GenAI for Software Development: Assignment 1

Lya Thummala
athummala@wm.edu

1 Introduction

Java code completion aims to complete code for methods or classes automatically. The N-gram model is a probabilistic language model that predicts the next token in a sequence by learning the probabilities of token occurrences in the training data and selecting the most likely subsequent token. In this assignment, I implement an N-gram probabilistic language model to assist with code completion in Java systems. Specifically, I downloaded numerous Java repositories using GitHub Search, tokenize the source code using Javalang, preprocess the data, train the N-gram model by recording token sequence probabilities, and perform predictions on incomplete code snippets. Finally, I evaluate the model's performance using accuracy metrics. The source code for this work can be found at **mygithubrepo.com**.

2 Implementation

2.1 Dataset Preparation

GitHub Repository Selection:

To build a dataset for training the model, I sourced Java repositories from GitHub using the GitHub Search tool, and then applied filters to ensure high-quality repositories, selecting only those that met the following criteria:

Language: Java, Minimum commits: 30, Minimum contributors: 15, Maximum Issues: 10, Minimum stars: 150.

Then I randomly selected 100 projects from the retrieved repositories and extracted Java methods for further processing. The extraction process focused on isolating relevant methods while eliminating boilerplate or redundant code.

Data Cleaning & Tokenization

The extracted methods underwent preprocessing to ensure consistency and relevance. I removed comments, reformatted the code, and standardized identifiers where necessary. For tokenization, I used a Python package specialized in Java source code parsing (javalang) to break down methods into meaningful tokens.

Dataset Splitting

The cleaned dataset included 1894369 methods, and was divided into training, validation, and test sets:

- **Training Set:** 80% of the methods; 1217417
- **Validation Set:** 10%; 260875
- **Test Set:** 10%; 260876

Additionally, I generated a vocabulary consisting of 788864 unique tokens from the combined datasets, ensuring minimal occurrence of unknown tokens during model training.

2.2 Train, Test, Evaluation

Model Training & Evaluation

I trained multiple N-gram models with different context window sizes (n = 3, 5, and 9) to determine the optimal configuration for Java code completion. Perplexity was used as the primary evaluation metric, as lower perplexity indicates better predictive accuracy.

The 9-gram model emerged as the best-performing configuration, achieving the lowest perplexity value of **3.3087** on the validation set. I then used this model to generate token predictions for the test set and report results for 100 sample predictions.

Model Testing

Using the selected 9-gram model, I evaluated its performance on the full test set, analyzing prediction accuracy and perplexity. The model achieved a perplexity score of **3.2196**, confirming its efficiency in predicting Java code tokens. To further evaluate the trained model, I implemented a token prediction script (predict_next_token.py) that takes a given context and predicts the next likely token based on the learned probabilities. This script allows for qualitative assessment by manually testing predictions on real Java code snippets. The script will choose a common Java keyword as a backup if the given context is not found in the training data. This tool shows how the N-gram model works in real-time code completion.

Instructor-Provided Corpus Evaluation

To further validate the model, I trained and evaluated it on a separate dataset provided by Professor Mastropaolo. The best-performing model for this dataset was the 9-gram configuration, which yielded perplexity values of **1.3781** on the validation set and **1.3720** on the test set. Like previous experiments, I generated predictions for this dataset and analyzed the model's performance in detail.

3 Conclusion

This project successfully implemented an N-gram-based approach to Java code completion and demonstrated that higher-order N-gram models improve predictive accuracy. By training and evaluating models with different context window sizes, I identified the 9-gram model as the best-performing configuration, achieving the lowest perplexity values on both my dataset and the instructor-provided corpus. These results highlight the effectiveness of statistical language modeling for code completion tasks.

While the model performed well, there are still areas for improvement. One potential next step is experimenting with advanced smoothing techniques to further refine token probability estimations. Using deep learning methods like transformer models could also boost accuracy by recognizing more complex patterns in the code. Additionally, broadening the dataset to include various repositories and coding styles could help improve generalization.