Improvement of Source Code Conversion for Code Completion

Mika Turk
Supervisors: Maliheh Izadi, Arie van Deursen
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Improvement of Source Code Conversion for Code Completion

Mika Turk
Delft University of Technology
Delft, The Netherlands

***Abstract*—Code Completion is advancing constantly, with new research coming out all the time. One such advancement is CodeFill, which converts source files into token sequences for type prediction. To train the CodeFill model, a lot of source files are needed which take a long time to convert before training can begin. Converting the file the end-user is working on for completions is also essential for the total latency as longer files can affect the experience of using the model. In this study we aimed to improve the performance and success rate of this conversion. Our results indicate that we increased both the performance by 83 times or more depending on the input file length and the success rate by up to 45%.**

1. Introduction

Code Completion is the task of using the source code a developer has already written and predicting what they will write next. Having a computer predict what a developer will write next instead of them having to type it themselves saves them time if the suggestions are good and appear quickly. CodeFill has introduced a conversion from python source code to token sequences, this makes it possible for CodeFill to predict types [1]. Converting the content of the file the developer is working on is crucial as this influences the total latency from the developer typing something to the suggestion appearing on their screen. Having as little errors as possible is also important as an error in conversion results in no suggestions at all.

The main contributions of this paper are:

- An improved version of CodeFill's python conversion function, with support for multi-threading and with a higher success rate for conversion.
- An improved codebase for every part of CodeFill's conversion process, with type annotations and documentation strings.
- Extending CodeFill's conversion to a new language; JavaScript.

We make our code for all parts of this work available. [1]

2. Background and Related Work

In the past, there has been a lot of research on the topic of providing automatic code suggestions, this section will state some of the papers that have contributed to this topic regarding generating the code and developer satisfaction with the results.

Providing automatic code suggestions using machine learning has been studied before [2], they studied a specific subset of code suggestions, namely suggestions for firmware development, more specifically SSD firmware. Their goal was to build a model to auto-complete code for SSD firmware and do so in a way that keeps the source code secure, as the company developing this software cannot afford to have the source code leaked. To achieve this they set out to train a new model using GPT-2 by training it on SSD firmware and tune the parameters specifically for this purpose.

GPT-2 was also used to generate more specific files than just generic source code by [3]. They used it to generate Simulink models and find bugs in the Simulink toolchain, improving upon the results achieved by their closest competitor, DeepFuzzSL.

Another method to generate code was described by [4], they tackled the problem of generating source code in a strongly typed programming language given a label carrying a small amount of information about the code that is desired.

Generating code using a small amount of extra information in addition to the previous code as context was also described in [5], where they use a code comment to suggest an entire code block.

Generating code blocks is a topic that has had a lot of research done, another paper on this topic is by [6].

Improving existing models has been a topic of many papers because great models can often be improved to perform even better, one such example of a model that was improved was [7]. They studied many aspects of improving existing models and creating new ones and presented them as a toolbox for other model developers to use. Preparing the data properly and evaluating the model automatically were the two main contributions of the paper, but their work on integrating models with an IDE such as VS Code also contributed a lot to the research of this paper.

Most work in this field is dedicated to creating new and better ways to suggest code that developers will like, however, there has also been research dedicated to developers and their habits using these tools, [8]. They tested whether software engineers use autocompletion features differently than other developers, analyzing the acceptance

1. https://github.com/mikaturk/codefill-conversion-improvement

rates and targets of autocompletion in IDEs. Their results were insignificant to show that this difference exists but their research is still applicable.

3. Approach

To improve CodeFill, this paper focuses on the conversion to token sequences described by CodeFill [1].

3.1. Python Conversion

To uncover slow parts of the data processing pipeline, where most function calls are generic python or `pandas` functions, we decided to use a program that outlines on which lines the most time is being spent, a line profiler. By looking at the function calls and their documentation, more efficient versions can be used that utilize vectorization which significantly speeds up operations on `pandas` DataFrames. Another optimization step is not to use DataFrames at when not strictly necessary, using a list of lists and converting this into a DataFrame later can save a lot of time due to being a simpler data structure.

When giving all global variables the type `GLOBAL_VARIABLE`, the original conversion looked at the line number and text content at the same time, then selecting all rows which match both equations. This is very inefficient because doing the text comparison on such large pieces of text takes a lot of time and could be partly avoided by only doing the text comparison for rows that matched the line number we are looking for.

