 2020, pp. 7871–7880.

[128] J. Li, Y. Wang, M. R. Lyu, and I. King, “Code completion with neural attention and pointer networks,” *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 4159–4165, Jul. 2018.

[129] L. H. Li, M. Yatskar, D. Yin, C.-J. Hsieh, and K.-W. Chang, “Visualbert: A simple and performant baseline for vision and language,” *arXiv preprint arXiv:1908.03557*, 2019.

[130] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” *arXiv preprint arXiv:1511.05493*, 2015.

[131] Z. Li *et al.*, “Sysevr: A framework for using deep learning to detect software vulnerabilities,” *arXiv preprint arXiv:1807.06756*, 2018.

[132] Z. Li *et al.*, “Vuldeepecker: A deep learning-based system for vulnerability detection,” *arXiv preprint arXiv:1801.01681*, 2018.

[133] Z. Li, L. Qu, and G. Haffari, “Context dependent semantic parsing: A survey,” *arXiv preprint arXiv:2011.00797*, 2020.

[134] C.-Y. Lin and F. J. Och, “ORANGE: A method for evaluating automatic evaluation metrics for machine translation,” in *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, Geneva, Switzerland: COLING, 2004, pp. 501–507.

[135] P. Liu, O. Tripp, and X. Zhang, “Flint: Fixing linearizability violations,” *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 543–560, 2014.

[136] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, “Retrieval-augmented generation for code summarization via hybrid gnn,” *arXiv preprint arXiv:2006.05405*, 2020.

[137] Y. Liu *et al.*, “Multilingual denoising pre-training for neural machine translation,” *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 726–742, 2020.

[138] Y. Liu *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[139] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 373–384, 2018.

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

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

[142] S. Lu *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[143] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 1412–1421.

[144] L. Lusa *et al.*, "Smote for high-dimensional class-imbalanced data," *BMC bioinformatics*, vol. 14, no. 1, p. 106, 2013.

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

[146] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 101–114, 2020.

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

[148] C. Mao, Z. Zhong, J. Yang, C. Vondrick, and B. Ray, "Metric learning for adversarial robustness," *Advances in Neural Information Processing Systems*, pp. 478–489, 2019.

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

[150] A. Mastropaolo *et al.*, "Studying the usage of text-to-text transfer transformer to support code-related tasks," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 336–347, 2021.

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

[152] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 392–402, 2015.

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

[154] ——, "Systematic editing: Generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.

[155] Microsoft, *CodeBLEU calculator (`https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans/evaluator/CodeBLEU`).* 2021.

[156] ——, *Codexglue leaderboard (`https://microsoft.github.io/CodeXGLUE/`).* 2022.

[157] ——, *Visual Studio (`https://visualstudio.microsoft.com`)*, 2018.

[158] J. C. Mitchell, *Foundations for programming languages*. MIT press Cambridge, 1996, vol. 1.

[159] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "Tbcnn: A tree-based convolutional neural network for programming language processing," *arXiv preprint arXiv:1409.5718*, 2014.

[160] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," *Thirtieth AAAI conference on artificial intelligence*, 2016.

[161] D. Movshovitz-Attias and W. W. Cohen, "Natural language models for predicting programming comments," *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 35–40, Aug. 2013.

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

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

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

[165] A. T. Nguyen *et al.*, "API code recommendation using statistical learning from fine-grained changes," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 511–522, 2016.

[166] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 180–190, 2013.

[167] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

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

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

[170] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: Sequence-to-sequence pre-training for learning the representation of source code," *arXiv preprint arXiv:2201.01549*, 2022.

[171] M. Ott *et al.*, "Fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 48–53.

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

[173] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[174] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318.

[175] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *arXiv preprint arXiv:2108.11601*, 2021.

[176] M. R. Parvez, S. Chakraborty, B. Ray, and K.-W. Chang, "Building language models for text with named entities," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 2373–2383.

[177] J. Patra and M. Pradel, "Semantic bug seeding: A learning-based approach for creating realistic bugs," *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 906–918, 2021.

[178] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning," *arXiv preprint arXiv:2010.00770*, 2020.

[179] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[180] K. Pei *et al.*, "Stateformer: Fine-grained type recovery from binaries using generative state modeling," *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 690–702, 2021.

[181] L. Phan *et al.*, "Cotext: Multi-task learning with code-text transformer," *arXiv preprint arXiv:2105.08645*, 2021.

[182] K. Plunkett and J. L. Elman, *Exercises in rethinking innateness: A handbook for connectionist simulations*. Mit Press, 1997.

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

[184] D. M. Powers, "Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.

[185] M. Pradel and S. Chandra, "Neural software analysis," *Communications of the ACM*, vol. 65, no. 1, pp. 86–96, 2021.

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

[187] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[188] C. Raffel *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv preprint arXiv:1910.10683*, 2019.

[189] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning," 2016.

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

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

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

[193] ——, "The uniqueness of changes: Characteristics and applications," *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 34–44, 2015.

[194] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev, "Refactoring with synthesis," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 339–354, 2013.

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

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

[197] S. Ren *et al.*, "Codebleu: A method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[198] D. M. Ritchie, S. C. Johnson, M. Lesk, B. Kernighan, *et al.*, "The c programming language," *Bell Sys. Tech. J*, vol. 57, no. 6, pp. 1991–2019, 1978.

