CODE COMPLETION PROJECT

The Code_Completion project aims to analyze and generate code completions using a code language model (CLM) for evaluating the quality and effectiveness of this model in generating accurate, relevant code snippets. By leveraging deep learning models and natural language processing techniques, the project seeks to create a comprehensive framework for code generation, evaluation, and improvement.

Pietro Girotto

Introduction

Code completion is a critical feature in modern integrated development environments (IDEs), significantly enhancing developer productivity by predicting and suggesting the next segment of code. This project explores the generation and evaluation of code completions using code language models (CLMs). The goal is to assess the performance of these models in completing missing segments of code and to propose methods for evaluating their accuracy and effectiveness.

In this report, a dataset of code snippets was generated from a public github repository, and each snippet was divided into three parts: the prefix (code before the cursor), the middle (the missing code to be predicted), and the suffix (code after the cursor). The task was to simulate a real-world code completion scenario, where the CLM is expected to predict the missing middle part of the code.

The selected CLM, Deepseek Coder, was used to generate completions for the missing code. These completions were then manually reviewed to assess their accuracy. Several metrics were also applied, including exact match and character-based metrics (ChrF), to evaluate the model's performance quantitatively. The correlation between these metrics and manual judgments was analyzed to propose an optimal set of evaluation metrics for code completion tasks.

This report details the dataset generation process, the use of a code completion model, the manual and automated evaluation methodologies and the key findings and learnings from the analysis.

This report is a read-through version of what is going on in the repository. If you'd like to know more about implementation, specifications or just delve deeper in the codebase feel free to start from the README.md file and work your way around the project.

Dataset Generation

To generate a dataset for code completion, several files from a github repository were selected. The files were chosen based on their variability in terms of structure and content to provide a wide range of code completion scenarios. As of right now the github extractor supports solely .ipynb .py .cpp .hpp files, but it can easily be upgraded to allow for more.

Splitting Code into Prefix, Middle, and Suffix

A custom script was written to simulate a code completion environment by splitting each file's code into three parts:

- **Prefix:** Code before the cursor, representing what has been typed by the user so far.
- **Middle:** The missing portion of the code, which the model is expected to predict.
- **Suffix:** Code after the cursor, representing the code following the missing part.

The aim was to ensure realistic completion scenarios, where the missing code could be in any position. As such it was chosen to randomly generate the samples given the contexts and the length of each prefix, middle and suffix was random as well (given an upper and a lower bound).

Example Generation

The script generated between 50 examples by iterating through selected code files. Each example consisted of a prefix, middle, and suffix, and was stored in a structured format for easy input into the code completion model. The github folder the dataset was based upon is a reinforcement learning project, as such it covers various coding scenario:

- Function implementations
- Control flow structures (loops, conditionals)
- Object-oriented constructs (class definitions, method invocations)
- Linear algebra
- Advanced DS reasoning

Dataset Summary

The final dataset consisted of 50 code completion examples, each containing:

- A prefix of 100 to 200 characters.
- A middle segment varying from 10 to 200 characters, representing the missing part to be predicted.
- A suffix completing the block of code composed of 50 to 200 characters.

This dataset provided a robust testing ground for evaluating the model's ability to handle different types of missing code segments in Python programs.

For each sample (composed of a prefix and a suffix) the middle part was predicted by the deep seek model. This model was chosen because it was fairly lightweight (for a LLM-based model), open source and it had good performances. Finally the completed dataset had also a predicted value for the middle part of the sample and it was ready for evaluation.

