GenAI for Software Development: Assignment 1

Ella Bryant
egbryant01@wm.edu

1 Introduction

The Java Code Recommender via N gram models sought to automatically complete the code for Java methods. This is done through the usage of an N-gram style language model. The N-gram language model learns the probability of occurrences from a training data set and chooses the proceeding token with the highest probability. Specifically, this was completed by pulling numerous Java repositories using GitHub, preprocessing the data, tokenizing the data, training the N-gram model through the sequence frequencies, and performing predictions on incomplete Java code. Finally, evaluations of the model's performance were completed using accuracy metrics and the perplexity measure. The source code for our work can be found at ViolettGee/Java-Code-Recommender-via-N-gram-models-: The project will contain a Generative AI model for recommending code fragments via various N-gram models.

2 Dataset Preparation

GitHub Repository Selection. We start by using the GitHub Search tool (Search), using the metric "lang:Java". This metric produced 13.4 million repositories, and from this selection about 200 repositories were selected for further analysis. Within these repositories, the folders containing the Java files were found, and the file paths were recorded into "Assignment1Data.csv". Using the data within that file, "Data_Collection.py" automates the export of each of the java files from each of these list folder paths saving each to the "Raw_Data" folder. This uses the Python Request module to avoid importing the entire repositories and conserving data size. There were 8190 Java classes extracted using this process.

Preprocessing. First, we process each of text files within the "Raw_Data" folder containing each of the Java classes. Using the Python javalang module, the methods from each of the files are separated and added to a pandas data frame, leaving initially 57,840 Java methods. These methods are then filtered removing any exact copies, those that are not ASCII characters, removing methods that are too large or small, removing comments from the methods, and removing boilerplate code. The preprocessing then left 27,744 Java methods that are saved to "preprocessed_data.csv".

Tokenization: The preprocessed data is then tokenized using the Python javalang module and stored in "tokenized_data.csv".

3 Model Implementation

Dataset Splitting. To create the training and test set, we selected them in sequences of 5 where the first 4 where added to the training set and the 5^{th} was added to the test set. Thus, splitting 80% of the methods (22,195) into the training set and 20% of the methods (5,549) into the test set.

General Implementation. When each model is initializing, the model is created based on a context window size. The model initializes a dictionary that tracks the occurrences of that size sequence in the training data. More specifically, the dictionary keeps track of each sequence of one less than the context window size and creates an internal dictionary tracking the frequency of tokens after that sequence. Once initialized, this dictionary is used to calculate the most probable proceeding token and concatenated together to create a block of Java code.

Model Training and Evaluation. To find the most optimal context window size for the training data, context window sizes (n = 1,…,7) were iterated through. The performance of each was evaluated using the logarithmic perplexity metric, where the lower values indicate better performance. After evaluating the models, n=7 was selected as the best-performing model, as it achieves the lowest perplexity of 1.116688E+6308.

4 Model Evaluation

Model Testing. Using the selected 7-gram model, we generated prediction for the entire test set. However, for ease of analysis, we report only the first 100 predictions stored in the "validatation_set.csv". In addition, we also computed the average accuracy on the test set, which was 0.8967. Furthermore, we also compute the perplexity over the full test set. Our 7-gram model achieves a perplexity of 2.043337E+2212 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 Professor Mastropaolo. In this case, the best performing model corresponds to n = 1, yielding the lowest perplexity of 2.251479E+4876. The average accuracy was computed as 0.9445 and the perplexity on the test set was computed as 1.293548E+291.