CSCI 455/555: GenAI for Software Development

Assignment 1

Nathaniel Callabresi
Prof. Antonio Mastropaolo

February 20, 2026

Abstract

Predicting code tokens is a foundational task in automated software development, enabling tools such as intelligent code completion. This paper details the implementation of an end-to-end system for Java method-level token recommendation. We constructed a robust Mining Software Repositories (MSR) pipeline, extracting 80,567 cleaned Java methods from 700 high-quality GitHub repositories (selected based on a $> 4,000$ star threshold). We evaluated several N-gram configurations ($n \in \{3, 5, 7\}$) using Lidstone smoothing across cumulative training sets of increasing size. Our results indicate that the 3-gram model trained on 15,000 methods achieved the optimal performance with a validation perplexity of 39.78 and a provided test set perplexity of 35.38. Paradoxically, increasing both the context window and the training data volume led to higher perplexity, suggesting that the massive breadth of the dataset introduced a "vocabulary explosion" and stylistic inconsistency that reduced predictive confidence. This highlights that for probabilistic code models, data consistency is often more critical than sheer volume.

1 Introduction & Background

The modern software development lifecycle relies heavily on developer productivity tools, with code completion being one of the most impactful. By suggesting the next likely token in a sequence, these tools reduce cognitive load and minimize syntax errors.

1.1 Background on N-grams

An N-gram model is a probabilistic language model that predicts the probability of the next token t_n based on a fixed context window of the previous $n-1$ tokens. Formally, it approximates the probability:

$$P(t_n|t_{n-1}, t_{n-2}, ..., t_{n-(N-1)}) = \frac{\text{Count}(t_{n-(N-1)}, ..., t_{n-1}, t_n)}{\text{Count}(t_{n-(N-1)}, ..., t_{n-1})}$$

While simple, N-gram models are effective at capturing local syntactic patterns, making them a suitable baseline for code recommendation.

1.2 Objective

The objective of this project is to develop and evaluate a complete, end-to-end code recommendation system. This involves:

1. **Data Engineering:** Building an MSR pipeline to mine, clean, and tokenize a high-quality Java dataset from top-tier GitHub repositories.
2. **Model Implementation:** Training probabilistic N-gram models for $n \in \{3, 5, 7\}$ using advanced smoothing techniques.

3. **Empirical Evaluation:** Selecting the best configuration via validation perplexity and evaluating the final model against both provided and self-mined test sets.

4. **Analytic Insight:** Investigating how external factors like dataset scaling and stylistic breadth impact the predictability of source code.

2 Mining Software Repositories (MSR) & Dataset Construction

2.1 Data Collection

I fetched 700 Java repositories via the GitHub API. The selection criteria prioritized quality and uniqueness: I filtered for repositories with more than 4,000 stars and excluded all forks to prevent data duplication. To ensure a high degree of stylistic breadth, I sampled up to 20 Java files per repository rather than exhausting single projects.

2.2 Extraction & Preprocessing

Using the `javalang` library, I parsed the source files to isolate method-level declarations. To maintain a high-quality corpus, I applied strict filtering rules: removing non-ASCII characters, dropping methods with fewer than 10 or more than 512 tokens, and removing methods with fewer than 10 unique tokens. This pipeline yielded a final pool of 80,567 cleaned methods.

2.3 Tokenization

Raw methods were tokenized into space-separated strings. The tokenizer was configured to treat keywords, identifiers, literals, and punctuation (e.g., `;`, `.`, `{`, `}`) as distinct tokens to ensure the N-gram model could learn the structural grammar of Java.

2.4 Dataset Splits

I reserved 1,000 methods for a validation set (V) and 1,000 methods for a self-created test set. The training data was organized into three cumulative sets: T_1 (15,000 methods), T_2 (25,000 methods), and T_3 (35,000 methods). This design allowed us to observe how scaling the volume and breadth of training data impacts model performance.