Manual Review of Completions

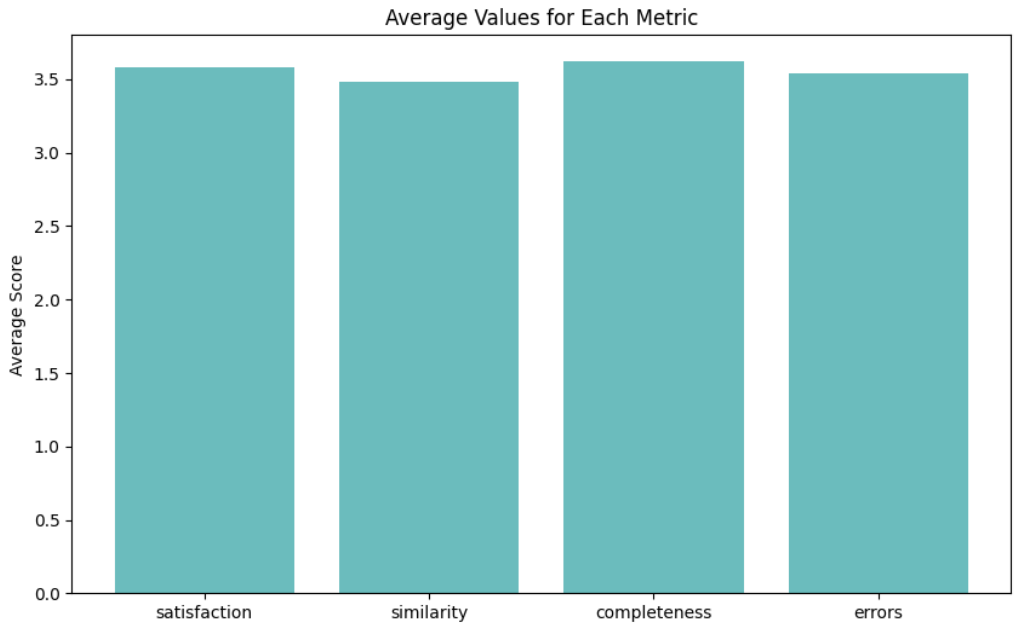
After generating code completions using the selected model, Deepseek, a manual review was conducted to assess the quality and accuracy of the completions. The purpose of this manual evaluation was to provide a qualitative assessment of how well the model predicted the missing code (the middle segment) based on the provided prefix and suffix.

Methodology for Manual Evaluation

Each completion was compared with the actual missing code from the dataset. The evaluation was based on the following criteria:

- **Overall Satisfaction**: If I were to be suggested this code while working on a project, how satisfied would I be with it?
- **Similarity**: How similar is the generated code to the original code?
- **Completeness**: With respect to the original code, how complete is the generated code? Does it include all the necessary parts?
- **Errors**: How many errors are present in the code? (The higher the score, the fewer the errors therefore the better).

Overall Review Results



It is possible to see from this graph that qualitatively speaking the model performed quite well. We have on average 3.5/5 points in all categories. For a more in-depth analysis and more plots take a look at the **evaluation/analysis.ipynb** file.
It was also clear during my evaluation that the model consistently outputted more text than needed. This will be quite a problem during quantitative evaluation because it renders many metrics not so useful.

I have decided to give a good score even to samples that generated more text than needed because if I were to be coding and in the generated part there's something I don't really need, it is not a problem. Indeed I can just easily remove it and keep the valuable suggestion.

Quantitative Analysis of Model Performance

In this section, we summarize the results obtained from evaluating the model's performance on the code completion task using various metrics. The dataset consisted of code snippets divided into prefix, middle (the missing code), and suffix, with the task being to predict the middle part based on the context of the prefix and suffix. The following metrics were applied to quantitatively assess the quality of the generated code.

Exact Match

The exact match metric is a simple and direct comparison between the generated and the correct middle code. As anticipated, the exact match score was low, with only 4.00% of generated code snippets being identical to the correct ones. This was expected due to the inherent variability in the task.

ChrF Score

Character-level F-score (ChrF) assesses the similarity between two strings based on character n-grams. The model achieved a ChrF score of 37.58. This low score aligns with the observation that the model tends to generate more code than necessary, resulting in over-generation errors. ChrF, while capturing

some degree of correctness, highlights that most generated code deviates substantially from the true answers, even when considering only character-level changes.

