WILLIAM & MARY

CHARTERED 1693

CSCI 455/555: GenAI for SD

Spring 2026

Prof. Antonio Mastropaolo

Assistant: Md. Zahidul Haque Alvi

Recommending Code Tokens via N-gram Models

Your goal is to implement a probabilistic language model (*i.e.,* an N-gram model) that assists with code completion for Java methods. Formally:

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

where t_n is the next token and the preceding $n-1$ tokens form the context.
Note: For an n-gram model, you use the previous $n-1$ tokens as context to predict the n-th token (e.g., a 5-gram uses tokens 1–4 to predict token 5).

Task Requirements

- **Corpus construction**: The language is fixed (*i.e.,* Java) and the unit of analysis is method-level.
- **Context window**: Treat n as a tunable hyper-parameter. You must experiment with $n \in \{3, 5, 7\}$ and report perplexity for each.
- **Evaluation**: Intrinsic evaluation using perplexity.

  Perplexity measures how confident the model is when predicting the next token; **lower is better**.

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

  Important: When computing perplexity, $P(w_i \mid \text{context})$ refers to the probability that the model assigns to the **ground-truth next token** w_i. This is computed from the model distribution $P(\cdot \mid \text{context})$, regardless of the token $\hat{w}_i$ that the model would predict (i.e., $\arg\max_w P(w \mid \text{context})$).

Required Pipeline and Implementation Guidelines

You must implement the following end-to-end pipeline:

1. **Mine and prepare data (replicate the pipeline we saw during lab)**:
   - Define and justify criteria to select repositories and Java files (e.g., popularity, activity, size, exclusion rules).
   - Extract method-level code from the selected files.
   - Apply the same preprocessing used for the provided datasets (remove non-ASCII characters, filter methods with fewer than 10 tokens, remove duplicates).

- **Dataset splits**: Define three training sets (T_1, T_2, T_3), one validation set (V), and one test set (Te). The training sets must be capped at: $|T_1| \le 15{,}000$, $|T_2| \le 25{,}000$, and $|T_3| \le 35{,}000$ instances. For the purposes of this assignment, it is acceptable to use approximately 1,000 examples for validation (V) and 1,000 examples for testing (Te).
- **Tokenize methods**: Tokenize raw Java methods into space-separated tokens. Ensure that keywords, identifiers, literals, operators, and punctuation symbols (e.g., `(`, `)`, `{`, `}`, `;`, `.`) are emitted as separate tokens.
- **Build the vocabulary**: For each training set (T_1, T_2, T_3), build a *separate* vocabulary from scratch using only that training set. Tokens not present in the corresponding vocabulary must be mapped to `<UNK>` during validation and testing.

2. **Train models**: Train n-gram models for $n \in \{3, 5, 7\}$ on the training set.
3. **Handle unseen n-grams (required)**: Implement smoothing to avoid zero probabilities (e.g., add-α or backoff). Document your choice and hyper-parameters.
4. **Select the best configuration**: Choose the best model using the validation set (lowest perplexity).
5. **Evaluate on two test sets**: Evaluate the selected model on:
   - the **provided test set** with ~1K examples, and
   - your **self-created test set** with ~1K methods mined using the same pipeline.

   Note: *We are not designing the self-created test set to study distribution shifts; the provided test set is expected to already capture such variation.*
6. **Compute perplexity correctly**: When computing perplexity, use the probability assigned to the **ground-truth next token** at each position (not the predicted token). This is computed from $P(\cdot \mid \text{context})$ regardless of $\arg\max$ predictions.
7. **Generate output files**: Produce two JSON files following the format in the next section: `results-xxxxxx.json` (provided test set) and `results-yyyyyy.json` (self-created test set).

Warning: Zero-probability n-grams cause division by zero when computing perplexity; smoothing/backoff is mandatory.

Input Format and Examples

- **Input**: Your script reads a `.txt` file from the command line. Each line contains one tokenized Java method (space-separated tokens).

  Input format example:

```
public void setName ( String name ) { this . name = name ; }
public int getAge ( ) { return this . age ; }
public boolean isEligible ( int minAge ) { return this . age >= minAge active ; }
public String formatUser ( String prefix ) { String s = prefix +  ... }
public void updateStats ( int delta ) { if ( delta > 0 ) { this . score  ...  }
```

JSON Output Format Each JSON file should contain results for each test method. Example structure:

```
{
  "testSet": "provided.txt",
  "perplexity": 4.39,
  "data": [
    {
      "index": "ID1",
      "tokenizedCode": "public void run ( ) { }",
      "contextWindow": 3,
      "predictions": [
        {
          "context": ["public", "void"],
          "predToken": "run",
          "predProbability": 0.72,
          "groundTruth": "run"
        },
        {
          "context": ["void", "run"],
          "predToken": "(",
          "predProbability": 0.85,
          "groundTruth": "("
        }
      ]
    }
  ]
}
```

Note: `predProbability` is the probability that maximizes $P(t_n \mid \text{context})$, *i.e.,* the probability of the token selected by your model given the current context. You must also record the corresponding `groundTruth` token.

Submission Instructions

Deadline: Friday Feb 20th, 2026 @ 11:59 PM.
Your submission must include:

- A 3–4 page report describing (i) MSR procedures (i.e., how instances were collected, cleaned etc...), (ii) the model training procedure, and (iii) the evaluation results, with links to relevant implementation in the repository.
- Source code for training and evaluation (well-documented, with clear function boundaries).
- Two JSON output files, both following the format above: `results-xxxxxx.json` generated on the provided test set, and `results-yyyyyy.json` generated on your newly created test set.
- A short `README` describing:
  - how to install dependencies,
  - how to run training/validation/testing,
  - where outputs are written,
  - any hyper-parameters you tuned (e.g., n, smoothing).

Each student must submit on Blackboard the link to their GitHub repository containing the materials listed above.

Grading Rubric (100 Points)

Component	**Points**	**Criteria**
Report Quality	20	3–4 page report covers: (i) data mining procedures (how instances were collected, cleaned), (ii) model training procedure, (iii) evaluation results with perplexity for $n \in \{3,5,7\}$. Includes links to code in repository.
N-gram Implementation	25	Correct n-gram models for $n \in \{3,5,7\}$. Proper smoothing implemented (add-α or backoff). Perplexity computed correctly using ground-truth token probabilities.
Test Set Creation	15	Self-created test set (~1K methods) follows required preprocessing: non-ASCII removal, 10-token filter, deduplication, proper tokenization.
Validation & Best n Selection	10	Validation set used to compare perplexity across $n \in \{3,5,7\}$. Best n selected based on lowest validation perplexity.
Code Quality & Replicability	20	Well-documented source code. Clear function boundaries. Easy to follow and reproduce results.
README File	10	Covers: dependency installation, running train/validation/test, output locations, hyper-parameters tuned (n, smoothing).

Total: 100 points