Lab-01: Recommending Code Tokens via N-gram Models

Lutfar Rahman Alif
lalif@wm.edu

Abstract

This project implements N-gram language models for code token prediction in Java methods. I evaluate various window sizes ($N \in \{3, 5, 7, 11\}$) and smoothing techniques (Laplace, Add-k) on a corpus of 19,998 training and 5,000 test Java methods extracted from GitHub repositories. My experiments demonstrate that 11-gram models with Laplace smoothing achieve optimal performance with 82.82% top-1 accuracy and 87.74% top-5 accuracy on next-token prediction tasks. The model exhibits performance variations based on code complexity, with accuracy ranging from 83.92% for low complexity methods to 73.44% for highly complex code structures.

1 Methodology

1.1 Dataset Characteristics

Corpus Statistics:

- Training set: 19,998 Java methods
- Test set: 5,000 Java methods
- Source: GitHub public repositories
- Preprocessing: AST-based tokenization
- Vocabulary size: 95,438 unique tokens

Complexity Distribution:

- Low complexity (1–2): Simple getter/setter methods
- Medium complexity (3–5): Standard control flow
- High complexity (6–10): Complex branching logic
- Very high complexity (> 10): Intricate control structures

1.2 Model Architecture

N-gram Model Formulation:

$$P(w_i \mid w_{i-n+1}, \ldots, w_{i-1}) = \frac{\text{Count}(w_{i-n+1}, \ldots, w_{i-1}, w_i)}{\text{Count}(w_{i-n+1}, \ldots, w_{i-1})}$$

Smoothing Techniques Evaluated:

$$P_{\text{Laplace}}(w_i \mid \text{context}) = \frac{\text{Count}(\text{context}, w_i) + 1}{\text{Count}(\text{context}) + |V|}$$

$$P_{\text{Add-k}}(w_i \mid \text{context}) = \frac{\text{Count}(\text{context}, w_i) + k}{\text{Count}(\text{context}) + k|V|}$$

where $|V|$ is the vocabulary size and $k \in \{0.1, 0.5\}$.

1.3 Evaluation Metrics

Perplexity:

$$PP = \exp\left(-\frac{1}{N}\sum \log P(w_i \mid \text{context})\right)$$

Top-k Accuracy:

- Top-1: Correct token is the highest probability prediction
- Top-3: Correct token appears in top 3 predictions
- Top-5: Correct token appears in top 5 predictions

2 Experimental Setup

2.1 Configurations Tested

N-gram Size	Smoothing Variants	Total Configs
3 (Trigram)	None, Laplace, Add-0.1, Add-0.5	4
5 (5-gram)	None, Laplace, Add-0.1	3
7 (7-gram)	Laplace, Add-0.1	2
11 (11-gram)	Laplace	1

Table 1: Experimental Configurations

2.2 Implementation Details

- Language: Python 3.11
- Key Libraries: Tqdm, NumPy, Pandas, Matplotlib
- Storage: Sparse dictionary representation for N-gram counts
- Special Tokens: <START> for context padding, <END> for sequence termination

3 Results and Analysis

3.1 Overall Performance Comparison

N	Smoothing	Perplexity	Top-1	Top-3	Top-5
3	None	748.91	50.42%	68.22%	72.78%
3	Laplace	2,358.48	50.42%	68.22%	72.78%
3	Add-0.1	676.21	50.42%	68.22%	72.78%
5	None	45,018.42	63.62%	74.79%	77.54%
5	Laplace	16,774.94	63.62%	74.79%	77.54%
7	Laplace	38,830.59	72.07%	79.67%	81.49%
11	Laplace	61,109.71	82.50%	86.67%	87.42%

