N-Gram Language Modeling for Java Code

Author: Alice Ji
Course: Gen AI for Software Development CSCI 455
Professor: Antonio Mastropaolo
Date: February 20 2026
Repository: `https://github.com/alicexji/N-Gram`

Abstract

This project investigates statistical language modeling for source code using n-gram models trained on Java repositories mined from GitHub. A large corpus of tokenized Java methods was constructed and partitioned into training, validation, and test sets. Models with $n \in \{3, 5, 7\}$ were evaluated using add-alpha smoothing and backoff smoothing, with validation perplexity used for model selection. To support efficient large-scale evaluation, probability computation was optimized through memoization and indexed next-token lookup. The best-performing configuration was automatically selected and used to generate structured JSON outputs containing token-level predictions and test perplexity. Results show that higher-order n-gram models with backoff smoothing substantially outperform add-alpha under sparsity, achieving the lowest validation perplexity. The project presents a complete end-to-end pipeline—from repository mining to model evaluation—highlighting both statistical modeling principles and practical scalability considerations.

Dataset Construction

To construct the dataset, I first cloned the Google Guava repository as a seed project. Guava is a well-established, actively maintained Java library and served as a stable baseline during early development and validation of the extraction pipeline. Then additional repositories are retrieved automatically using the GitHub Search API. Repositories were filtered using the criteria:

- Language: Java
- Minimum stars: ≥ 1000
- Minimum repository size: ≥ 1000 KB
- Not a fork
- Recently updated (pushed within the past year)

```
q = (
    f"language:java stars:>={min_stars} "
    f"fork:false size:>={min_size_kb} pushed:>={pushed_after}"
)
```

These filters were designed to ensure that the dataset reflects realistic, production-quality Java code rather than inactive repositories.

API results were paginated until the desired number of repositories was collected. A short delay was introduced between API requests to respect GitHub rate limits. All selected repositories were cloned locally using the `GitPython` library.

Method Extraction and Tokenization

Java source files were processed to extract individual methods. Each `.java` file was first cleaned to remove non-ASCII characters to prevent encoding and parsing issues. Each method was tokenized (methods containing fewer than 10 tokens were discarded to remove trivial samples) and stored as a single whitespace-separated sequence, with one method per line. This representation allows each method to serve as an independent training instance for the language model.

The dataset was partitioned into training sets:

- $T1$: 15,000 methods
- $T2$: 25,000 methods (first 25,000 methods; includes $T1$)
- $T3$: 35,000 methods (first 35,000 methods; includes $T2$)

The nested structure ($T1 \subset T2 \subset T3$) enables controlled comparison of model performance as training data size increases, isolating the effect of additional data while keeping earlier examples fixed.

In addition, a validation set of 1,000 methods and a self-created test set of 1,000 methods were created. The validation set was used for hyperparameter selection, while the test set was reserved for final evaluation. This design ensures a clean separation between training, validation, and test data, preventing data leakage during model selection.

Training

I implemented count-based n-gram language models with two different smoothing strategies: add-alpha and backoff. All models were trained and evaluated using validation perplexity for model selection.

Add-Alpha Baseline I first implemented a standard add-alpha smoothed n-gram model with $\alpha = 0.1$. For each training split, models were trained with $n \in \{3, 5, 7\}$. The vocabulary was constructed exclusively from the training data, and out-of-vocabulary tokens were mapped to `<UNK>`. The implementation for this model is in `modeling/ngram_model.py`.

After running the model with add-alpha, I observed that $n = 3$ consistently outperformed $n = 5$ and $n = 7$ on the validation set. This suggested that higher-order models were suffering from data sparsity: as n increases, many contexts appear rarely or not at all, causing the probability mass to be overly diluted by add-alpha smoothing. To address this issue, an additional backoff model was implemented.

Justification for Backoff Unlike add-alpha, which smooths every n-gram uniformly, backoff attempts to use the highest-order model available and only falls back to lower-order models when necessary. If all higher-order contexts are unseen, the model ultimately backs off to a unigram distribution. The implementation for this model is in `modeling/backoff_ngram_model.py`.

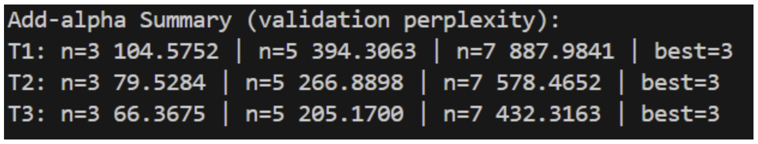
At inference time, the model attempts the highest-order n-gram first; if the n-gram count is zero, which means there is no context, it multiplies by β and recursively backs off to shorter contexts until a seen context is found. Unigram probabilities are smoothed with add-alpha to prevent zero probability assignments. I used a backoff factor $\beta = 0.4$ and unigram smoothing $\alpha = 0.1$.

To prevent very slow and expensive performance, probabilities are memoized. Once $P(\text{context}, w)$ is computed, it is cached and reused. This was necessary because backoff is expensive and would take too much time to run if we didn't save previous computations.

Each computed probability for a $(\text{context}, \text{token})$ pair is stored in a cache and reused when requested again.