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

[200] R. Rolim *et al.*, “Learning syntactic program transformations from examples,” *Proceedings of the 39th International Conference on Software Engineering*, pp. 404–415, 2017.

[201] A. Romanovsky, “Exception handling in component-based system development,” *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, pp. 580–586, 2001.

[202] B. Romera-Paredes and P. Torr, “An embarrassingly simple approach to zero-shot learning,” *International conference on machine learning*, pp. 2152–2161, 2015.

[203] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, “Dobf: A deobfuscation pre-training objective for programming languages,” *arXiv preprint arXiv:2102.07492*, 2021.

[204] R. Russell *et al.*, “Automated vulnerability detection in source code using deep representation learning,” *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 757–762, 2018.

[205] B. G. Ryder and M. L. Soffa, “Influences on the design of exception handling acm sigsoft project on the impact of software engineering research on programming language design,” *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 4, pp. 29–35, 2003.

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

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

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

[209] F. Schroff, D. Kalenichenko, and J. Philbin, “Facenet: A unified embedding for face recognition and clustering,” *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015.

[210] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[211] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, “Questions developers ask while diagnosing potential security vulnerabilities with static analysis,” *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 248–259, 2015.

[212] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[213] C. Sun, A. Myers, C. Vondrick, K. Murphy, and C. Schmid, "Videobert: A joint model for video and language representation learning," *Proceedings of the IEEE International Conference on Computer Vision*, pp. 7464–7473, 2019.

[214] Q. Sun, Y. Liu, T.-S. Chua, and B. Schiele, "Meta-transfer learning for few-shot learning," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 403–412, 2019.

[215] B. W. SUTER, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *IEEE Transactions on Neural Networks*, vol. 1, no. 4, p. 291, 1990.

[216] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

[217] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 99–108, 2015.

[218] W. Tansey and E. Tilevich, "Annotation refactoring: Inferring upgrade transformations for legacy applications," *ACM Sigplan Notices*, vol. 43, no. 10, pp. 295–312, 2008.

[219] C. K. Tantithamthavorn and J. Jiarpakdee, "Explainable ai for software engineering," *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1–2, 2021.

[220] D. Tarlow *et al.*, "Learning to fix build errors with graph2diff neural networks," *arXiv preprint arXiv:1911.01205*, 2019.

[221] L. Torrey and J. Shavlik, "Transfer learning," in IGI global, 2010, pp. 242–264.

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

[223] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," *arXiv preprint arXiv:1901.09102*, 2019.

[224] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine transla-

tion," *NIER Paper, Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[225] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[226] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," *arXiv preprint arXiv:2101.02518*, 2021.

[227] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 683–693, 2017.

[228] A. Vaswani *et al.*, "Attention is all you need," *Advances in Neural Information Processing Systems 30*, I. Guyon *et al.*, Eds., pp. 5998–6008, 2017.

[229] J. Wang, F. Zhou, S. Wen, X. Liu, and Y. Lin, "Deep metric learning with angular loss," *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2593–2601, 2017.

[230] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 261–271, 2020.

[231] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM computing surveys (csur)*, vol. 53, no. 3, pp. 1–34, 2020.

[232] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[233] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in Neural Information Processing Systems*, pp. 6563–6573, 2019.

[234] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, no. 1, pp. 1–40, 2016.

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

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

[237] Y. Xian, C. H. Lampert, B. Schiele, and Z. Akata, “Zero-shot learning—a comprehensive evaluation of the good, the bad and the ugly,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 9, pp. 2251–2265, 2018.

[238] T. Xie, “Intelligent software engineering: Synergy between ai and software engineering,” *International symposium on dependable software engineering: Theories, tools, and applications*, pp. 3–7, 2018.

[239] C. Xu, S. Ma, and D. Lo, “Plp 2021: Workshop on programming language processing,” *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pp. 4171–4172, 2021.

[240] H. Xu, S. Fan, Y. Wang, Z. Huang, H. Xu, and P. Xie, “Tree2tree structural language modeling for compiler fuzzing,” *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 563–578, 2020.

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

[242] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.

[243] P. Yin and G. Neubig, “A syntactic neural model for general-purpose code generation,” *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 440–450, 2017.

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

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

[246] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, “Neural detection of semantic code clones via tree-based convolution,” *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 70–80, 2019.

[247] E. Yudkowsky, “Artificial intelligence as a positive and negative factor in global risk,” *Global catastrophic risks*, vol. 1, no. 303, p. 184, 2008.

[248] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” *Proceedings of the 41st International Conference on Software Engineering*, ICSE ’19, 783–794, 2019.

[249] R. Zhao, D. Wang, R. Yan, K. Mao, F. Shen, and J. Wang, “Machine health monitoring using local feature-based gated recurrent unit networks,” *IEEE Transactions on Industrial Electronics*, vol. 65, no. 2, pp. 1539–1548, 2017.

[250] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” *Advances in Neural Information Processing Systems*, vol. 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., pp. 10 197–10 207, 2019.

[251] Y. Zhou and A. Sharma, “Automated identification of security issues from commit messages and bug reports,” *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pp. 914–919, 2017.

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

[253] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, “μVuldeepecker: A deep learning-based system for multiclass vulnerability detection,” *IEEE Transactions on Dependable and Secure Computing*, 2019.