Assignment 1: Code Completion with N-Grams

By Chuntian Chi, Matthew Chen, and Kate Balint

Introduction

In this project, we used a dataset with 469,779 Java methods located on Hugging Face from the paper Evaluating the State of Semantic Code Search. The code was obtained by a search on libraries.io for open-source GitHub repositories with valid licenses, specifically projects used or referenced by at least one other project and that were not forked in order to avoid duplicates. The repositories were sorted in order of popularity, defined by the number of stars and forks from other users. The data was processed with method granularity, so each entry of the data is a single method.

The methods were further filtered to ensure that they were representative of what programmers created. Methods that were shorter than three lines, test methods, constructors, or extension methods like toString or __str__ were omitted since they are likely not substantive, informative, or fully implemented. Duplicates were also removed to prevent copied or auto-generated methods from biasing the model. As outlined in The Adverse Effects of Code Duplication in Machine Learning Models of Code, methods were considered duplicates if the sets of their tokens and the sets of their literals and identifiers had a Jaccard similarity of greater than 0.8 and 0.7, respectively. Test methods were identified simply by whether they had the word "test" in their declaration.

We used 26 thousand methods in total since that is enough for our task and extracted them in the form of a CSV file. We approached the problem of generating code from this data by first tokenizing each method in our training corpus. Our model is then trained using these methods, adding each token to the model's dictionary of token embeddings and noting a window of the tokens before and after to provide context. Using the first few tokens from each method in the test corpus, the model generates code by choosing tokens from its dictionary based on both context and calculated probabilities of each token from the training corpus. Once the model has generated output based on an excerpt of code from the test corpus, the model's accuracy is calculated based on token matching with the complete code from the test data.

We implemented the N-gram model with K-smoothing for probability estimation, and handled unseen tokens with a special <unk> token. We built the vocabulary and counts of N-gram occurrences during training. Perplexity was calculated using K-smoothed probabilities and used as a measure of model performance to evaluate how well it predicted the test corpus.

Hyperparameter Tuning

We split our dataset into train (80%) and test (20%) data. We evaluated our models' performance using perplexity and accuracy of token matches between our generated codes and the test data. Our best data (with hyperparameters explained below) achieves perplexity around 235 on the test corpus. To accelerate the process of generating outputs for the purpose of submission, we only generate completions for the first 100 test cases in our submitted code.

Tokenizers: We experimented with different tokenizers (GPT2, LLaMA, CodeBERT and Javalang) to preprocess the raw Java codes. We chose Javalang as the final tokenizer for this assignment. Hugging face models (GPT2, LLaMA, CodeBERT) are well-known pre-trained language models, but they are general-purpose tokenizers and do not capture every nuance of Java syntax really well, while Javalang is a Java-specific library that provides an easy way to tokenize and parse Java codes. It was designed specifically for Java codes and handles Java language structures well, such as keywords, identifiers etc. We experimented with all 4 tokenizers on our model, and the generated codes suggest Javalang provides the best performance. As our assignment language is fixed, we decided to use Javalang.

N-Gram model: We chose $n = 2$ as our default parameter for the n-gram model. We tested from $n = 1$ to $n = 5$. A trigram model results in the best training performance (~99 training perplexity), but has relatively poor testing performance (~ 435 testing perplexity), indicating a model overfit. So we decide to use a smaller n value $n = 2$ as our parameter (~ 106 training perplexity, ~ 235 testing perplexity). We decide to use a bigram model as it is balanced on model complexity and performance (unigram model captures very short dependencies which is not enough for more complex sequences, while higher order models increase model complexity and sparsity).

K-smoothing: We used add-k smoothing and chose $k = 0.01$ as our default parameter. We tested k values ranging from $k = 0.01$ to $k = 5$, eliminating k values that are too big. Among all the values we tested, $k = 0.01$ yields the best results. We think that this is because with a very big k value, probabilities are spreaded to evenly across all possible n-grams and hence no meaningful predictions were made.