CSCI 680 AI for Software Engineering: Assignment 1

Enze Xu
exu03@wm.edu

Yi Lin
ylin13@wm.edu

1 Introduction

Code completion in Java aims to automatically complete the code for Java 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. This can be formulated as:

$$P(t_n \mid \ldots, t_{n-2}, t_{n-1}) \approx P(t_n \mid t_{n-(N-1)}, \ldots, t_{n-2}, t_{n-1}) = \frac{\text{Count}(t_{n-(N-1)}, \ldots, t_{n-1}, t_n)}{\text{Count}(t_{n-(N-1)}, \ldots, t_{n-1})} \quad (1)$$

where t_n and t_{n-1} are sequences of code tokens, N is a hyperparameter in the N-gram model, defining the length of the context window.

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/EnzeXu/CSCI680_N_Gram.

2 Implementation

2.1 Dataset Preparation

GitHub Repository Selection. We begin 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=50, minimum number of contributors=5, minimum number of branches=5, and minimum number of stars=5. These criteria yield a total of 14,364 repositories. From this collection, we randomly select 100 repositories for further analysis.

Repository Cloning and Preprocessing. Using Python's **subprocess** package, we clone the selected repositories from GitHub. We then filter out any files that did not have a ".java" extension. The remaining Java files are renamed, and we apply the **re** package to strip out comments and blank lines. After preprocessing, we determine that the 100 repositories contain 26,857 Java classes and 312,971 Java functions.

Code Tokenization. We utilize the **javalang** Python package to tokenize the Java files in both training and test sets. This package allows us to extract tokens by interpreting the syntactic structure of the code so that some words like "<int>" can be split into "<", "int", and ">".

Dataset Splitting. To create the test set, we randomly select 100 classes from the dataset, which remain fixed for all experiments described in the results section. Given that our dataset exceeds the required size, we vary the training set size (in terms of the number of Java classes) across 125, 250, 500, 1,000, 2,000, 4,000, 8,000, and 16,000 classes to evaluate the effect of training set size on N-gram model performance in terms of average precision.

2.2 Vocabulary Generation

For each Java class, we generated both N-length and (N-1)-length tuples of sequences by sliding a window across the code, from the beginning to the end of each statement. The generated tuples are stored as the keys of two dictionaries: self.n_grams and self.n_minus_1_grams, respectively, with

their corresponding values being the frequencies of each sequence tuple in the training set. Additionally, we recorded which tokens are likely to follow a specific (N-1)-length sequence in the dictionary: self.n_minus_1_grams_next_available. These three dictionaries together form our vocabulary.

2.3 Prediction Method

To predict the next token, we strictly adhere to the formula provided in Eq. 1 to select the most frequent token. In the implementation, having already created the necessary dictionaries in the previous section, we first iterate over all possible next tokens in the available list. Each token is combined with the previous (N-1) tokens to form an N-length sequence. We then compare the count of each N-length sequence, and select the token with the highest frequency as the predicted next token.

3 Results

To evaluate the effect of training set size on the performance of an N-gram model in terms of average precision, we generate training datasets of varying sizes (in terms of the number of Java classes): 125, 250, 500, 1,000, 2,000, 4,000, 8,000, and 16,000. The test dataset is fixed at 100 Java classes for all experiments. Table 1 presents the results, where N denotes that $N-1$ previous tokens are used by the N-gram model to predict the next token. The best-performing N for each training set size is highlighted in bold. As shown in the results, model precision improves as the training set size increases. Using the largest training set, our model achieves its highest precision of 0.6544 at N=9. We also present a heatmap (Fig. 1) to more effectively convey the distribution and trends in our experimental results.

	Number of JAVA classes in Train Set							
N	125	250	500	1,000	2,000	4,000	8,000	16,000
2	**0.2954**	0.3069	0.3225	0.3050	0.3418	0.3264	0.3547	0.3746
3	0.2901	**0.3093**	**0.3364**	**0.3662**	**0.3791**	**0.4025**	0.4209	0.4962
4	0.2460	0.2722	0.3011	0.3418	0.3596	0.3972	**0.4462**	0.5838
5	0.1919	0.2206	0.2518	0.2922	0.3125	0.3575	0.4386	0.6216
6	0.1592	0.1829	0.2098	0.2498	0.2690	0.3142	0.4178	0.6429
7	0.1344	0.1552	0.1781	0.2147	0.2355	0.2752	0.4002	0.6525
8	0.1185	0.1362	0.1577	0.1881	0.2081	0.2430	0.3848	**0.6544**
9	0.1085	0.1246	0.1428	0.1691	0.1884	0.2191	0.3717	0.6528
10	0.1021	0.1165	0.1321	0.1552	0.1727	0.2013	0.3605	0.6488

Table 1: Average precision of N-gram models with varying training set sizes (number of Java classes). The best-performing N for each training set size is highlighted in bold.

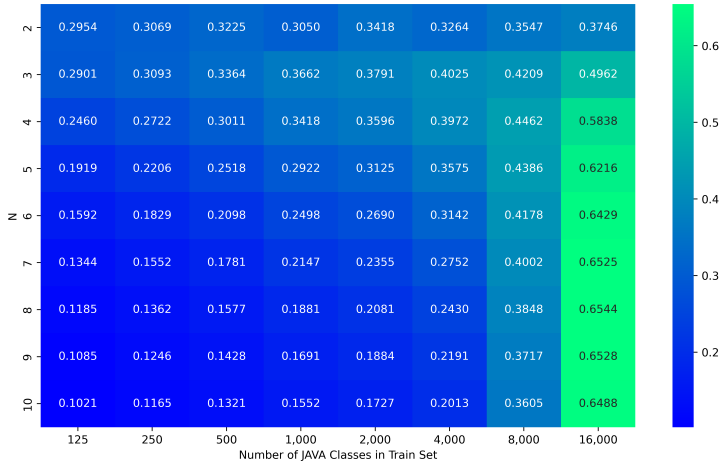

Figure 1: Precision heatmap with different number of JAVA classes in the training set (x-axis) and different sequence length N (y-axis). The color gradient represents the precision, with higher values in green and lower values in blue.