my thesis

short description
Bachelor thesis in the field of study “Computational Engineering” by N. Stewens
Date of submission: January 10, 2023

1. Review: Gutachter 1
2. Review: Gutachter 2
3. Review: noch einer

Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

field of study:
Computational Engineering
Institut
Arbeitsgruppe

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, N. Stewens, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 10. Januar 2023

N. Stewens

Contents

1 Introduction

1.1 Background and Motivation

Code Completion is an established feature of certain programming environments. Being able to quickly find the most likely next :term speeds up the progress of writing a script as well as guides amateur programmers in learning to utilize libraries sensibly.

There are two approaches to code completion. *Static code completion* forms a list of relevant predictions based on static code analysis and usually sorts the suggestions alphabetically. Its counterpart, *intelligent code completion* sorts the predictions based on relevancy. This is done using machine learning, more specifically transformer models which are trained on code samples to learn what is most likely to complete the statement. Intelligent code completion speeds up the coding experience for the user as they are more likely to find the appropriate next step at the top of the list.

There are a multiple pre-trained transformer models like the *bidirectional encode representations from transformer* (BERT) and the *generative pre-trained transformer* (GPT) designed for solving natural language problems.

Code completion is such a problem, however brings its own set of difficulties concerning syntax and structure of the code. Code is much more definitive than plain text. If it was treated the same as plain text, it would most likely cause a compiling error. When comparing a code snippet to a verbal description of that same script, the code snippet would have to stay the same to keep its meaning while the verbal description could look many different ways while still containing the same information.

Previous works by the Meta research group have tackled this issue by feeding *abstract syntax trees* (AST) to the transformer instead of only plain text. AST are a form of code representation that contain the structure of the code as well as the code itself. This makes it possible to make reasonable predictions and generate code that will compile.

Yet there are other ways to represent code than AST. For instance, a *control flow graph* (CFG) provides information about the sequence of statements in a code snippet. The edges of the graph show in what order the code is traversed. This gives additional information about structures such as conditionals and loops.

For example a regular language model would have to interpret a while-loop as something that is to be repeated and understand what a condition is. While it is possible to train a language model to do those things, adding information about the control flow will reduce training time and the amount of training data needed. The aim is also to ensure the model works properly.

(example graph)

This research thesis will evaluate how well a CFG performs in code completion compared to AST. To take on this task, the same transformer model will be adapted to take CFG as the input instead of AST.

2 Foundations

This chapter serves as an introduction to transformers and.

2.1 Transformers

[1]

3 Generating data

3.1 Control Flow Graph

Using Abstract Syntax Trees to train a transformer model, rather than plain code files shows to improve results greatly. [source?] This is due to additional structural information contained in the Abstract Syntax Tree. This ensures there is no ambiguity when processing the code. [...more explanation]

However, Abstract Syntax Trees are not the only option for this code completion problem. Control Flow Graphs provide more structural information that could be utilized to further improve the score of this code completion task.

3.2 Building the CFG file from py2cfg

In order to compare the use of control flow graph (CFG) data to that of Abstract syntax trees we first need to generate the CFG data in a way that is processable. In Marcel's work, the data is fed into the transformer as a json-file containing a list for each body of code.

3.3 Traversing the CFG

Bibliography

[1] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers”, in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 150–162.