BLEU Score

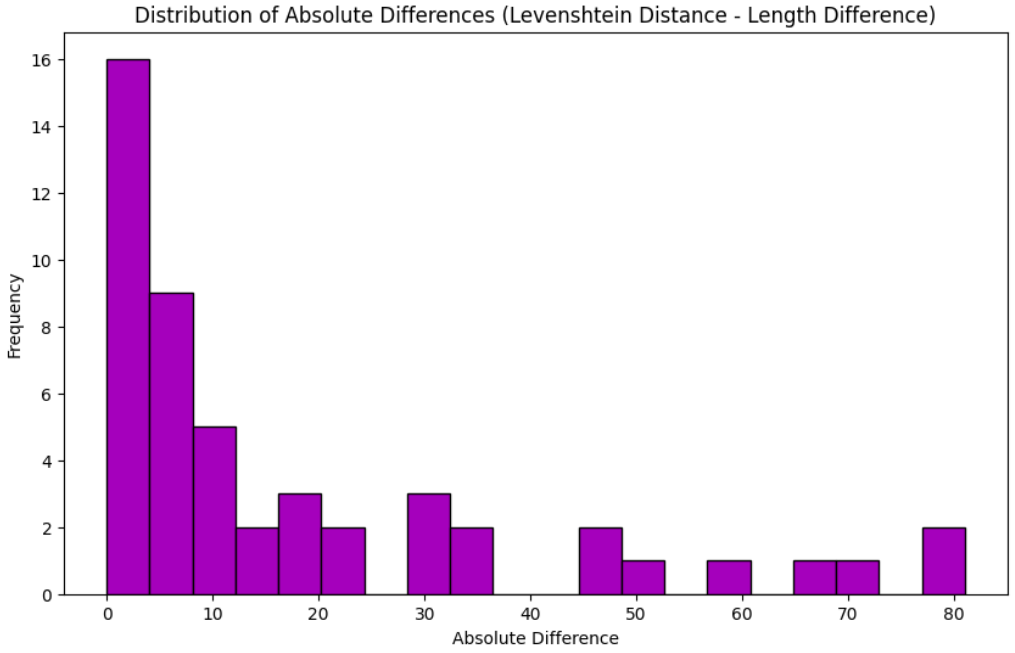
The BLEU score, a commonly used metric in machine translation tasks based on token-level n-grams, performed poorly, with a score of 0.14. BLEU is typically challenging to optimize for in code generation due to its emphasis on precise token matches. The model's tendency to produce longer outputs likely contributes to this low score, mirroring the challenges identified with ChrF.

METEOR Score

METEOR, another machine translation metric designed to capture semantic similarities, resulted in a score of 0.42. Although this score is higher than ChrF and BLEU, it remains below the ideal threshold. METEOR emphasizes recall more than precision, making it less sensitive to the model's over-generation tendencies. However, it does better capture the semantic closeness of generated code to the original.

Levenshtein Distance

Levenshtein Distance quantifies the number of single-character edits needed to transform the generated string into the true code. The analysis revealed a strong correlation between Levenshtein Distance and length difference, reinforcing the observation that the model consistently generates more code than required. The closer the difference between the Levenshtein Distance and the length discrepancy, the more likely the generated output includes elements of the correct code. This metric further substantiated the model's over-generation issue, while also confirming that, despite the excessive length, the generated code often contains the correct segments.



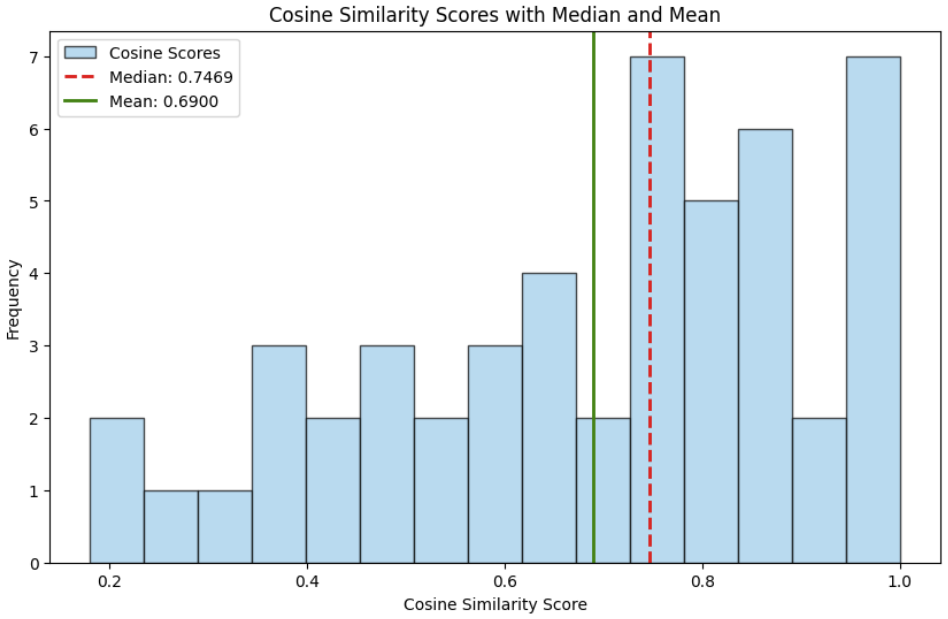
Precision-Recall-F1 Score for Tokenized Data

