Report: N-Gram Tokenization and Method Extraction for Java Code Completion Model

Code: Under spring-boot folder -> ngram_model.py

Objective:
The purpose of this task is to build an N-gram based probabilistic language model for Java method code completion. The steps involve extracting Java methods from a repository, tokenizing them, and then applying N-gram models (Unigram, Bigram, Trigram) to learn patterns in the methods. The model is expected to aid in auto-completing Java method names based on previous tokens.

Step 1: Selecting a Repository

We initially worked with the Spring Boot repository. This repository contains thousands of Java files across multiple subfolders, including test files and system files. It was selected because it provides ample code to work with and is structured in a way that is typical of large Java projects.

We used the following commands to download and set up the repository:

```
git clone https://github.com/spring-projects/spring-boot.git
cd spring-boot
```

The repository is structured with directories like `src`, `buildSrc`, and `spring-boot-tests`, which contain the majority of the Java methods we extracted.

Step 2: Method Extraction with `tokenize_methods.py`

The method extraction script, `tokenize_methods.py`, is responsible for reading Java files from the repository and extracting methods. A key challenge was to only extract method signatures with more than one token so that the N-gram model could learn meaningful patterns.

We employed regex to extract method signatures and method bodies. In order to ensure compatibility with Bigram and Trigram models, we implemented a method that only tokenizes methods with at least three tokens. We also set up timeout-based extraction for very large files and skipped files larger than 10 MB to avoid processing delays.

Key Aspects of the Script:
1. Regex-Based Extraction: The script uses regex to extract Java method signatures and method bodies. This is crucial because methods in Java are often complex with generics, modifiers, and access levels (e.g., `public`, `private`).

2. Tokenization: Each method body is tokenized using regex, which splits the body into meaningful tokens like keywords, operators, and method names.
3. File Size Skipping: The script skips files larger than 10 MB, avoiding processing timeouts.
4. Timeout Handling: If a file takes more than 30 seconds to process, it is skipped.
5. Progress Monitoring: We used the `tqdm` library to display progress bars for tracking file processing.

Here is how we ran the script:

```
python tokenize_methods.py
```

After running the script, the method tokens are saved into a file named `tokenized_methods.txt`.

Step 3: N-Gram Model Application with `ngram_model.py`

After extracting and tokenizing methods, we applied N-gram models to generate predictions for code completion. N-gram models predict the next token based on the previous one or two tokens, depending on the model (Unigram, Bigram, Trigram).

Key Aspects of the Script:
1. N-Gram Generation: The script constructs N-grams from the tokenized methods by taking groups of one, two, or three consecutive tokens.
2. Vectorization: The `TfidfVectorizer` is used to convert the N-grams into a numerical format that can be processed by the machine learning model.
3. Classification Model: We used a Linear Support Vector Classifier (`LinearSVC`) to classify the token sequences and predict method tokens.
4. Performance Metrics: Precision, Recall, and F1 Score are calculated to evaluate model performance for each N-gram model.

We ran the N-gram model script using:

```
python ngram_model.py
```

When prompted, we specified whether to run the Unigram, Bigram, Trigram, or all models sequentially. After running, the script outputs performance metrics for each model.

Example Output:

```
Please specify the N-gram model (Unigram, Bigram, Trigram, All): All
Running Unigram model...
Precision: 65.53
Recall: 65.53
F1 score: 65.53

Running Bigram model...
Precision: 89.79
Recall: 89.79
F1 score: 89.79

Running Trigram model...
Precision: 95.71
Recall: 95.71
F1 score: 95.71
```

Step 4: Results

The performance of the Trigram model was significantly better than the Unigram model, as it could predict method names based on more context (i.e., the two preceding tokens). The Bigram model performed moderately well but was less accurate than the Trigram model.

Conclusion

Through this process, we successfully extracted and tokenized Java methods from a large repository and applied N-gram models to them. The results show that Trigram models perform the best, providing a high level of accuracy in predicting method tokens based on context. The scripts we developed are robust, allowing for timeout handling and progress monitoring.