LEARNING TO EDIT SOURCE CODE

Saikat Chakraborty

April 25, 2021

1 Background

Self-correcting software or self-modifying software is one of the key stepping stones for absolute automation in software. A software that can – a) identify and repair any potential flaw, or b) change part of its source code following the code-change pattern that the developers. Previous research efforts showed that changes made by real developers are repetitive [5]. Such repetitive nature of changes enables us to build an intelligent machine that can learn those code change patterns and apply them wherever necessary. *Automatic Code Change* task can be formulated as a translation task, where the intelligent machine will translate a piece of code to a changed code. Consequently, Machine Translator tools such as Neural Machine Translators (NMT), are a natural fit for this task. Previous research efforts [3, 6] 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 code; the decoder generates the changed or corrected code. While existing approaches use NMT techniques from natural language, code is fundamentally different from natural language. More precisely, unlike natural language, code exhibits defined syntactic and semantic structure, and the possible name of identifiers can be virtually infinite. Thus further investigation is needed to build a usable tool for learning automatic code change.

2 Research Overview

In my research, I offer a thorough investigation into the usage of NMT in the Automated Code Change task. Some key insights guide my investigations and subsequent tool development efforts –

1. The tool **must** understand what the input code is. The encoder of the NMT **must not** solely rely on the code lexicons. It should be able to reason (albeit implicitly) about the input code's syntax and semantics.
2. Generated code by the decoder (generator) **must** generate *syntactically correct* code. Any generated code that is not syntactically correct is a completely unusable.
3. The decoder (generator) **must** make sure that the generated code is *contextually correct*. The code **must** – a) use appropriate names for appropriate identifiers, b) maintain proper usage of APIs, and c) exhibit developers' and/or organizational coding practices and rules. Fulfilling this requirement will embellish an automated code editor in real development environment.

In light of these insights, we first developed CODIT [2][1]. I investigated the syntactic correctness guarantee of generated code through Context-Free Grammar (CFG) of the programming language. 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 as well as automated program repair.

While CODIT successfully generates syntactically correct code, it does not provide any provision to learn about the contextual correctness. Nor CODIT is designed to learn developers' coding patterns and practices. For instance, suppose in an organization, developers tend to catch an exception locally, while other developers may throw that to the caller. While one of these are more correct than the other depending on the context, from CODIT 's perspective, both are syntactically correct. We cannot deterministically say which one CODIT will prefer to generate over the other.

[1] Published in TSE'20

We solve the contextual correctness problem with a key insight into how any machine learning model works. In any ML model, conceptually, there are two parts – first, understand the input code's contextual semantics, and second, reasoning about the task at hand. Similarly, in the case of NMT based code change tool should learn – a) understand the code contextual semantics and learn to generate syntactically and contextually correct code, and b) learn to apply the change patterns. Since the first task does not really depend on the code changes, we can pre-train an NMT model with raw code samples. Such training aims to make the encoder understand the code semantics and generate syntactically and contextually correct code. With all these insights, we developed PLBART [1][2] – A pre-training mechanism for simultaneously understanding and generating code. We leveraged denoising auto-encoding [4] to pre-train the encoder and decoder. PLBART showed a great promise in a wide variety of downstream software engineering tasks, including Automated Code Change.

We learned a key lesson from the experiences with CODIT and PLBART . While high-level code change patterns are repetitive, individual code changes are usually much more diverse. For instance, *safeguarding unsafe APIs with exception handling* is a common high-level pattern that developers follow while changing code. However, in reality, the developer may handle it locally or through to the caller. Among these two alternatives, which one developer would choose depends on the development circumstances. For a developer, such a piece of information about the circumstances can be bug/issue reports, test cases, code review comments, etc. Depending on this auxiliary direction, a developer may choose to prefer one change over the other. With this insight, I have been designing a model to incorporate such an auxiliary source of information. My initial finding in this work shows that we can incorporate additional information, and such information boosts the Automated Code Change tool's performance.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2021.

[2] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[3] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering(TSE)*, 2019.

[4] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[5] Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. The uniqueness of changes: Characteristics and applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 34–44. IEEE, 2015.

[6] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

[2] I am jointly primary contributor in this paper.