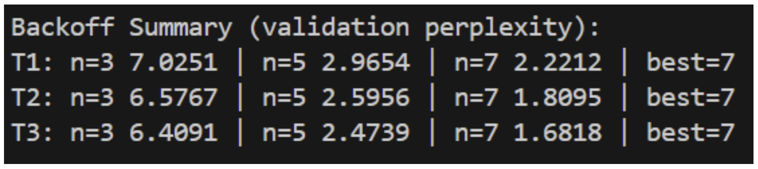
```
self._prob_cache: Dict[Tuple[Tuple[str, ...], str], float] = {}

cached = self._prob_cache.get(key)
if cached is not None:
    return cached
```

Model Selection Procedure For each of the three training splits ($T1$, $T2$, $T3$), I trained models with $n \in \{3, 5, 7\}$ for both add-alpha and backoff smoothing strategies. Perplexity was computed using the probability assigned to the ground-truth next token at each position.



```
Add-alpha Summary (validation perplexity):
T1: n=3 104.5752 | n=5 394.3063 | n=7 887.9841 | best=3
T2: n=3 79.5284 | n=5 266.8898 | n=7 578.4652 | best=3
T3: n=3 66.3675 | n=5 205.1700 | n=7 432.3163 | best=3
```

Figure 1: Validation perplexity comparison across training sets using add-alpha.



```
Backoff Summary (validation perplexity):
T1: n=3 7.0251 | n=5 2.9654 | n=7 2.2212 | best=7
T2: n=3 6.5767 | n=5 2.5956 | n=7 1.8095 | best=7
T3: n=3 6.4091 | n=5 2.4739 | n=7 1.6818 | best=7
```

Figure 2: Validation perplexity comparison across training sets using backoff.

The configuration (training split, n, and smoothing method) with the lowest validation perplexity was selected as the final model. Its hyperparameters were saved to a configuration file called `best_config.json` and later used for evaluation on the test set.

Observation While add-alpha smoothing performed adequately for lower-order models, higher-order models exhibited increased perplexity due to sparsity. Backoff smoothing substantially reduced validation perplexity, particularly for $n = 5$ and $n = 7$, confirming that selective backing off to lower-order contexts is more effective than uniformly smoothing all n-grams.

Evaluation

After selecting the best configuration using validation perplexity, the final model evaluated on two different test sets. One created by me called `test_self.txt` and one provided called `provided.txt` Evaluation was performed using the `stage_json` pipeline, which loads the saved configuration, rebuilds the selected model, and generates structured JSON outputs containing predictions and perplexity.

Perplexity was computed using the probability assigned to the ground-truth next token at each position. This is implemented in `evaluate_to_json.py` as:

```
p_gt = model.prob(context, gt)
total_log_sum += math.log(p_gt)
total_N += 1
```

Here, `gt` denotes the ground-truth next token from the test sequence. The final perplexity is computed by normalizing the accumulated log probability by the total number of tokens and exponentiating.

Prediction Reporting In addition to computing perplexity, the evaluation pipeline records the model's argmax prediction for each context. For efficiency, a precomputed next-token index is used to approximate the most likely continuation without iterating over the entire vocabulary. However, these predictions are used only for reporting purposes in the JSON output and do not affect perplexity computation.

Results and Discussion

Best Model Outcome Across all configurations evaluated during validation, the best-performing model used training set T3, order $n = 7$, and backoff smoothing. This configuration achieved the lowest validation perplexity (1.6818), substantially outperforming the add-alpha baseline (66.3675).

The strong performance of the 7-gram backoff model suggests that longer contexts were informative when supported by selective smoothing. While higher-order n-grams suffer from sparsity under add-alpha smoothing, backoff allows the model to exploit long contexts when available and revert to shorter contexts only when necessary. Using the selected configuration (T3, $n = 7$, backoff), the model achieved:

- Perplexity = **1.6432** on the self-created test set,
- Perplexity = **5116.3263** on the instructor-provided test set.

For comparison, during debugging I also evaluated a lower-order configuration (T3, $n = 3$, backoff), which yielded:

- Perplexity = 6.1054 on the self-created test set,
- Perplexity = 384.9139 on the instructor-provided test set.

The very low perplexity (1.6432) on the self-created test set indicates that the selected 7-gram backoff model models that distribution extremely well. This is expected, as the self-created test data was mined using the same pipeline and therefore closely matches the training distribution.

In contrast, the instructor-provided test set yielded a much higher perplexity (5116.3262), indicating substantial distributional shift. The 7-gram model relies heavily on long contexts; when such contexts are unseen, repeated backoff and multiplicative discounting produce very small probabilities, resulting in large perplexity values.

Interestingly, the lower-order 3-gram backoff model generalizes better to the instructor dataset. Although it performs worse in-distribution, its reliance on shorter contexts makes it less sensitive to sparsity and domain mismatch.

These results illustrate a tradeoff: higher-order models achieve extremely low perplexity when evaluation data matches the training distribution, while lower-order models may generalize more robustly under distribution shift. Validation-based selection correctly identified the best in-distribution model (T3, $n = 7$, backoff), but cross-dataset evaluation reveals its sensitivity to domain differences.