LLM Code Generation Evaluation

Vojtěch Kváš
ČVUT - FIT
vojtechkvas1@gmail.com

May 2, 2025

1 Prepare code completion dataset

The data sample was taken from the `member` module of the backend component of the new information system for the Sincoolka club. The system is implemented in `Java`. Since Java tends to produce verbose code, the first 300 characters were skipped. Then, a random start and end point were selected within the remaining content to extract a middle section. The length of this middle part was also chosen randomly, with a maximum limit of 200 characters. This limit correlates with the average size of a typical function. A total of 41 samples were generated for prediction purposes.

Data preparation was performed in the Jupyter notebook `data_pre_processing.ipynb`.

2 Evaluation pipeline

2.1 Automatic evaluation

For automatic evaluation, the `bigcode/tiny_starcoder_py` model was selected due to its relatively small size of 164 million parameters, which allows it to run efficiently on modest hardware. Despite being described on Hugging Face as *"trained on Python data"*, the model is also capable of generating Java code.

The code related to this task is contained in the Jupyter notebook `automatic_model_evaluation.ipynb`. The most significant challenge encountered was loading the CodeBLEU metric, as it had dependency issues. This was resolved by installing CodeBLEU directly from its GitHub repository.

Selected metrics were:

2.1.1 Exact match

Exact match evaluates whether the predicted output is identical to the reference output, character for character. It was pick with knowlege that probably minority of prediction will gain True value. It was not suprised that no prediction was exacly correct.

2.1.2 CHRF

CHRF (Character n-gram F-score) evaluates similarity based on character n-gram precision and recall. It is especially useful for morphologically rich languages and small-scale outputs like code.

2.1.3 ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) primarily measures recall of overlapping units such as n-grams, word sequences, and word pairs. It is helpful for assessing how much of the reference is captured in the generated code.

2.1.4 TER

TER (Translation Edit Rate) calculates the number of edits needed to transform the generated output into the reference. It is an intuitive metric where lower scores indicate better similarity.

2.1.5 BLEU

BLEU (Bilingual Evaluation Understudy) measures the precision of n-grams between the generated output and reference, with a brevity penalty to discourage short outputs. It is widely used in machine translation and adapted for code generation.

2.1.6 CodeBLEU

CodeBLEU extends BLEU by incorporating code-specific properties like syntax, data flow, and structure matching. It provides a more accurate evaluation for programming languages than standard BLEU.

2.2 Manual metric

The manual evaluation metric is based on a scale from 0 to 10. This scale was chosen because it is quick to enter using a keyboard and can be easily normalized to a range between 0 and 1. The scores are assigned according to the following criteria:

- 0 - Not code at all.

- 1 - Not Java code.
- 2 - Java code that will not compile.
- 3 - Syntactically correct Java code, but semantically incorrect or nonsensical (e.g., references future lines).
- 4 - Valid code, but unrelated to the project.
- 5 - Code is valid but not related to the specific file.
- 6 - Code is related to the file.
- 7 - Some similarity to the expected code.
- 8 - Nearly usable as-is.
- 9 - Can replace the expected code with minor or no changes.
- 10 - Exact match with the expected code.

Even though the evaluation scale is defined, assigning specific values can sometimes be subjective; for instance, scores like 2 and 9 may be interchangeable depending on interpretation. The manual metric was added using script `manual_model_evaluation.py`. This script was designed to facilitate easy visual comparison of different strings, making human evaluation more intuitive 1.

3 Trust the metrics

The best way to analyze relationships between metrics is through a correlation matrix, which was calculated in the notebook `evaluate_metrics.ipynb`. Additionally, all metrics were visualized using a graph. Before plotting, all metric values were standardized to a range between 0 and 1 2.

From the graph, no strong conclusions can be drawn — no metric achieves a score higher than 85%, and only some moderate correlations between metrics are observable.

The correlation matrix 3 provides deeper insight. The worst-performing metric appears to be TER (Translation Edit Rate). In the correlation matrix, no inversion or standardization is applied, so the ideal correlation value for TER is -1, since it measures the number of edits — fewer edits imply higher similarity. However, the correlation with `manual_evaluation` is unexpectedly positive, indicating a potential misalignment in what TER measures versus human judgment.

