Using N-Gram Language Models to Predict Code Tokens

Author: James He

Course: CSCI 455 - Generative AI for Software Development

Professor: Antonio Mastropaolo

Date: Feburary 2026

Repository:
`https://github.com/James-Kai-He/CSCI455-GenAI_For_Software_Development/tree/main/A1`

Abstract

This report describes the design, training, and evaluation of N-gram language models for Java code token recommendation. Nine N-gram models were trained with three context sizes of $n \in 3, 5, 6$ using three test set sizes of $T1, T2, T3$ with smoothing performed. The corpus was mined using open-source GitHub repositories and hyperparameter tuning was performed on a validation set using Perplexity as the metric. The best configuration used the trigram model trained on T3 with alpha = 0.001. This achieved a validation of perplexity of 265.96, a test perplexity of 312.35, and a perplexity of 429.43 on the provided test set. Results show that larger training sets improve predictive ability and lower-order N-gram models with light smoothing outperform high-order models.

Data Mining and Corpus Construction

Repository Selection and Criteria

Java repositories were mined from GitHub using the GitHub Search API and then filtered by the following quality criteria to ensure the data reflects well maintained real world code. The methods used to pull and filter the data can be seen from here.

1. **Language Filter:** Only repositories that use Java as the primary language were selected
2. **Popularity:** Only repositories with more than ten stars were selected
3. **Activity:** Repositories with more than two years of inactivity were excluded.
4. **License:** Only repositories with open-source licenses (MIT, Apache 2.0, BSD) were used.
5. **Size Constraints:** Repositories with fewer than 5 Java source files or fewer than 500 lines of code were excluded.

Method-Level Extraction and Preprocessing

Each repository was cloned and processed through a preprocessing pipeline adapted from the provided lab. The methods used can be seen from here.

1. **Method extraction**: Java methods were isolated using a parser that targeted public, protected, and package-private method declarations. Anonymous methods and lambda expressions were excluded.
2. **Non-ASCII removal**: All characters that were non standard ASCII were removed

3. **Token-length filtering:** Methods with fewer than 10 tokens after tokenization were discarded, as they are too short to provide meaningful n-gram context.
4. **Deduplication:** Exact duplicate methods were removed. Similarly very similar duplicates were removed using Jaccard similarity with a threshold of 0.9
5. **Tokenization:** The Java methods were tokenized into space-separated tokens to ensure that keywords, identifiers, literals, operators, and punctuation symbols each appear as distinct tokens.

Dataset Splits

After preprocessing, the dataset was split into four sections.: Three cumulative training sets with sizes of 15,000 (T1), 25,000 (T2), and 35,000 (T3) method instances, a validation set of 1,000 instances , and a self-created test set of 1,000 instances. There is also a test set of 1,000 instances provided by the course.

N-gram Model Design and Training

Model Architecture and Vocabulary

An N-gram language model estimates the probability of the next token given the preceding $n-1$ tokens as context using maximum likelihood estimation from corpus counts:

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

A separate vocabulary was built for each training set (T1, T2, T3). Only tokens appearing at least once in the training set were included; all others were mapped to a `[UNK]` token during inference. Each method sequence was padded with $n-1$ start-of-sequence tokens `<s>` and $n-1$ end-of-sequence tokens `</s>` before n-gram extraction.

The three training sets yielded the following vocabulary sizes and unique n-gram type counts:

Training Set	Vocabulary Size	3-gram Types	5-gram Types	7-gram Types
T1	62,157	574,649	1,110,609	1,432,792
T2	89,931	873,946	1,746,926	2,299,868
T3	113,245	1,140,745	2,332,975	3,115,560

Table 1: Vocabulary sizes and unique n-gram type counts per training set.

Smoothing: Add-α

To prevent zero-probability n-grams, which cause undefined perplexity, add-α smoothing was used. The smoothed probability of an n-gram is:

$$P_\alpha(t_n \mid \text{context}) = \frac{\text{Count}(\text{context}, t_n) + \alpha}{\text{Count}(\text{context}) + \alpha \times |V|}$$

where $|V|$ is the vocabulary size of the respective training set. Five values were evaluated on the validation set with values $\alpha \in \{0.001, 0.01, 0.1, 0.5, 1.0\}$. The final α was selected as whichever value yielded the lowest validation perplexity. Because zero-probability n-grams cause division by zero when computing perplexity, smoothing is necessary to calculate perplexity.