3 Model Training Procedure

3.1 Vocabulary Handling

A separate vocabulary was built from scratch for each training set. I implemented a `<UNK>` cutoff of 3; any token appearing less than three times was mapped to an unknown token. This dynamically filters out obscure, project-specific identifiers, effectively mitigating the "vocabulary explosion" problem.

3.2 N-gram Configurations & Smoothing

Models were trained using the `nltk` library for $n \in \{3, 5, 7\}$. Because zero-probability N-grams cause division-by-zero errors when calculating perplexity, smoothing is mandatory. I utilized Lidstone smoothing with $\gamma = 0.01$. Standard Laplace smoothing was found to shift too much probability mass in a sparse code vocabulary; a smaller γ successfully handled zero-frequency N-grams without heavily penalizing known code sequences.

4 Evaluation & Results

4.1 Evaluation Metric

The primary metric used is perplexity, which represents the model's uncertainty when predicting the ground-truth next token. A lower perplexity indicates higher confidence.

4.2 Validation Phase Results

Table 1 summarizes the validation perplexity scores. The 3-gram model consistently outperformed larger context windows. Larger windows ($n = 5, 7$) suffered from severe data sparsity due to the high variability of identifiers across 700 repositories.

Training Set	3-gram	5-gram	7-gram
T_1 (15k)	**39.78**	213.76	837.97
T_2 (25k)	47.93	275.62	1090.87
T_3 (35k)	54.30	320.25	1273.79

Table 1: Validation Perplexity across cumulative training sets and N-gram orders.

4.3 The Scaling Paradox

The results revealed a "Scaling Paradox": perplexity worsened as the training set grew and as n increased. I hypothesize that this is due to a breadth-first mining approach. Adding more data from a massive mix of repositories introduced inconsistent naming conventions and varied coding styles, scattering the model's probability distribution and reducing its confidence.

4.4 Final Testing Phase

The winning configuration (T_1, $n = 3$) was evaluated against the two final test sets to verify robustness.

Test Set	Perplexity
Provided Test Set (`provided_test.txt`)	35.38
Self-Created Test Set (`test.txt`)	41.37

Table 2: Final Perplexity for the selected 3-gram model on independent test data.

5 Discussion & Analysis

5.1 Breadth vs. Depth

The strategy prioritized breadth by pulling methods from 700 different repositories. While this captures the general syntax of the Java language, it fails to learn the consistent "dialect" or specific patterns found within a single project. Probabilistic N-grams are highly sensitive to identifier distributions; as we incorporated more repositories, the unique token count (vocabulary size) grew exponentially relative to the observed structural patterns.

5.2 Data Consistency and Code Quality

The degradation in performance on larger training sets suggests that sheer data volume is not a solution for probabilistic models. The inconsistency in coding conventions across such a diverse range

of contributors introduces noise. A deeper mining approach, focusing on thousands of methods from a smaller set of high-quality repositories, would likely provide a more cohesive training signal and better predictability. Furthermore, because the repositories were sorted by stars, the smaller, initial subsets likely contained more standardized, well-documented, and thus more predictable code compared to the expanded sets.

6 Next Steps & Future Work

1. **Prioritizing Depth:** Mining 50 repositories but thousands of methods each to ensure stylistic consistency.
2. **Advanced Smoothing:** Experimenting with Kneser-Ney smoothing for sparse vocabularies.
3. **Structural Models:** Transitioning to models that understand Abstract Syntax Trees (ASTs) or architectures like Transformers (e.g., CodeBERT) that can better handle long-range dependencies and sparse vocabularies.

7 Conclusion

This study revealed that for probabilistic language models, dataset consistency and code quality are more valuable than sheer data volume. While I successfully trained a functional 3-gram model, the system highlighted the limitations of fixed context windows when faced with high stylistic variability across diverse repositories.

8 Repository & Links

The source code, README, and generated JSON result files are available at the following link: https://github.com/F4llow/GenAI_Assignment1. There are also extensive instructions on how to reproduce my findings. Additionally, an evaluate.py script is included in the repository, allowing users to evaluate the model on any tokenized test set via a command-line interface.