The second weakest metric is CodeBLEU. I initially had high expectations for this metric, given that it is specifically designed for code and even supports Java syntax. However, upon inspecting the graph 2, a potential issue was identified: the model wrongly predicts import statements. Since imports can be deterministically generated by IDEs, and this behavior was suspected to distort the results, such predictions were removed from the dataset. The resulting correlation matrix 4 showed significant improvement — the correlation for CodeBLEU increased by 0.12. Nevertheless, this improvement was not enough for CodeBLEU to outperform other metrics.

Surprisingly, the standard BLEU metric demonstrated a stronger correlation with `manual_evaluation` than CodeBLEU.

ROUGE and CHRF both showed similar levels of correlation with `manual_evaluation`, and a strong correlation between each other. After excluding import-related predictions 4, their correlation with human judgment increased even further. As a result, CHRF emerged as the metric with the highest overall correlation to `manual_evaluation`.

4 Conclusion

The final selected metric is CHRF, as it demonstrated the highest correlation with `manual_evaluation` on predictions that excluded import statements. CHRF is also better suited for code evaluation than ROUGE, as it operates at the character level, making it more sensitive to small differences in syntax and structure.

A major disappointment was CodeBLEU. However, this can be partially explained by the fact that the evaluated code samples were too short to leverage CodeBLEU's strengths, such as Abstract Syntax Tree (AST) comparisons. Additionally, the quality of the predicted code was generally low. If such code were generated in my own IDE, I would likely ignore it or disable the feature altogether.

I genuinely enjoyed working on this task, as it allowed me to explore both the capabilities and limitations of code generation models and evaluation metrics in a practical setting.

There are several directions for potential improvement. One option is to fine-tune the model, which is feasible since I have at least six additional modules that could be used as training data. Another direction is to explore the use of large language models (LLMs) as evaluation metrics. LLMs could provide more nuanced, context-aware judgments that align more closely with human evaluation. Since the current model is relatively small and old, it would be easy to find a larger and more capable LLM for evaluation. The entire project is available on GitHub: `https://github.com/vojtechkvas/llm_code_prediction`.

```
--- Item 2, FILENAME: data/member/dto/MemberResponseDto.java ---
---------------- Middle Expected ----------------


    private String firstName;

    @Sche

---------------- Middle Predicted ----------------


;
    private String name;

    @Schema(example = "Benzinov")
    private

Enter manual evaluation (e.g., 0 for bad, 10 for best):
```

