(Joseph) Cole Ramos
07300
12 November 2021

Project Proposal: Augmenting Code Generation Models using Correctness Annotations

Project Mentor: Professor Vincent Hellendoorn - Assistant Professor of Software Engineering
Does various working on learning algorithms applied to code, and how to properly represent code as an input for machine learning (ML) methods

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

Project Description

Motivation

Methods for algorithmically generating code have become increasingly popular in recent years as a way to save programmers time and effort. For example, GitHub's Copilot is a code completion tool that is able to generate function bodies if given a function header and natural language description of the desired behaviour. Such models are usually created using machine learning, and in particular, by applying natural language processing (NLP) models to the problem. However, code has certain properties that can cause NLP techniques to fail. Sufficiently large models can learn to output syntactically correct programs that at first glance look like real programs, but from a correctness point of view may be nonsensical (for an example, see the last page). The goal of this project is to modify these NLP models in a way that incorporates information from "verifiers" so that they are able to more often output correct programs.

What we will do:

Many different tools that formally verify some property about software (compilers, type checkers, static analyzers) involve a process of explicitly or implicitly annotating elements of the code. For example, a symbolic execution based analyzer for determining if a program has memory leaks may annotate each line of the program with the state of various pointers at that line. The overall judgement of the verifier is a function of the annotations of the program, and the annotations themselves are a function of the content of the previous lines and annotations. The idea is that, if we annotate the training data provided to NLP-based code generation models, that these models will learn to add these annotations to their own generated programs. Work done in natural language problems [6] shows these types of annotations on their own, without any changes to the method's loss function, may help improve the performance of the model. The process of forcing the model to generate its own annotations may give the model extra syntactic and structural information for what correct code looks like, without even explicitly telling it anything about what these annotations really "mean". We can additionally try modifying the loss function of the NLP method's to incorporate information from these annotations, such as by penalizing generated code that has more memory errors. How exactly to modify this loss function to account for the annotations is an open question that this research will attempt to address. We plan to retrain a currently existing model using our annotation method in order to see if this method does actually lead to improvements in the model's ability to output correct code with respect to the verifier.

Project Goals

75% goal

- Retraining of a previously existing NLP based code generation model using the same training data set, except with annotations
- No modification to the loss function
- Experimental results comparing the performance of the original model and the new model trained on the annotated data

100% goal

- Retraining of a previously existing NLP based code generation model using the same training data set, except with annotations
- No modification to the loss function
- Experimental results comparing the performance of the original model and the new model trained on the annotated data
- Experiments should be done with at least two separate verifiers/ annotaters to investigate the generalizability of the technique. These verifiers should deal with different classes of bugs

125% goal

- Retraining of a previously existing NLP based code generation model using the same training data set, except with annotations.
- Experimental results comparing the performance of the original model and the new model trained on the annotated data.
- Experiments should be done with at least two separate verifiers/ annotaters to investigate the generalizability of the technique.
- An additional round of experiments with the models that involve modifying the loss function for training to penalize incorrect code according to the judgments by the verifier.

Milestones

1st Technical Milestone for 07-300: End of Semester: Finding a NLP-based code generation model to improve upon that has an easily accessible and usable training dataset. The training dataset should preferably be already compiled or source code that requires no compilation. Trying to automatically compile code during the training process would be a large technical obstacle due to the need to standardize the development environment and account for dependencies. I also plan to find at least one verifier/ annotator that works in the same language as the originally trained model. While many verifiers do these annotations implicitly, it may be harder to find one that can be easily adapted to produce these annotations in text form. I will also find backup candidates in case any issues arise when using the model or verifier in 07-400. This part may involve additional research to make sure I understand how each of these models/ verifiers works so that I know they will be compatible with my approach.

07-400 Bi-weekly Milestone 1: February 1st: Retrain the original NLP model locally with no changes to make sure that the model when trained in my environment performs similarly to what was claimed in the model's publication. Ensure the verifier runs locally and that the source code for the verifier also correctly compilers locally. Deal with any configuration issues.

07-400 Bi-weekly Milestone 2: February 15th: Research potential adjustments to the NLP model's loss function to incorporate data from these annotations. Have a working version of the annotator, and annotate all the training data from the original model with our correctness annotations.

07-400 Bi-weekly Milestone 3: March 1st: Research methods for benchmarking the performance of the code generation model. The original model's publication should likely include information on this, but small adjustments may need to be made for the new model. Perform small tests of feeding in the annotated training data, and develop a method of removing the annotations from the auto generated code: if the model is trained using annotations, then it should also output annotations. To actually run the generated code, these would likely have to be scrubbed unless the language itself already supported these types of annotations.

07-400 Bi-weekly Milestone 4: March 15th: Fully train the model without modifying the loss function using the annotated training data. Perform qualitative comparisons between it and the original model.

