GenAI for Software Development: Assignment 1

Whitt Sanders
wjsanders@wm.edu

Kayla Anderson
kganderson01@wm.edu

Arya Shryock
ijrayshryock@wm.edu

1 Introduction

N-gram Recommender aims to automatically complete the code for a Java method or class. 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 pygments, 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 performance of the model using accuracy metrics. The source code for our work can be found here.

2 Implementation

2.1 Dataset Preparation

Dataset Creation Process: Starting with SEART's github repository search, we found well-developed Java repositories. We specified repositories with a minimum of 1000 nonblank lines, at least 100 stars, 10 collaborators, and at least one commit since 2023 to ensure some basic repository quality standards. We then limited the maximum code lines to 10000 to ensure that there were no massive repositories that overshadowed and diluted the method pool. The search returned approximately 1300 repositories, which we pared down to 18 repositories of medium to small size, processed them with pydriller, and ended with a single csv file containing approximately 300,000 processed methods for cleaning.

Cleaning: We make sure to include only relevant Java methods that are given by removing duplicate methods, filtering ascii methods, removing outliers in length, removing boilerplate methods, removing comments, and adding start/end tokens. This leaves us with about 30,000 methods to train the model.

Code Tokenization: We utilize pygments to tokenize the extracted Java methods.

Dataset Splitting: To create the training, testing and validation sets, we randomly select about 6000 methods from the cleaned corpus. The we sample 80% of methods representing the training set and the remaining 20% is further split into 10% + 10% respectively test and validation sets.

2.2 Model Training Methodology

Model Training & Evaluation: We train multiple N-gram models with context window sizes from $n = 2$ to $n = 10$ to assess their performance. 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 test set, we select $n = 4$ as the best-performing model, as it achieves the lowest perplexity of 8.2803.

Model Testing: Using the selected 4-gram model, we generate predictions for the entire validation set. However, for ease of analysis, we report only 100 randomly-selected predictions. In addition to generating predictions (a brief example is provided below), we also compute perplexity over the full test set. Our 4-gram model achieves a perplexity of 8.9987 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 (found in training.txt). In this case, the best-performing model corresponds to $n = 2$, yielding perplexity values of 1486377.5392 on the validation set and 1462705.8934 on the test set. As in the previous case, we conclude by generating predictions for the test set, following the same methodology.

2.3 Model Training and Evaluation Explanation

Ngram Model Creation Example: The n-gram model is creating by looping over the tokens in the corpus. The grams are stored in a dictionary, with the $n-1$ tokens acting as the key. The preceding possible token is then appended. For example a trigram with the first two tokens as 'OperationStatus' and '(';

```
('OperationStatus', '('): ['true', 'true', 'false', 'false',
'true', 'true', 'true', 'true', 'true'],)
```

The most probable word given $n-1$ tokens is found by using the $n-1$ tokens as a key to the dictionary, picking the word that appears the most. If the key isn't found it returns NA with a zero probablity. Chaining the most probable words is how code prediction is done. Here is an example using a trigram model as the same tokens as before; limiting the length to 5 more tokens, displaying its associated probability.

```
('true', 0.7777777777777778)
(')', 0.6111111111111112)
(';', 0.8095238095238095)
('\text{\n}', 0.998810232004759)
('\}', 0.5163164953062137)
['OperationStatus', '(', 'true', ')', ';', '\text{\n}', '\}']
```

Evaluation: Perplexity is calculated by taking the first $n-1$ tokens in each given test method and finding the probability for the n^{th} token to be next. This process continues for the rest of the method and the $\log_2$ of the probabilities are summed. If a token has 0 probability from the trained model $\log_2(1^{-10})$ is instead added to the sum to prevent undefined values from affecting the calculations. The summed probability is then divided by the total number of tokens to give the average log probability. 2 is then raised to the power of the negative log probability to get the perplexity. The perplexity of every method in the test set is then averaged together to get the final reported perplexity value.