Rachel Zheng
rzheng02@wm.edu

Gen AI for Software Development: Assignment 1

1 Introduction

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, preprocess the data, tokenize the source code, 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/rachelzheng03/NGramModel.

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", license = MIT, minimum number of commits = 50, last commit after June 1, 2024, minimum number of stars = 1000, and minimum number of code lines = 1000. These criteria yield a total of 154 repositories. From this collection, we select 6 repositories for further analysis. We clone and extract 442,785 Java methods using the PyDriller code provided by Prof. Mastropaolo.

Cleaning: We make sure to include only relevant Java methods by first removing duplicate methods, methods with non-ASCII characters, and boilerplate methods (ie. getters and setters). We notice that some of the extracted methods include multiple methods, so we filtered those out as well. Then, we removed comments from the methods since we are focusing solely on code generation. Finally, we filtered out outliers, specifically the methods that fell below the 5th percentile or above the 95th percentile in terms of length.

Code Tokenization: We utilize the Pygments Python package to tokenize the extracted Java methods. To make our model more generalizable, we replace function names with "function" and parameter names with "argi" where i denotes the ith parameter of the Java method.

Dataset Splitting: We split the cleaned corpus into training, test, and eval set. To do this we sample 80%, 10%, and 10% of the methods for the training, test, and eval set respectively.

2.2 Model Training, Evaluation, and Testing

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 = 7. 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 172.165.

Model Testing: Using the selected trigram model, we generate predictions for 100 randomly selected methods from the test set, starting with the first 3 tokens of each method. In addition to generating predictions, we also compute perplexity over the full test set. Our trigram model achieves a perplexity of 184.772 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 6939.486 on the validation set and 7379.199 on the test set. As in the previous case, we conclude by generating predictions for the test set, following the same methodology.