JetBrains Internship application

Part 1:

First thing to do was, naturally, the script for dataset creation. I found 3 suitable recent projects that could work well for this task. The first I chose was pure python networking project for building TCP protocol over UDP from scratch. I expected this project to work the best. The second dataset was created from ReactJS with Firebase as real time database for a local event in my village. There was also CSS in this dataset, because I wanted to know, how the model deals with this kind of code. The last was again ReactJS project but with PHP backend for easy file management website.

After I prepared these 3 repositories, I wrote a simple script that parsed those files into JSON format like this:

“prefix” – which is the code before cursor

“middle” – which is the hidden code that model will generate

“suffix” – the code after middle

Part 2:

I started with implementing the bigcode/tiny-starcoder model from huggingface. At the beginning it looked great, and I was surprised it didn’t take long, and it generated something that was not nonsense. However, after some time I realized that the model absolutely ignored the “suffix” part and wrote much longer “middle” or hidden part of code that I wanted to. The problem was that the tiny-starcoder is not meant for “Fill In the Middle” or FIM generation. So, I was looking for a model that was capable of FIM, I tried more of them but the bigcode/starcoder2-3b showed good results and the starcoder 2 supports FIM. There were problems, that the model still ignored some parts of the code or generated parts of code twice. After reading some discussions on the internet, I tried few recommendations, and it started working.

After the model was generating code that made sense, I implemented three evaluation metrics: exact match, chrf and Levenshtein distance. These are metrics that are being commonly used. (Face, Metric: chr_f, n.d.), (Face, Metric: exact_match, n.d.) (Face, Metric: levenshtein, n.d.)

I chose **exact match** because it is the easiest and most straightforward method to see if model filled the code correctly. I think is good to see if the model was 100% correct and it also informs me that the code is the same and I don’t have to manually check.

ChrF++ (Character F-score) is a metric that evaluates characteristic of match on a character level and takes into consideration the substrings. It says if the model generated close correctness even if the results are not identical. The bigger the number, the better.

Levenshtein Distance (Edit Distance) calculates how many operations (insertions, deletions, changes) are needed to get from generated code to the original. Sometimes the distance is only a few changes, and the prediction is the same. The smaller the number, the better.

Combination of these 3 metrics covers different kind of mistakes, that can occur in generation. Exact match informs if the model generated identical answer, chrF++ the information of relevant closeness between generated and real output and Levenshtein distance helps understand how many changes are needed to improve generated code. So, combination of these metrics provides complex evaluation of results.

Part 3:

The best results were on the third dataset, likely because of factors such as:

React (JavaScript) and PHP often have clear, consistent coding patterns, the model could have been better trained on JavaScript and PHP code as they are popular languages and both, ReactJS and PHP codes usually have clear, specific syntaxes.

The evaluation of this dataset:

Exact match:

21 from 49 are exact match

28 from 49 are different

CHRF: 81.69639 (average)

Levenshtein distance: 19.65306 (average)

From my manual review I find out, that many of the 28 not identical results were due to different variable naming or different syntax in the SQL queries (in the PHP) or I left a comment that the model did not expect and therefore not generated that. However, there were around 8 results that were indeed wrong and would cause an error.

The **Character F-score** metric correlates the best with my own judgement, as it captures similarity at the character level rather than requiring exact matches. It does not penalize small deviations as strictly and tends to score higher for “reasonable” answers that are similar to the original — just like a human evaluator would.