Given that we are dealing with code rather than human language, the token-based precision, recall, and F1 scores provided a more fitting evaluation. Using the CLM's tokenizer, the tokenized data yielded a

precision of 0.58, recall of 0.61, and F1 score of 0.59. These scores correlate more closely with my manual evaluation, confirming that while the model generates more tokens than needed, it often includes a significant portion of the correct tokens. The balance between precision and recall demonstrates that the generated code captures both breadth and detail, even if it lacks brevity.

Neural Sentence Embeddings

Finally, cosine similarity between neural sentence embeddings provided a high-level, semantic comparison of the generated and correct code. The mean and median cosine similarity scores (~0.70 and ~0.75, respectively) closely aligned with my manual satisfaction ratings, averaging 3.5 stars. These findings support the hypothesis that, while the model over-produces code, the generated output is highly semantically similar to the correct code.



Conclusions

The evaluation of the code completion task using Deepseek Coder highlighted several key insights into the model's capabilities and limitations. The following conclusions can be drawn from both the manual and quantitative analyses:

1. **Over-Generation**: A recurring issue with the model was its tendency to generate more code than needed. This was observed both qualitatively during manual review and quantitatively through metrics like Levenshtein Distance, ChrF, and token-based precision-recall scores. While the excess code did not significantly impact manual satisfaction ratings (as developers can easily remove unnecessary code), it led to lower scores on metrics that heavily penalize mismatched tokens or characters.
2. **Strong Semantic Similarity**: Despite the model's tendency to over-generate, the semantic content of the generated code often aligned well with the intended completion. Metrics such as neural sentence embeddings and manual evaluations of similarity showed that, in many cases, the model produced code that was contextually relevant, even if verbose.

3. **Traditional Metrics Fall Short**: Metrics commonly used in machine translation, such as BLEU and ChrF, proved less useful for evaluating code generation tasks, especially when the model generates longer outputs. These metrics, which prioritize token-level matches, fail to capture the broader semantic similarities that can still lead to useful code completions. Alternative approaches, such as precision-recall on tokenized data and neural embeddings, provided more reliable assessments of the model's real-world performance.
4. **Manual Evaluations Correlate with Token-Based Metrics**: Interestingly, the manual satisfaction ratings correlated more closely with token-based metrics like precision, recall, and F1 scores, as well as the cosine similarity from neural embeddings. This suggests that, for practical coding applications, developers value the inclusion of correct tokens and meaningful suggestions, even if the generated code contains extra elements.
5. **Practical Usefulness**: From a developer's perspective, the model's suggestions were often helpful, even if they were not perfect matches. The qualitative scores of 3.5/5 on average across satisfaction, similarity, completeness, and error-free execution reflect that the generated code, while imperfect, frequently provided a solid starting point for completing tasks.

Future Work

Moving forward, it would be valuable to focus on reducing over-generation and refining the precision of predictions. Fine-tuning the model to better understand the expected length and scope of the completion task could lead to improvements in both manual and automated evaluation metrics. Additionally, the development of more specialized evaluation metrics tailored to code completion, which prioritize semantic relevance and functional correctness over exact token matches, would further enhance the assessment process.

Overall, while there are clear areas for improvement, the Deepseek Coder model demonstrates promising capabilities in generating contextually appropriate code suggestions, making it a valuable tool for developers in real-world coding environments.

Metric	**Score** ([0,1] higher=better)
Exact Match	0.04
ChrF	0.37
BLEU	0.14
METEOR	0.42

Tokenized Precision	0.58
Tokenized Recall	0.61
Tokenized F1	0.59
Neural Sentence Embeddings	0.7

Metric	**Average Score** ([0,1] higher=better)	**Median Score** ([0,1] higher=better)
Overall Satisfaction	0.77	0.8
Similarity	0.7	0.8
Completeness	0.72	0.8
Errors	0.71	0.6