07-400 Bi-weekly Milestone 5: March 29th: Run experiments to get quantitative data comparing the new model and the original model. This may involve retraining both models from scratch several times to account for randomness in the training process.

07-400 Bi-weekly Milestone 6: April 12th: Get the second verifier/ annotator working, annotate the training data with this new annotator, and repeat the experiments from milestone 5. This should be easier after having done this whole process for another annotator already.

07-400 Bi-weekly Milestone 7: April 26th: Implement an adjusted loss function that accounts for the correctness of annotations. Rerun the experiments from milestone 5 using this new adjust loss function. If there are multiple potential types of loss functions that can account for this verifier information, then try as many as time permits.

The paper/ writeup for the project will start being developed after we have experimental results in milestone 5, and will be worked on throughout milestones 5 to 7.

Literature Search

There are two main topics for the project: verifiers and models.

Verifiers

- [1] - This paper described a tool that tried to detect and fix memory errors in C/C++ using a process of symbolic execution, where the state of objects was kept track of at each non-trivial line of the code. This paper, combined with background knowledge on type annotations, was the inspiration for incorporating annotations into the training process for code generation models
- [2] - Described a static analysis tool for detecting/ correcting heap errors in C/Java. May not be a good fit since it seemed to do most of its analysis in an intermediate language that likely will not have a lot of training examples.

I will likely need to continue looking for more examples of verifiers, as while I have read both papers for these verifiers, I have not looked at their source code to judge how easy they will be to adapt. Any verifiers that use symbolic execution or Hoare logic will be good candidates for this project, as both approaches involve implicit or explicit annotations. Strong type annotations may also be valid. Annotations from a compiler are potentially an option, but are likely not an interesting avenue for research as sufficiently large models can generally produce syntactically valid code.

Models

Based on discussions with my mentor, it seems that transformer models are quite popular for these types of tasks. My hope is to find an open-source, pretrained transformer model with a public training/ testing dataset in a language matching that my verifier supports. Most of these papers were suggested to me recently by my mentor, and have not been fully read yet.

- [3] - Paper behind GitHub Copilot. Background on current state of the art models
- [4] - Related work attempting to augment non-parametric language models with additional structural information
- [5] - Weak supervised learning for source code using static analysis
- [6] - Similar idea to this project, but done in the context of labels for normal NLP, non-code tasks
- [7] - More information on state of the art large language models

Resources Needed

I will likely need access to a computer/ server with a GPU to perform the training and evaluations of my model, as these models tend to involve a large number of parameters. It will probably be best to do it on a standard environment like AWS or a CMU compute cluster. My mentor said the university should be able to provide me with this (AWS credits). He had already gone through this process before for training his own models.

In terms of software, I plan on using a verifier and model with publicly available code and training data. The use of other people's training data and an automated annotator will save me the difficulty of having to manually collect training data, which would otherwise increase the time and resources required for this project.

Bibliography

1. Seongjoon Hong, Junhee Lee, Hakjoo Oh. “ MemFix: static analysis-based repair of memory deallocation errors for C”. ESEC/FSE 2018: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. October 2018. Pages 95–106. https://doi.org/10.1145/3236024.3236079
2. Claire Le Goues, Rijnard van Tonder. “Static automated program repair for heap properties”. ICSE '18: Proceedings of the 40th International Conference on Software Engineering. May 2018. Pages 151–162. https://doi.org/10.1145/3180155.3180250
3. 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
4. Xu, F. F., He, J., Neubig, G., & Hellendoorn, V. J. (2021). Capturing Structural Locality in Non-parametric Language Models. arXiv preprint arXiv:2110.02870. https://arxiv.org/abs/2110.02870v1
5. 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
6. 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
7. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Sutton, C. (2021). Program synthesis with large language models. arXiv preprint arXiv:2108.07732. https://arxiv.org/abs/2108.07732

Motivating Example

The below results come from a several billion parameter model trained by Professor Vincent Hellendoorn (not published). The first screenshot is the prompt fed into the model, specifying a basic tree-like data structure and the header for a desired function. The second screenshot is the code completion generated by the model. While a syntactically valid program, the response has several logical errors, like a redundant second null check, and mistakenly setting p = p.parentNode twice in one loop (once in the for loop header, and once in the body). The goal of this project is to make it so these types of logical mistakes are less common in the output of the model.

Prompt:

```
class Node {
private String name;
private final Node parentNode;

public Node(String name, Node parentNode) {
    this.name = name;
    this.parentNode = parentNode;
}

/**
* Returns the root of the tree containing this node, by recursively
* visiting its {@code parentNode} property until it comes back {@code null}.
@return The root of the node's tree.
*/
public Node findRoot() {
```

Response:

```
for (Node p = this; p != null; p = p.parentNode) {
                    if (p.parentNode != null) {
                        return p;
                    }
                    p = p.parentNode;
                    if (p == null) {
                        return null;
                    }
                }
                return null;
}
```