Figure 1: Manual metric console

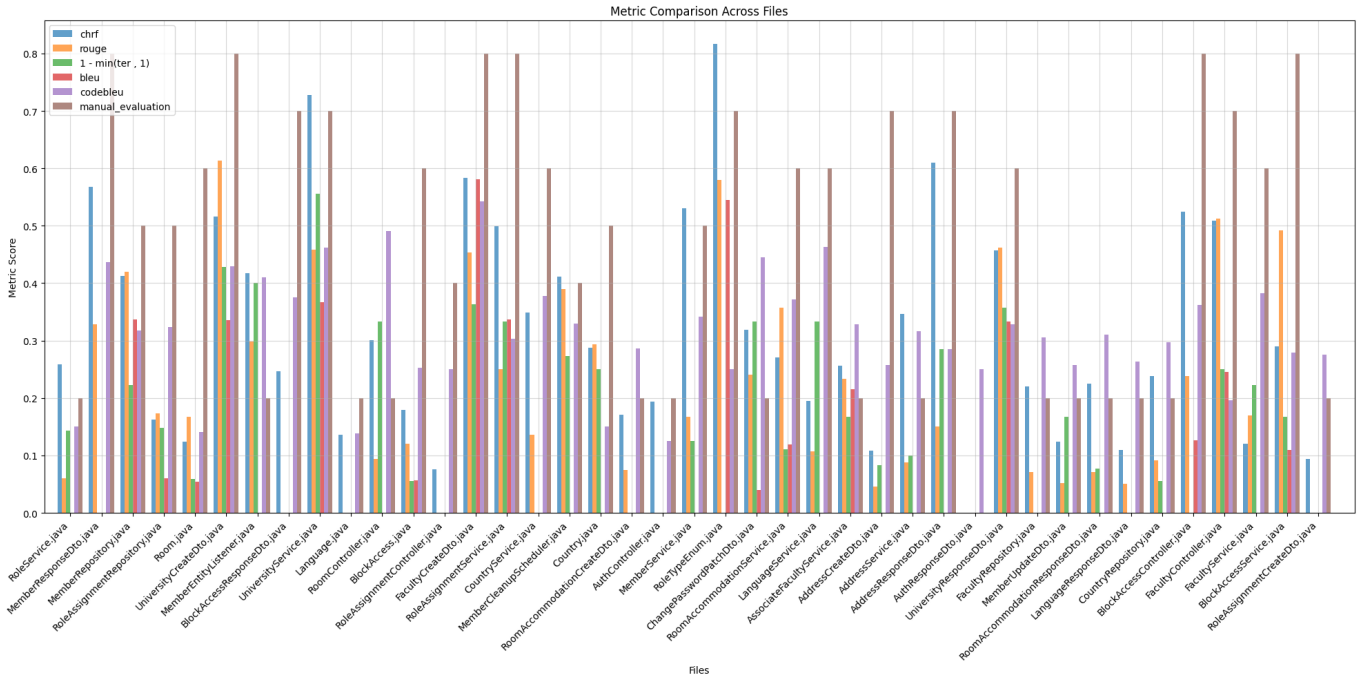

Figure 2: Graph of metrics

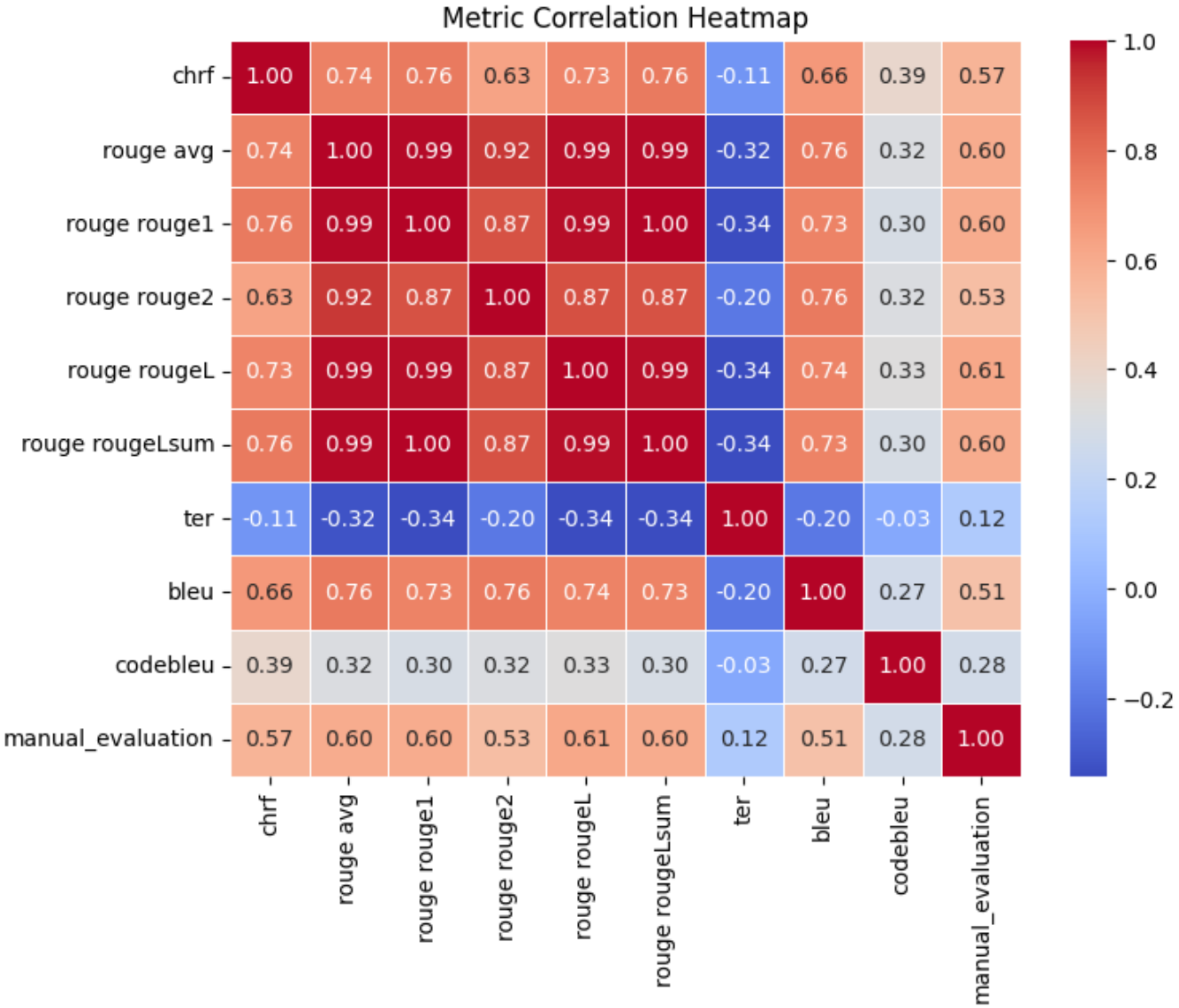

