GenAI for Software Development: Assignment 1

Ted Tran tmtran03@wm.edu
Quang Hoang qhoang@wm.edu
Justin Liu jliu48@wm.edu

1 Introduction

Java code completion aims to automatically complete the code for methods or classes. The N-gram is a language model that can predict the next token in a sequence by learning the probabilities of token sequences based on their occurrences in the training data and choosing the token with the highest probability to follow. In this assignment, we implement an N-gram probabilistic language model to assist with code completion in Java systems. Specifically, we download numerous Java repositories using GitHub Search, tokenize the source code using Javalang, preprocess the data, train the N-gram model by recording the probabilities of each sequence, and perform predictions on incomplete code snippets. Finally, we evaluate the model's performance using accuracy metrics. The source code for our work can be found at **https://github.com/theantigone/ngram-java-ai**

2 Implementation

2.1 Dataset Preparation

GitHub Repository Selection. We start by using the GitHub Search tool (https://seart-ghs.si.usi.ch/) to compile a list of well-developed repositories, applying the following filters: language="Java", minimum number of commits=100. These criteria yield a total of 41,093 repositories. This ensures a high-quality, manageable corpus focused on method-level Java code. We clone and extract 8000 Java methods from 2515 Java classes using the following process: we beautified the methods and placed them into a Google sheet. After obtaining 10328 methods, we converted the Excel sheet of all the methods into a singular text file.

Cleaning: We make sure to include only relevant Java methods that are given by the application of the: following polishing criteria. Firstly, we remove duplicate methods. We eliminated near-identical methods (Type-1 clones) except for comments. Next, we kept only methods containing ASCII characters. Then, we removed outliers by discarding methods whose lengths fall outside the 5th–95th percentile. After that, we remove methods with common boilerplate patterns such as getters and setters. Lastly, we removed inline and block comments out of methods. To ensure we filtered the data properly, we printed the dataset size at each step to ensure quality control.

Code Tokenization: We utilize the **Pygments** (specifically **JavaLexer**) Python package to tokenize the extracted Java methods

Dataset Splitting: To create the training, test, and eval set, we randomly select 7561 methods from the cleaned corpus. Then we sample 80% of methods representing the training set and the remaining 20% is further split into 10% + 10% respectively test and eval.

Vocabulary Generation: We generate a vocabulary that contains 22520 number of code tokens. The vocabulary is generated considering training + evaluation + test set. In this way, we avoid unknown

tokens.

Model Training & Evaluation: We train multiple N-gram models with varying context window sizes to assess their performance. Initially, we experiment with n = 3, n = 5, and n = 9. To evaluate the impact of different N-values, we use perplexity as our primary metric, where lower values indicate better performance. After evaluating the models on the validation set, we select n = 3 as the best-performing model, as it achieves the lowest perplexity of around 176.664.

Model Testing: Using the selected 3-gram model, we generate predictions for the entire test set. However, for ease of analysis, we report only the first 100 predictions. In addition to generating predictions (a brief example is provided below), we also compute perplexity over the full test set. Our 3-gram model achieves a perplexity of 196.650 on this dataset.

Training, Evaluation, and Testing on the Instructor-Provided Corpus: Finally, we repeat the training, evaluation, and testing process using the training corpus provided by Prof. Mastropaolo. In this case, the best-performing model corresponds to n = 3, yielding perplexity values of around 25.979 on the validation set and around 10,887.294 on the test set. As in the previous case, we conclude by generating predictions for the test set, following the same methodology.