Llama fine tune with intellij-community code data

In this guide, I'll outline my methodology for fine-tuning a code transformer using a dataset from a public repository. The primary workflow can be segmented into two key phases: data extraction and model refinement.

Data scraping

For this part, I gathered data from the public repository of IntelliJ Community Edition using Python. Initially, I focused on scraping Python and Java as a preliminary demonstration, but there's potential to expand to other languages like Kotlin. To extract method code from each programming language file, I utilized an Abstract Syntax Tree (AST) and isolated functions (methods) associated with a class parent.
For organization, I segregated the methods into a training and testing set, allocating 70% to train.txt and the remaining 30% to test.txt. In future, I could introduce a validation set, especially beneficial for tasks like early stopping. To manage computational resources and prevent overwhelming file sizes, I implemented an integer parameter, n_files, to control the number of files processed. Given that the text files for Java can grow to over ~10 GB, this precaution becomes essential.
While I've currently employed an approach based on text files for this proof of concept, I could implement a more streamlined approach for practical applications. I could extract classes dynamically, batching them on-the-fly, and directly feed them to the model for training, eliminating the need for intermediate text files.
For now the resulting files are:

- python_train.txt
- python_test.txt
- java_train.txt
- java_test.txt

Model fine-tuning

In this segment, I detail the fine-tuning methodology employed for Code Llama. Initially, I extract datasets from the text files and structure the samples to align with the requirements of Llama:

```
prompt="{prefix} <FILL_ME> {suffix}", completion="{middle} _<EOT>"
```

Subsequently, the model undergoes validation, where both accuracy and Levenshtein distance serve as metrics. The outcomes are presented in Table 1 for Java and Python for 150 (n_files parameter).

	Similarity	**Accuracy**

Python	0.2932	0.125
Java	0.536	0.2333

Table 1: Results before fine tuning.

Within this table, metrics for accuracy and similarity are delineated. Accuracy evaluates if the predicted method name precisely matches the original, hence representing a stricter criterion. On the other hand, similarity is computed as:

$$Similarity = 1 - \frac{levenstein(prediction, actual)}{max(length(prediction), length(actual))}$$

This metric leverages the Levenshtein distance and offers a more lenient perspective, illustrating the alterations required to transform one string into another, normalized by their respective lengths.
After the first evaluation, the model is trained using LoRA and the resulting version is evaluated again on the test set. The final results can be viewed on Table 2 for both Java and Python. After a minor fine-tuning, there's a noticeable improvement in performance, with a rise of ~2% in similarity and ~2.5% in accuracy for Python. When it comes to Java, there is also an improvement of ~10% and ~5% for accuracy, similarity respectively.
Based on the improvements observed, it's evident that the model effectively learns from the dataset. Training on the entire corpus is expected to yield even better results.

	Similarity	Accuracy
Python	0.31419	0.149
Java	0.5849	0.333

Table 2: Results after fine tuning.