Figure 3: Correlation matrix

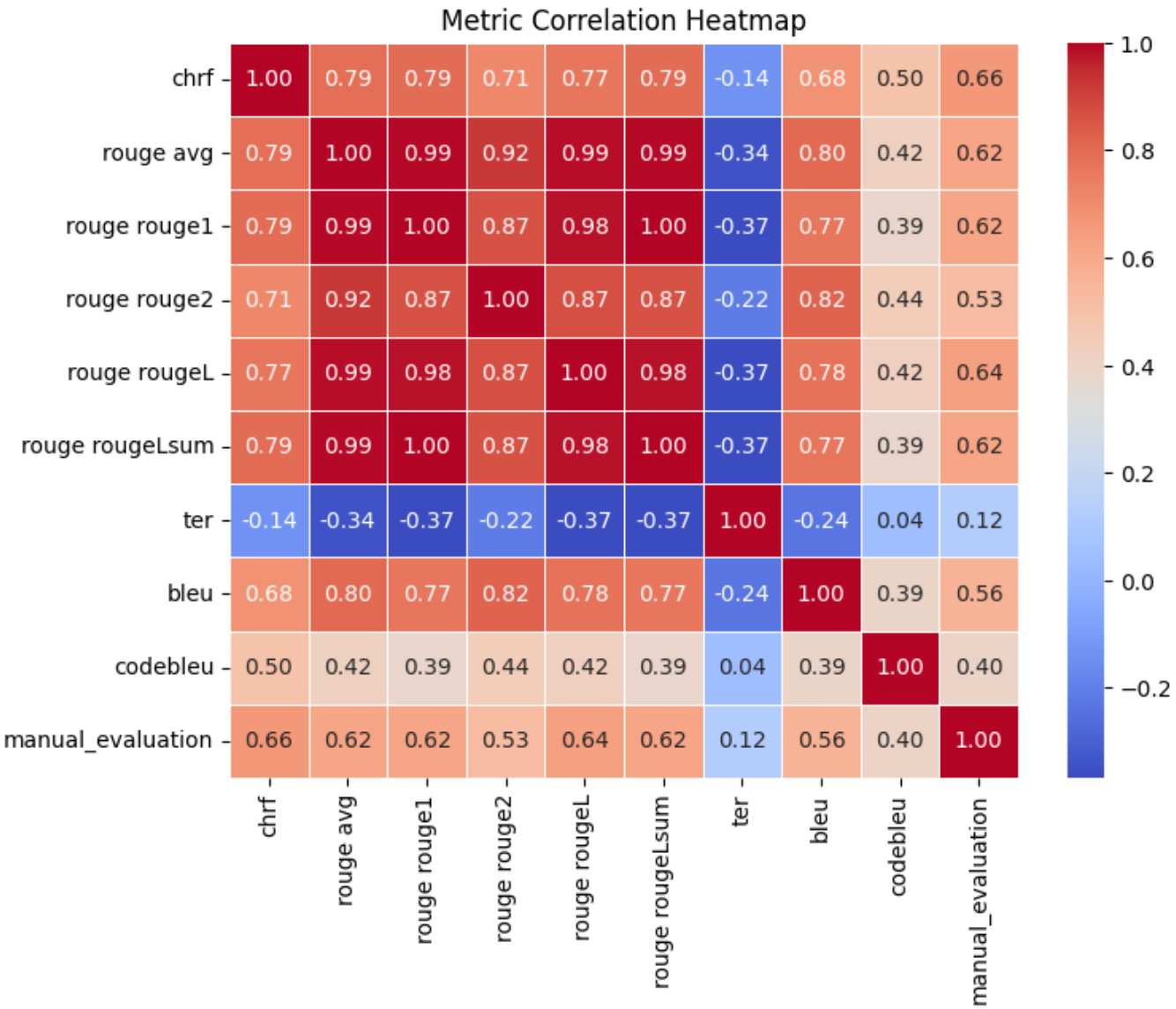

Figure 4: Correlation matrix without import