Tiny Starcoder for Code Completion

Mateusz Wojciechowski

October 2024

Contents

1 Choosing files for the completion task **3**

2 Picking the code completion tool **3**

3 Deciding how to prepare data for completion **3**
3.1 Randomly picking parts for completions 3
3.2 More structured approach for picking completion parts 4

4 Generating completions **4**

5 Manually assigning labels **4**

6 Automatic metrics for completion evaluation **5**
6.1 BLEU 5
6.2 ROUGE-L 5
6.3 CodeBLEU 6

7 What do metric values say about completion quality? **6**

8 Conclusions **7**

1 Choosing files for the completion task

The first task in this project was finding files in my Github repositories which were well suited. I thought that they should be diverse enough to contain some common python code structures like classes, functions, conditional statements and loops. I also wanted them to contain various length of the mentioned structures to tackle completions in different situations. After some time I decided to go with files you can find in the *files_for_completion* folder.

2 Picking the code completion tool

I analysed the tools provided in the task description, and read about them in detail. Since I didn't have substantial computational resources I picked the *tiny_starcoder*. Its documentation was clear and it was lightweight enough to run on my laptop without any issues.

3 Deciding how to prepare data for completion

This part of the task was probably the biggest challenge I faced. I didn't really know how the tiny_starcoder would perform thus I couldn't decide how to prepare the data that I was going to feed to it. I had a few ideas, I managed to implemented all of them but finally choose one of them to perform all tests on.

3.1 Randomly picking parts for completions

I implemented choosing completion parts on random and as easy as it might seem, it wasn't. I wanted to check many possibilities. Because of that I tried a few things: I took one or more (up to 3) lines of the code as middle, and then based on the length of the file I selected lengths of both prefix and suffix. Another slightly modified approach was to try and randomly **begin "middle" with n-th char** of the picked line (I wanted to simulate completing a line that a user has already written to some extend). I also tried making suffix and prefix longer by assigning them **all the remaining code** below and above "middle" (I thought the tool lacks more context for the completion). You can find my efforts in the *generate_random_structured_positions.py* and *generate_structured_positions_full_context.py* in the *unused_code* folder.

3.2 More structured approach for picking completion parts

As I' am writing this subsection you might already guess my previous approach wasn't successful. The problem was that it didn't matter how I picked completion parts randomly the tiny_starcoder wasn't able to complete them well at all. As all the results I obtained using that splitting methods were inaccurate I had to try something else. To assign some meaningful labels and then judge how well the metrics evaluate completions I had to have some diversity in my dataset. This was when I implemented an algorithm which **selected completion parts in different types of python code structures** such as: conditional statements, loops, classes and functions. The part that was meant to be completed was somewhere in the middle of those structures, and the rest of them were prefix and suffix. I used this approach to generate *completion_data/structured_positions.json* file which I then used to get output from the tiny_starcoder The whole process is implemented in *generate_structured_positions.py*.

4 Generating completions

With the files divided into various prefix, middle, suffix structures it was time to feed the data into tiny_starcoder. Even after i achieved some success with splitting the file I still often had a problem with code generation. It seemed to me that no matter what I did the **tiny_starcoder didn't generate the code in the right place** if there was some kind of not-empty suffix. The new code was generated mostly at the end, after prefix an suffix. Even though I tried formulating the prompts differently and changing models parameters I still couldn't solve this problem. Because of that in some cases I ignored the order of generation and just focused on what the model has generated as a completion. I also ignored the fact that the model might sometimes run out of tokens and generate code with a wrong syntax, as setting max_new_tokens to higher values caused even weaker model performance. I obtained completions for each example from my dataset, and saved the data into *completion_data/completions.json* file. You can find this part of the project in *generate_completions.py*.

5 Manually assigning labels

That part of the project was another that required a lot of thinking and attempts. I knew from the start that simply viewing a completions as correct or incorrect would

be too superficial and unsuitable for further analysis with automatic metrics. Since the completions turned out to be quite diverse I managed to come up with a few labels. I started with the simplest one **Perfect_Match** for code that was perfectly generated and identical to the missing part. There were also some examples of code that was written differently but was equivalent in terms of functionality so I assigned them **Equivalent_Functional_Match** label (I also included cases when the middle was correctly written but located in the wrong place as described above). Tiny_starcoder also generated some code which functionality differed from the original middle. Such examples received **Partial_Functional_Match** label. The last category contains code that is fundamentally different from what was required and has a **Functionally_Different**. The complete dataset with assigned labels is located in *completions_labeled.json* in the *completion_data* folder.

6 Automatic metrics for completion evaluation

Apart from calculating exact match and chrf I choose three additional metrics using which I performed completion evaluation. I described briefly each of them below.

6.1 BLEU

BLEU (Bilingual Evaluation Understudy) is a metric designed to evaluate the similarity between a generated sequence and one or more reference sequences. It was initially developed for machine translation but is now also applied in evaluating generated code. It calculates the geometric mean of the precision of n-grams between the generated text and the reference text(s) and applies a brevity penalty to penalize short translations.

6.2 ROUGE-L

ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation) is a metric that measures the longest common subsequence between the candidate and reference sequences. It evaluates recall, precision, and F1-score based on the longest matching subsequence. Precision measures how much of the candidate text matches the reference, while recall measures how much of the reference is captured by the candidate.

6.3 CodeBLEU

The CodeBLEU metric is an adaptation of BLEU designed specifically for programming languages. It extends the BLEU metric to account for language-specific features by incorporating syntactic and semantic information.

7 What do metric values say about completion quality?

I will briefly describe my observations and intuitions about each metric's performance on differently labeled completions. **Exact match**, as expected, precisely indicates whether a completion is identical to the intended code. While valuable in cases of exact correspondence, it lacks deeper insight when assessing even slightly varied completions. The **chrf** metric doesn't appear to be effective for evaluating code completions. Since it measures character-level similarity, it fails to account for functional accuracy, often assigning lower values to correct but structurally different completions. **BLEU** can sometimes rate non-working code as a good completion. Although it generally improves evaluation by capturing structural similarity, it may overvalue completions that are functionally different but visually closer to the reference than code that works correctly but varies structurally. **ROUGE** tends to excel at identifying perfect matches, which yield a score of 1.0. Compared to previous metrics, ROUGE seems better at indicating code quality, as functionally similar code often scores higher than functionally incorrect. However, ROUGE may score functional but differently formatted code lower, as it favors structural similarity. Out of all the metrics tested, I find **CodeBLEU** aligns most closely with my judgment. CodeBLEU assigns the highest values to perfect matches but also appropriately high values to functionally equivalent code written in a different style. I believe **these two categories should stand out in code completion evaluation**, as they represent the ultimate goals of accuracy and flexibility in code completion tools. It does on the other hand have some downsides as in some situations it might overvalue partially correct code. I calculated those metrics for comparing completion with original "middle" as well as for comparing original code with the code after completion. My conclusions about the metrics are more or less the same.

To get values of all metrics for each generated completion I have written *calculate_metrics.py*. Results were saved into *results_data/results.json* for completion and "middle" comparison and into *results_data/results_code.json* for the comparision of whole code parts.

8 Conclusions

Tiny_starcoder definitely was much worse at completing code than I expected. On the plus side the problems I encountered while doing this project made me try a few interesting approaches and made me realize how difficult of a task it is to build an effective code completion model. Surely the next time I need such tool I would probably go for a bigger version of starcoder or some model equivalent to it. This experiment definitely forced me to get creative and at some points tested my patience, nonetheless at the end of the day it was fun to work an and introduced me to a new field of machine learning which I would surely like to explore.