By looking at files that have unusual extensions like ones that do not end in `.py`, but do contain it, we increase the amount of files able to be used for training. Making tweaks to the core functions ensures that less files crash during the conversion process is another way to increase the amount of files available for training.

3.2. Javascript Conversion

To convert JavaScript source files into token sequences, we used `js-tokens` to tokenize the source files into lists of tokens, these lists contained the type of token which was used to determine the actions taken. To produce each converted line we use a list of strings to keep track of the current output line. The following list contains the actions of the conversion by the token type they match on:

- Any whitespace or comment: Skip this token
- Line terminator: Check if the current output list is empty, if it is, skip, else concatenate the list with spaces in between the strings, add the result to the end of the output file, empty the list.
- Any other token type: Add the token type as a string to the current output line list.

4. Experiment Design

Our improvements will be tested on the CodeFill dataset, which was collected from GHTorrent [9] where they only retreived code from repositories with more than 20 stars that were not forked (58k repositories).

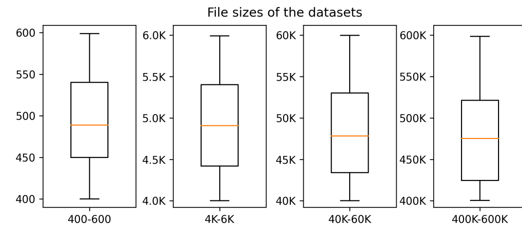

Figure 1: File size distributions of the evaluation dataset

4.1. Dataset

The CodeFill dataset contains a small amount of large files, 616 out of 1.8 million files are over 1 Megabyte in size, we do not include them in our further research since they are prohibitively expensive to convert and are largely repetitive. The part of the dataset we will be using for evaluation consists of four seperate datasets, in 4 different orders of magnitude: 400-600, 4K-6K, 40K-60K, and 400K-600K bytes. Figure 1. These four datasets were chosen because they are not too small that the files would be unrealistic for most developers and not too big for the same reason. The reason we only chose to use a small range each time is to keep the file sizes for each dataset close enough that the resulting conversion times would be similar enough that using the average of this would not paint a skewed picture. After picking these datasets, we reduced the datasets down to 1000 for the three smaller filesizes and 500 for 400K-600K. To reduce the datasets, we will use `random.choice` with a fixed seed so we get the same files across runs.

4.2. Configuration

We use the python package `line_profiler` to examine the performance of the conversion along with custom timing scripts that track the execution time of each file. We use python package `joblib` to use make use of all cores while running the conversion. If not specified otherwise, we use the configuration used in the CodeFill paper [1]. Our experiments are conducted on a machine with two NVIDIA GeForce RTX 2080Ti GPUs, an AMD Ryzen Threadripper 1920X 12-Core Processor, 64 Gigabytes of quad channel DDR4-2400, and three Samsung 960 EVO 1TB NVMe SSDs in RAID-0 (scratch disk). The CPU

4.3. Research Questions

Main Research Question: To what extend can we improve CodeFill's conversion technique in terms of speed and coverage?

Sub Questions:

- How can we speed up the conversion from python source files to token type sequences?

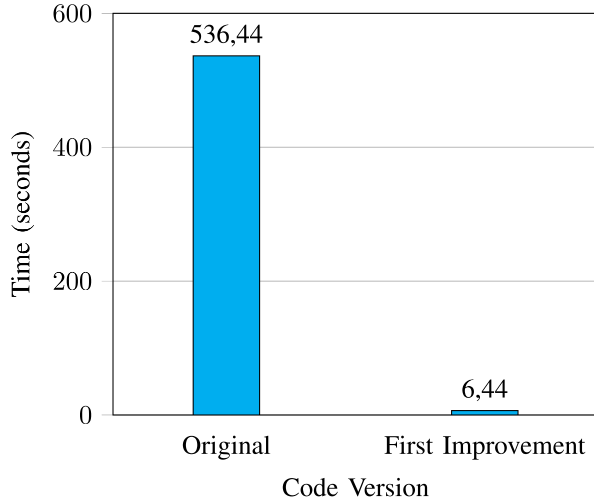

Figure 2: Original vs First Improvement

- How can we increase the amount of files that can be successfully converted?
- Can we extend the conversion to other dynamically-typed languages such as JavaScript?

4.4. Evaluation Metrics

The metrics we will be using to evaluate the improvements to CodeFill will be execution time of the conversion and the amount of files that were successfully converted.

