(Joseph) Cole Ramos
07300
11 December 2021

Project Milestone Report

Project Mentor: Professor Vincent Hellendoorn - CMU Institute for Software Research

Project Webpage: https://github.com/Joseph-Ramos-CMU/annotated-code-ml

Progress Report

Most of the work so far has been reading up on papers recommended by my mentor in order to gain a better understanding of the field, and in an attempt to find the base model and static analyzer to use. This reading included the following three papers:

[1] Evaluating Large Language Models Trained on Code

This paper describes OpenAI's Codex model, which powers GitHub's Copilot and is a popular model in the field. I was hoping to use this model as the base for my own, then adjust its parameters based on the new methodology I am proposing. Codex did something similar, starting with the GPT-3 natural language model and tuning its parameters for source code generation. However, Codex's parameter weights are not publicly available. The API is available in a closed beta, so other papers like [3] have been to compare their model's performance against it, but this means I cannot use it as a base model.

Despite this, the paper did offer a fair amount of useful information on the process of collecting and preprocessing training data, as well as how to evaluate the model. This paper used "functional correctness" (number of unit tests passed) as the performance metric, as opposed to some type of similarity score to a reference solution (the BLEU metric). The paper puts forward good reasoning for using "functional correctness" as a metric, though this will potentially be a logistical issue for my project. Evaluating functional correctness requires compiling and running the outputs from the model in a sandbox/ containerized environment, which introduces some configuration issues and makes running the experiments harder than simply computing the similarity between two uncompiled versions of source code.

The paper does make its evaluation dataset publicly available if I wish to compare my model's effectiveness against it.

[2] Building language models for text with named entities

This paper proposes a model that uses a two-level system to help improve performance of text generation when dealing with vocabularies that have infrequent words. Though the context was not specific to code generation, the methodology was applied to both getting item names in the context of cooking recipes and variables names in Java source code. The first level would predict the type of the next element, if relevant. In the context of generating Java code, these types are the normal Java types. The model would then generate the actual element based on the conditional distribution for the predicted type. This model did outperform the original model it was built off of (not compared to Codex as Codex was not released by this point). It also outperformed a model where types were simply treated as another parameter.

The relative ineffectiveness of treating type information as simply another parameter could undermine my current approach, as it originally only involved adding these annotations from the static analyzer as part of the input for the training/ test data. However, this paper was mainly focused on the issue of vocabularies with infrequently used words, while my approach is focused on the more broad issue of correctness, though I may see similar results if I try to adapt their work.

[3] Neural Program Generation Modulo Static Analysis

This paper presented a quite different model that used static analysis as a weak supervisor to address the problem of code generation, specifically in the context of full method bodies (instead of just next token prediction). It used the static analysis tool to learn an "attribute grammar", which then affected the training of the neural network for the code generation. This approach did outperform GPT-3 and Codex (though Codex was unable to be tuned to this training data due to the weights for Codex being not publicly available). Despite the promising results and relation to guiding machine learning using static analysis, this paper's approach is too different from my planned approach to use it as a base model. However, it did include some information on evaluating performance in a way that is different from BLEU but still does not require compiling and running the output code, which could be useful for my approach. It also provided information about the specific static analysis checks it performed, so I could use the same static analysis checks in order to compare the performance of my approach to this approach.

Other work

The previous papers mainly covered different models I could potentially build on for the task of source code generation. In terms of finding an appropriate static analyzer, I was having some trouble. I was looking for a Python static analyzer since Python was a commonly used language in the training data for most models I saw, but most of the popular Python static analyzers like Pyflakes lacked good documentation on their internal workings (it may exist, but most discussions on them are from a user perspective, not a developer perspective). Many are open-source though, so this documentation may exist if I keep looking. The static analyzer presented by Mukherjee et al. in [3] might also be a good topic to look into, but for now I lack a definitive static analyzer that I want to adapt.

Reflection on the Initial Plan

Major Changes

Based on the results reported by Parvez et al. [2], simply adding analyzer annotations as part of the training data may not be enough to lead to any significant improvements. Fortunately, though, Parvez et al. also present an alternative approach I could try adapting, with the 2-level scheme. I plan on attempting the simple annotation strategy as a first approach ([3] still seemed to think annotations were a promising avenue of research, though this was only mentioned briefly), then trying the 2-level scheme to see if there is improvement. The usage of types versus static analyzer annotations are different in a few different ways, like the granularity of the annotations and the underlying problem being addressed (sparse vocabularies versus correctness with respective to the analyzer), meaning my approach is not just a trivial adaptation of the work done by Parvez et al.

Meeting the Milestone

In the original milestone, I planned on already finding at least one static analyzer/ model pair to work with. Parvez et al. 's model in [2] seems like a good one to build on. The original model that they built on was the SLP-Core model presented in [4]. This was actually developed in part by my mentor, so SLP-Core could also make a good baseline model. However, I cannot fully decide on a model without finding a compatible static analyzer that works on the same language the model was trained on. Due to the previously mentioned issues I have not been able to find one yet. I may broaden my search to both Java and Python static analyzers as Java is also a popular language for code generation training data (all 4 models cited here work on Java), and Java's properties like static types and more strict syntax may make static analysis easier.

Surprises

[2] seemed to show that simple annotations on their own are not enough to significantly help the model, though this was in a different problem context. [3] seemed optimistic about the potential of annotations as a route for future research, but this was only discussed briefly.

Revisions to the 07-400 Milestones

If I use the models in [2] or [4] as a baseline, I will likely need to change some of my milestones with regards to modifying the loss function of the model. [2] and [4] use n-gram based models, which do not train using explicit loss functions like neural networks do. In this case, these milestones will be changed to be about implementing the 2-level approach described in [2], and evaluating how it works for static analyzer annotations as opposed to using types. The approach [2] takes for evaluating the performance of the model is also what I will likely use as it does not require solving the logistical challenges of the functional correctness metric set forth in [1].

Resources needed

I will likely need some time on a GPU time or AWS credits in order to perform the training and evaluation of my model. My mentor mentioned using these for training his own models and how I will probably need to do something similar, so I will be in contact with him on how to do so. CMU does provide credits/ servers for classes, so obtaining these resources should be possible.

Bibliography

1. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. https://arxiv.org/abs/2107.03374
2. Parvez, M. R., Chakraborty, S., Ray, B., & Chang, K. W. (2018). Building language models for text with named entities. arXiv preprint arXiv:1805.04836. https://arxiv.org/pdf/1805.04836.pdf
3. Mukherjee, R., Wen, Y., Chaudhari, D., Reps, T. W., Chaudhuri, S., & Jermaine, C. (2021). Neural Program Generation Modulo Static Analysis. arXiv preprint arXiv:2111.01633. https://arxiv.org/abs/2111.01633
4. Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, ESEC/FSE 2017, pages 763–773. https://doi.org/10.1145/3106237.3106290.