Project Report: Code Completion using Fill-in-the-Middle Approach

1. Introduction

In this project, I aimed to explore the capabilities of open-source code completion models, specifically focusing on the fill-in-the-middle approach. This technique allows us to predict missing segments of code based on the provided prefix and suffix, simulating a user's cursor position. The ultimate goal was to analyze the performance of these models and evaluate the generated completions against the actual missing code.

2. Methodology

2.1 Dataset Creation

I started by creating a dataset containing code snippets that would be split into three parts: prefix, middle, and suffix. The process involved:

- Randomly selecting lines of code to simulate cursor positions.
- Extracting the code before the cursor as the prefix.
- Extracting the code after the cursor as the suffix.
- Defining the missing code segment as the middle part.

I aimed to obtain between 20 and 50 examples for a comprehensive analysis.

2.2 Model Selection and Execution

For the completion task, I first chose to work with the **Tiny_StarCoder** model due to its robust architecture and proven effectiveness in code generation tasks. Then I found the CodeLlama-7b-hf is performing much better, so I used it in the end. I implemented a script to apply the fill-in-the-middle technique on the dataset, generating the missing code segments based on the prefixes and suffixes.

2.3 Evaluation Metrics

To evaluate the model's performance, I computed several metrics:

- **Exact Match**: Measures the percentage of generated code that exactly matches the actual missing code.
- **chrF Score**: A character-based F-score that captures precision and recall, particularly useful for evaluating code where exact matches may be rare.
- **BLEU Score**: Measures the overlap between generated and actual code, focusing on n-grams.
- **Levenshtein Ratio**: Evaluates the similarity between two strings based on edit distance.
- **Syntactic Similarity**: Assesses the structural similarity between the generated and actual code, leveraging tools like BERT for semantic analysis.

3. Findings

After processing a subset of examples due to GPU limitations, the following results were obtained:

- **Exact Match**: 0
- **chrF Score**: 0.3276
- **BLEU Score**: 0.3276
- **Levenshtein Ratio**: 0.1123
- **Syntactic Similarity**: 0.8496

3.1 Analysis of Results

1. **Exact Match**: The zero exact match indicates that while the generated completions may be logically sound, they do not align perfectly with the actual missing code. This is a common challenge in code generation tasks where slight variations in code can have significant implications.

2. **chrF and BLEU Scores**: Both scores were relatively low, reflecting the complexity of capturing accurate code completion in a syntactically diverse programming language.

3. **Levenshtein Ratio**: The low value suggests that there were substantial differences between the generated code and the actual missing segments, indicating room for improvement.

4. **Syntactic Similarity**: The high syntactic similarity score indicates that the generated code shared structural similarities with the actual code, suggesting the model captures the contextual essence of the code snippets.

4. Lessons Learned

1. **Model Limitations**: The performance metrics highlighted that while the model is capable of generating syntactically correct code, achieving exact matches is challenging due to the inherent variability in coding styles.

2. **Importance of Evaluation Metrics**: Using a diverse set of evaluation metrics helped provide a more nuanced understanding of the model's performance, particularly when analyzing generated code.

3. **GPU Constraints**: Encountering GPU limitations during evaluation reinforced the importance of optimizing code and model parameters for efficient execution.

4. **Semantic Matching**: Incorporating semantic matching in evaluations allowed for a deeper understanding of the model’s contextual relevance, which is essential for practical applications in code completion.

5. Conclusion

This project provided valuable insights into the capabilities and limitations of code completion models. While the results indicate areas for improvement, they also highlight the potential for leveraging advanced models like StarCoder in code generation tasks. Future work could focus on refining the model’s ability to generate exact matches and further explore the integration of semantic analysis in code completion.