Lab 1 Report

Team members: Joshua Vasquez, Cathay Gao

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/cathieG/N_gram.git

2 Implementation

2.1 Dataset Preparation

GitHub Repository Selection. We start by using the GitHub Search tool (hQps://seart-ghs. si.usi.ch/) to compile a list of well-developed repositories, applying the following filters: language="Java", minimum saves = 1000, no more than 150 commits, exclude all forks. These criteria yielded a total of 10,949 repositories. From this collection, we randomly select 150 repositories for further analysis. We clone and extract 504,358 Java methods from over 100 Java classes using the following process: We first read in the csv file containing all the github repos. We then iterate over the column, "name", so that it has the correct git hub address. We then use Pydriller and javalang to access the github repos and extract the Java methods from them, putting them in a different csv file. Cleaning: We made sure to include only relevant Java methods by utilizing the Professor's preprocessing code for Lab0, modifying it so that it would work with our dataset. This got rid of the three types of duplicates. Code Tokenization: We utilized the pygments.lexer Python package to tokenize the extracted Java methods Dataset. To create the training, test and eval set, we randomly selected 80,000 methods from the cleaned corpus (due to processing concerns). We sampled 80% of methods representing the training set and the remaining 20% were further split into 10% + 10% for evaluation and testing, respectively. Vocabulary Generation: We generate a vocabulary that contains 192,671 numbers 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, n = 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 4907.94305106891. 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 87491.861084906 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 4814.692250029782 on the validation set and 496.2375889930067. As in the previous case, we conclude by generating predictions for the test set, following the same methodology.

Sample Output: