Recommending Code Tokens via N-gram Models

Yibarek Tadesse
Course: GenAI for Software Development (CSCI 455/555)
Instructor: Dr. Antonio Mastropaolo

February 2026

Abstract

This report details the implementation of a probabilistic N-gram language model designed for Java method code completion. The project pipeline involves mining a large-scale corpus of Java repositories from GitHub, robust preprocessing and tokenization using `javalang`, and training an Add-α smoothed N-gram model from scratch. The model was evaluated using perplexity across varying context windows ($n \in \{3, 5, 7\}$) and training subset sizes (capped at 15k, 25k, and 35k instances). The best-performing model achieved a perplexity of 1210.13 on the self-created test set and 1864.57 on the instructor-provided test set.

1 Dataset Construction (MSR Procedures)

To ensure the model learns from high-quality, practical code, I constructed a custom training corpus by mining open-source GitHub repositories.

1.1 Repository Selection & Mining

I used the GitHub API to fetch 500 top Java repositories matching specific quality criteria:

- **Popularity:** > 1000 stars to ensure well-maintained code.
- **Size:** > 10 MB to guarantee sufficient code volume.
- **Activity:** New activity pushed after Jan 1, 2025 to ensure code represents modern day coding practices.
- **Exclusions:** Forks were excluded to prevent data duplication, and competitive programming topics (like LeetCode) were filtered out to avoid isolated, non-production code styles.

Out of the 500 repositories targeted, 499 were successfully cloned locally. I then isolated Java files and randomly selected up to 10 java files from each repository, explicitly excluding directories named `test`, `example`, or `demo` for example to ensure the model trained strictly on application logic. This process resulted in the extraction of 4810 java files.

Listing 1: Snippet highlighting the strict GitHub API filtering criteria

```
params = {
    "q": "language:java␣stars:>1000␣size:>10000␣pushed
        :>2025-01-01␣archived:false␣-topic:leetcode␣-topic:
        interview",
    "sort": "stars",
    "order": "desc",
    "per_page": 100
}
```

2 Preprocessing & Tokenization

2.1 Method Extraction and Filtering

Using the `javalang` library, I then parsed the source code to extract method declarations. From the 4810 java files, there were 39866 methods found. However, some more work needed to be down filter out methods that were not To clean this data, I applied several rigorous filters:

1. **Non-ASCII Filter:** Dropped 1181 methods containing non-ASCII characters to ensure clean tokenization.
2. **Length Filter:** Removed 2186 methods with fewer than 10 tokens, as they lack sufficient context for meaningful N-gram sequences.
3. **Malformed & Duplicate Removal:** Filtered out 2557 methods with multiple signatures on a single line and incomplete methods. Also, 1468 duplicate methods were removed, resulting in a final dataset of 32474 clean methods.

2.2 Dataset Splits & Vocabulary

The dataset was randomly shuffled and split into the following subsets:

- **T1:** 15,000 methods
- **T2:** 25,000 methods
- **T3:** 30474 methods
- **Validation / Test:** 1,000 methods each.

For each training set, a unique vocabulary was built from scratch. Any token in the validation or test sets not present in the respective training vocabulary was mapped to the `<UNK>` token to strictly prevent data leakage.

3 Model Design and Training

The core of this project is to design a `NGramLanguageModel` that would calculate the conditional probability of the next token given the previous $n-1$ tokens.

To handle unseen token sequences, I implemented **Add-α smoothing** (with $\alpha = 0.1$) alongside a **Unigram Backoff** strategy. If a specific N-gram context has never been observed, the model would then fall back to predicting the most frequent unigram from the training distribution.

The models were trained across $n \in \{3, 5, 7\}$ for all three dataset sizes (T1, T2, T3) to allow for hyperparameter tuning via validation perplexity.

Listing 2: Computes the smoothed probability P(target | context).

```
def get_probability(self, context, target):

    count_ngram = self.context_counts[context][target]
    count_context = self.context_totals[context]

    # Add-alpha smoothing formula
    prob = (count_ngram + self.alpha) / (count_context +
        self.alpha * self.vocab_size)
    return prob
```

4 Evaluation and Results

The models were evaluated using perplexity, which measures the model's uncertainty when predicting the ground-truth token. A lower perplexity indicates that the model assigns a higher probability to the actual sequence of tokens, reflecting better predictive performance and a higher degree of confidence. The perplexity is calculated as:

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1...w_{i-1})}} \tag{1}$$

When calculating this metric, I used the probability assigned to the actual ground-truth token in the sequence at each step, rather than the model's argmax prediction.

After evaluating the combinations of context windows ($n \in \{3, 5, 7\}$) and training data scales (T1, T2, T3) against the validation set, the best-performing configuration was determined by the lowest validation perplexity. The optimal configuration was the T3 training set (capped at 35,000 instances) utilizing a 3-gram context window. This configuration was then applied to the final testing environments.

Final Test Perplexities:

- **Self-Created Test Set (`test.txt`):** 1210.13
- **Instructor-Provided Test Set (`given_test.txt`):** 1864.57

4.1 Error Analysis & Discussion

The significant perplexity gap between the self-created test set (1210.13) and the instructor-provided test set (1864.57) highlights the substantial impact of distribution shifts in software engineering datasets. Because the self-created set was drawn from the exact same GitHub repositories as the training data, the model had already learned the specific naming conventions, stylistic formatting, and domain-specific idioms of those codebases.

In contrast, the instructor-provided set likely originates from completely unseen projects featuring distinct variable names, proprietary library calls, and different structural conventions. When the model encounters these novel code elements, a higher frequency of tokens are forced into the `<UNK>` mapping or fall back to the generalized unigram distribution, resulting in lower confidence and higher overall perplexity.

Furthermore, while the 3-gram model performed best among the tested configurations, standard N-gram models are inherently limited by their narrow, localized context window. They rely strictly on the immediate preceding tokens and lack true semantic understanding of the source code. As a result, the model naturally struggles to capture long-range dependencies such as: matching closing parentheses, recalling variable type declarations from earlier in a method, or understanding complex nested conditional logic. This ultimately means that its predictive accuracy will be highly variable in real-world codebases.

5 Repository

For more information regarding the data implementations of the source code, visit the repo attached to this link `https://github.com/Yibarek1/n_gram_model`