5. Results

When talking about `DataFrame`, we are talking about a `pandas.DataFrame` from the python library `pandas`

After running several files with `line_profiler`, it became clear that the performance problems the original conversion suffered from mainly came from running `pandas.DataFrame.append` in a loop, this function adds a row to a `DataFrame` but also copies every row before it into a new `DataFrame` which results in a time complexity of $O(n^2)$. This results in the function taking 529.69 seconds out of a total of 536.44 seconds for a 77 Kilobyte file, this is 98.74% of the time of the entire function. By adding to a native python list and creating the `DataFrame` afterwards, these operations take almost no time at all compared to the rest of the function, as a result, the conversion time drops to 6.44 seconds, a 98.80% decrease, or 83x speedup.

After optimizing for single files, we removed the dependency on the file system for intermediate results and kept everything in memory until the end, this allowed us to use a multithreading library like `joblib` and parallelize the conversion on multiple threads, making the conversion of large amounts of files much faster as the independent conversion now run in parallel.

Secondary performance problems were due to inefficient use of libraries like stringifying the token then using string methods to extract the type, this was replaced with accessing the type directly. The `DataFrame.apply` function call was replaced with a function that directly achieved the task at hand, namely `DataFrame.isin` and assigning the value in the text column to the output of this function.

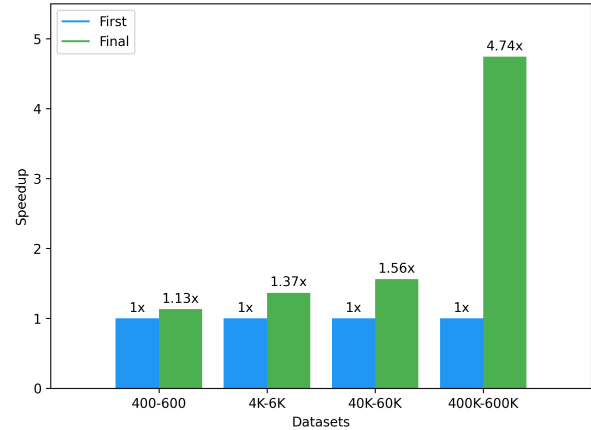

Figure 3: Conversion Speedup by Dataset

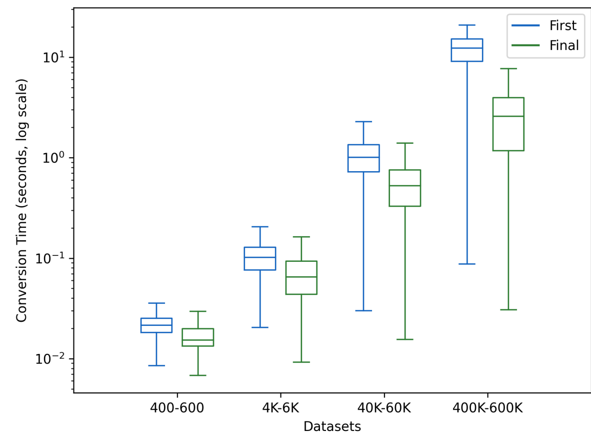

Figure 4: Conversion Time Boxplots by Dataset

After running the 4 datasets through the function after the first improvement and the final conversion function, we obtained the time it took each function to process each dataset, they can be seen in Figure 4

We created a simple JavaScript conversion that converts JavaScript into token sequences, example input and output can be seen in Figure 5. It could be more advanced, we talk about this more in section 9

Our new conversion algorithm is much faster than the previous version, as we removed a critical, but our new version makes this linear thus improving the time taken for larger files the most.

6. Discussion

Our results contribute to the speed, reusability and readability of the conversion process, we think this will make

```
import { join } from 'node:path'

/**
 * Says hello
 */
function sayHello() {
  console.log(`Hello world! ${join} hi ${1+2}`);
  // Uses built-in path.join
  console.log(join('./hello', 'world'))
}

sayHello();
```

```
IdentifierName Punctuator IdentifierName Punctuator IdentifierName StringLiteral
IdentifierName IdentifierName Punctuator Punctuator Punctuator
IdentifierName Punctuator IdentifierName Punctuator TemplateHead IdentifierName Punctuator StringLiteral Punctuator StringLiteral Punctuator TemplateMiddle NumericLiteral Punctuator NumericLiteral TemplateTail Punctuator Punctuator
IdentifierName Punctuator IdentifierName Punctuator IdentifierName Punctuator StringLiteral Punctuator StringLiteral Punctuator Punctuator
Punctuator
IdentifierName Punctuator Punctuator Punctuator
```