The implementation is as follows:

```
def calc_ngram_prob(self, ngram, k=0):
    context = ngram[:-1]
    count_ngram = self.ngram_counts[ngram]
    count_context_ngram = self.ngram_counts[context]
    probability = (count_ngram + k) / (count_context_ngram + k * self.vocabulary_size)
    return probability
```

Evaluation and Results

Perplexity on the Validation Set

Lower perplexity indicates a more confident and better-calibrated model. Perplexity is computed using $P(w_i \mid \text{context})$—the probability the model assigns to the *ground-truth* next token, regardless of which token the model would actually predict. Configurations (3 sets $\times$ 3 n-values $\times$ 5 α-values) were evaluated on the validation set, and the results are summarized in the table below. The best configuration (lowest perplexity) is highlighted in bold.

	$\alpha = 0.001$	$\alpha = 0.01$	$\alpha = 0.1$	$\alpha = 0.5$	$\alpha = 1.0$
T1, $n = 3$	371.22	576.77	1264.57	2614.99	3672.73
T1, $n = 5$	2402.37	3872.98	7407.81	12376.61	15365.65
T1, $n = 7$	6409.95	9146.99	14134.31	19815.69	22849.23
T2, $n = 3$	308.36	519.49	1235.01	2700.13	3885.27
T2, $n = 5$	2086.87	3737.91	7955.99	14284.73	18255.64
T2, $n = 7$	6074.46	9577.67	16343.04	24411.44	28835.89
T3, $n = 3$	**265.96**	472.82	1186.06	2691.42	3936.02
T3, $n = 5$	1809.69	3500.32	8055.93	15247.15	19905.25
T3, $n = 7$	5650.04	9556.83	17490.14	27330.37	32851.45

Table 2: Validation set perplexity for all 45 configurations

The best configuration was **T3,** $n = 3$**,** $\alpha = 0.001$ with a validation perplexity of 265.96. This model was then evaluated on both test sets and achieved perplexities of 312.35 on the self-created test set and 429.43 on the instructor-provided test set.

The self-created test set has lower perplexity (312.35) than the instructor-provided set (429.43). This is expected because the self-created test corpus is from the same repository pool as the training data and uses the same preprocessing pipeline, so its vocabulary and token distributions align more closely with the training data. The instructor-provided set may have a different origin and preprocessing pipeline.

Output Files

Two JSON output files are created according to the assignment specification. Each file records the test set label, overall perplexity, and per-method prediction entries. For each method, the output lists the tokenized code, the context window size n, and per-position records containing: (i) the context tokens, (ii) the predicted next token (argmax), (iii) the predicted token probability, and (iv) the ground-truth token.

- **`results-provided.json`** — Predictions and perplexity for the instructor-provided test set.
- **`results-test.json`** — Predictions and perplexity for the self-created test set.

Error Analysis

There is a performance gap between the self created and provided test sets which can be partially explained with the following:

1. **Out-of-vocabulary tokens:** Identifiers, class names, or library-specific tokens appearing in the test corpus that were never observed during training are mapped to `[UNK]`, reducing prediction accuracy for subsequent context windows that contain them.
2. **Data sparsity for longer contexts:** Even with the trigram model, many 3-token sequences in the test set appear only rarely or never in training.
3. **Naming convention variation:** Different repositories use different naming patterns for the same things, causing the model to predict the more common name from training even when context should favors the less common form.
4. **Ambiguity:** In contexts following assignment operators or method call parentheses, many tokens are equally probable which can lead to frequent mismatches even when the model is well trained.

Analysis

The experimentation confirms several things about N-gram language modeling for code token prediction at this dataset scale:

Training set size matters: Expanding the training corpus from T1 ($\leq$15K methods) to T3 ($\leq$35K methods) reduced the best validation perplexity by roughly 28% Larger corpora produce broader vocabularies which directly improving predictive confidence.

Lower-order n-grams perform better: The trigram ($n = 3$) model consistently outperforms the 5-gram and 7-gram models by a wide margin across all training sets and α values. This is likely due data sparsity: the number of unique n-gram types grows rapidly with n and vast majority of higher-order contexts are observed too infrequently to yield reliable probability estimates. One way to resolve this is through backoff or interpolation

Minimal smoothing is best. The smallest tested α (0.001) uniformly produces the lowest perplexity across all configurations, suggesting the training corpora are relatively dense at the trigram level.

Conclusion

This project demonstrates a complete end-to-end pipeline for probabilistic Java code token completion using N-gram language models. The best model—a trigram trained on the largest corpus (T3) with minimal smoothing ($\alpha = 0.001$)—achieves a validation perplexity of 265.96. This indicates that with the given dataset, broader larger datset size and shorter context windows are the most effective at improving prediction quality.

Future work could implement interpolated or Kneser-Ney smoothing and backoff strategies. There could also be a larger training set generated by relaxing the constraints for data selection.