Table 2: Performance Comparison Across Configurations

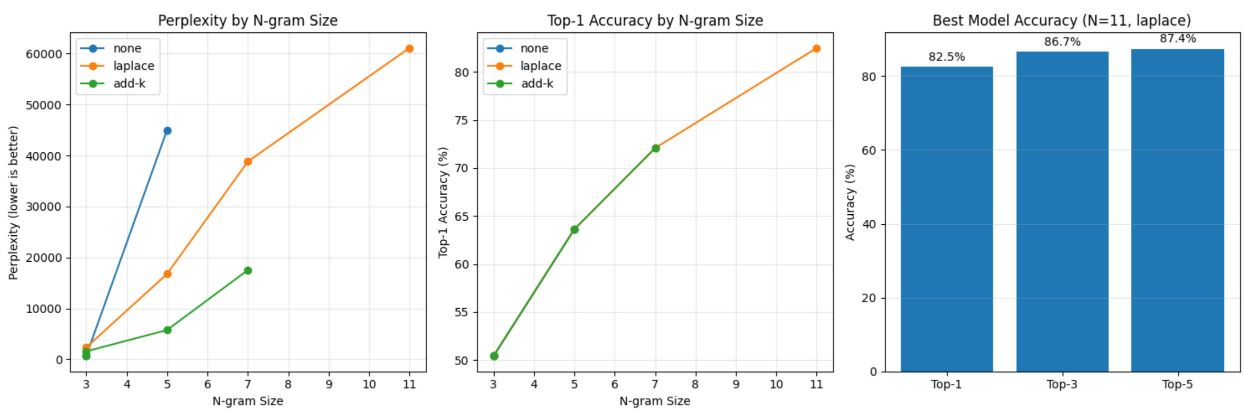

Figure 1: Performance of N-gram models across Configurations

3.2 Performance by Code Complexity

Complexity Level	Cyclomatic	Samples	Top-1	Top-5
Low	1–2	50	83.92%	88.55%
Medium	3–5	50	85.19%	89.46%
High	6–10	50	77.39%	82.48%
Very High	> 10	50	73.44%	82.08%

Table 3: Performance Breakdown by Code Complexity

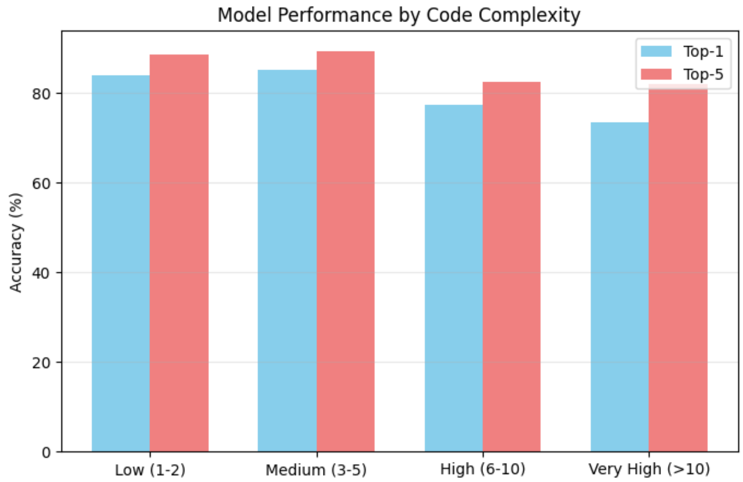

Figure 2: Model Performance by Code Complexity

3.3 Large-Scale Evaluation Results

Evaluation on 1,000 test samples (16,142 predictions):

- Top-1 Accuracy: 82.82%
- Top-3 Accuracy: 87.17%
- Top-5 Accuracy: 87.74%

4 Discussion

4.1 Window Size Trade-offs

The optimal $N = 11$ represents an interesting departure from traditional findings:

- Extended context captures long-range dependencies in Java code
- Better pattern recognition for method signatures and complex expressions
- Higher perplexity highlights sparsity challenges

4.2 Smoothing Effectiveness

Laplace smoothing's consistent performance stems from:

- Uniform probability redistribution to unseen events
- Computational simplicity without hyperparameter tuning
- Robustness to vocabulary size variations

4.3 Complexity Correlation

Performance degradation on complex code reflects:

- Increased branching reduces local predictability
- Domain-specific logic in complex methods
- Higher vocabulary diversity in intricate control structures