Figure 5: An example JavaScript code snippet and its converted version

it easier for others to work with CodeFill and adapt the conversion process to even more languages in the future. The first improvement is a big help for all file sizes but later changes have lesser impact on smaller files as the optimizations are focused parts of the code that have greater complexity than $O(n)$. Speedup of the conversion is an essential part of the progress towards using more data to train the machine learning models of today and tomorrow. This research contributes to more capable models trained on bigger datasets than ever before. Finally, the increase in code quality and readability will make for easier development of conversion functions for other languages so CodeFill and other Code Completion models can work with even more languages. The JavaScript conversion was cons

7. Threats to validity

The benchmarks that determined the speedup of removing `DataFrame.append` were ran using the `line_profiler` tool which increases overhead, without this overhead the function ran twice as fast.

The final conversion time benchmarks were ran with the same amount of input files, but not the same amount of output files, The amount of files that were successfully converted by the two conversion functions can be seen in Table 1

8. Conclusion

The new conversion functions speed up the process significantly, this allows the use of even larger datasets which can be used to achieve even greater accuracy. The changes in the conversion also resulted in a higher proportion of files being available to process, the 400K-600K dataset went from 307 to 446 out of 500, a 45% increase.

TABLE 1: Conversion Success Rate

Dataset	#Files	Version	#Successful
400-600	1000	First	798
		Final	936
4K-6K		First	752
		Final	868
40K-60K		First	505
		Final	732
400K-600K	500	First	307
		Final	446

9. Future Work

This paper has improved the performance of the conversion but there's always more performance to be gained, for example:

- Rely even less on `DataFrames` and use specific data structures to solve the problem at hand instead of a one size fits all solution. Mainly the replacement of global variables as this code is currently not optimal.
- Remove unnecessary string copies
- Using the `dis` library properly by calling its `dis.Bytecode` function instead of `dis.dis`, this outputs the details we want from the disassembler in a proper data structure instead of stringifying it and printing it to stdout, then turning it into a string and parsing it.

Converting Python 2 syntax into Python 3 syntax would allow for even more files to be converted which should get the amount of successfully processed files that do not contain errors themselves even closer to 100%. Reducing the error rate can also be achieved by fixing the ”None” bug. A last resort to get the desired performance could be to port the `converts` to a compiled language like C++ or Rust and call the function from Python, this would require more effort but would be particularly helpful when improving the data structure and removing as many memory copies as possible. JavaScript conversion is currently done using only a tokenizer, to improve the conversion by outputting a more detailed file, an AST library like `acorn` can be used, this is more work than using a tokenizer because each AST node type has a different structure and thus requires hand tuning of the output syntax. Using the same approach as we did when working with just tokens, a converter that has the same or better detail in the converted form can be created, with the trade-off that it takes more work to do this properly for JavaScript using `acorn`

10. Responsible Research

The code for all our research has been made available and we are using a public dataset from CodeFill [1]. Our

results can be reproduced by running the various scripts and notebooks in the repository.

Acknowledgments

I would like to thank Georgios Gousios for advising me and providing the server on which I did my research. I would also like to thank Frank van der Heijden, Jorit de Weerdt, Marc Otten, and Tim van Dam for their support during the project, we completed our separate research together and helped each other out during most steps of the process.

References

[1] M. Izadi, R. Gismondi, and G. Gousios, “Codefill: Multi-token code completion by jointly learning from structure and naming sequences,” 2022.

[2] J. Kim, K. Lee, and S. Choi, “Machine learning-based code auto-completion implementation for firmware developers,” *Applied Sciences*, vol. 10, no. 23, 2020.

[3] S. L. Shrestha and C. Csallner, “SLGPT: using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain,” *CoRR*, vol. abs/2105.07465, 2021.

[4] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, “Neural sketch learning for conditional program generation,” in *International Conference on Learning Representations (ICLR), 2018*, 2018.

[5] G. Heyman, R. Huysegems, P. Justen, and T. Van Cutsem, “Natural language-guided programming,” 2021.

[6] M. Hammad, Ö. Babur, H. A. Basit, and M. v. d. Brand, “Deepclone: Modeling clones to generate code predictions,” 2020.

[7] B. Barath, “Improving code completion with machine learning,” 2020.

[8] R. Amlekar, A. F. Rincón Gamboa, K. Gallaba, and S. McIntosh, “Do software engineers use autocompletion features differently than other developers?,” in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 86–89, 2018.

[9] G. Gousios and D. Spinellis, “Ghtorrent: Github’s data from a firehose,” in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 12–21, 2012.