GenAI for Software Development: Assignment 1

Rachel Huang (rhuang03@wm.edu), Jackson Taylor (jttaylor01@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/rrachelhuangg/code-generation-ngram-model.

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=25, minimum number of contributors=15, and minimum number of stars=150. These criteria yield a total of 7704 repositories. From this collection, we randomly select 100 repositories for further analysis. We clone and extract 26203 Java methods from 53 Java classes using the following process: we use the PyDriller code given to us in class to download the Java method code from the Github Search results csv file, and then we write the extracted methods to separate files in the extracted directory. This extracted directory with the extracted methods is the input for the subsequent cleaning step.

Cleaning: We make sure to include only relevant Java methods that are given by the application of the following polishing criteria: no duplicate methods, methods that only contain ASCII characters, methods that are not outliers based on their length, and no boilerplate methods. Though the Tokenizer code that we were given in class had cleaning methods to remove multiline and single line comments, we applied additional regex to clean out remaining multiline and single line comments that we discovered in the initial cleaned data.

Code Tokenization: We utilize the Pygments Python package and the Tokenizing code given to us in class to tokenize the extracted Java methods. We write each tokenized method to a separate line in our output file, so that our subsequent step of dataset splitting can use the new line characters to distinguish between each tokenized method.

Dataset Splitting: To create the training, test and eval set, we first shuffle the order of the methods from the cleaned corpus. Although it is standard to split the data along an 80-10-10 split for the training, test, and eval sets respectively, as per the project description, the first 100 methods became the test set and the rest of the methods were split 80-20 into the train and validation sets respectively. This partition was copied for all the following tests..

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, n = 7, 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 = 5 as the best-performing model, as it achieves the lowest perplexity of 1.11388. Evaluating the same model using the test set of 100 methods, it achieves a worse perplexity of 1.15704, then we generate predictions for the test set by randomly selecting a method from the test set and giving an n-1 window as input to the model for it to predict until 1 of 2 heuristics are satisfied: either a closing bracket ("}") brings the number of open brackets to 0, or 100 tokens are generated.

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 = 9, yielding perplexity values of 1.32322 on the validation set and 1.28262 on the test set. As in the previous case, we conclude by generating predictions for the test set, following the same methodology.