

Train your own Copilot for the Julia programming language

You have probably already used tools like GitHub Copilot to generate entire functions starting from a simple description in natural language. Such a task is known as “code generation”. Popular AI-based code generators such as Copilot are known to be very effective on “high-resource languages”, namely programming languages for which a lot of training material is available online. Differently, they may exhibit sub-optimal performance when working on low-resource languages such as Julia. Your task is to train a Transformer model specialized in code generation for Julia code. Below are the steps to follow:

📦 Mining Software Repositories: Your first step is to create a dataset containing Julia functions. Use your Mining Software Repositories skills to mine Julia projects from GitHub: clone and parse their latest snapshot to extract the functions. For the projects' selection, consider using <https://seart-ghs.si.usi.ch>. We recommend the `tree-sitter` library for extracting functions from source code, but any other library of your choice is ok. Remember to remove functions present in the test dataset (more information in Appendix A).

❓ Consider cleanings of the dataset that could be beneficial. First, while training for code generation usually requires a docstring of the function, you can still use functions without docstring by inputting, in that case, only the signature to the model. What about inner comments? If you keep them, this means that during training you are asking the model to generate, starting from the docstring and the function's signature, a function implementation which includes inner comments. What about removing functions that are syntactically invalid or represent dummy implementations? These choices are up to you.


🔧 Fine-tuning the model: Once your dataset is ready, you can start training your model. Use SmolLM-135M as the starting model and then train its larger variant SmolLM-360M on the same dataset. In this task, the model will take as input the signature of a Julia function and a docstring (if available) and will try to output a possible implementation for that function.

💡 Bonus points: Larger models, better performance! Try fine-tuning the largest version of SmolLM (see SmolLM-1.7B) using the *Parameter Efficient Fine-Tuning* (PEFT) technique seen in class.

📊 Evaluating the model: We provide a bash script to evaluate the performance of the model you trained. The evaluation is based on the MultiPL-E benchmark, which consists of 159 prompts (*i.e.*, docstring plus function signature), including unit test cases to assess the correctness of the model's implementations. The evaluation metric is the *pass@1*, which defines the percentage of functions that your model correctly implemented (*i.e.*, the implementation passes the tests) using only a single attempt. You can find more information on how to use this benchmark in Appendix A.

🔍 Statistical tests: How does your model perform compared to GitHub Copilot? In folder `benchmark/results`, we have collected the results of GitHub Copilot on the MultiPL-E benchmark. Check if there is a statistically significant difference between the performance of your models and that of GitHub Copilot. Does the size of the model impact the performance of the trained model in the code generation task? To answer this question you can compare the performance of the fine-tuned SmolLM-135M model against the SmolLM-365M (and SmolLM-1.7B, if you decide to go that way). Remember to use appropriate statistical tests.

Submission Instructions

 **Deadline:** Friday December 13th, 2024 @ 18:00.

Each group must submit on iCorsi an archive including:

1. A report describing how the training dataset has been built and how the models have been trained, with links to the used code. Also, the report must feature the statistical analyses comparing the trained models to Copilot as well as the trained models among themselves (to investigate the impact of the models' size).
2. For each trained model, a link to a json file with the following columns:
 - Input provided to the model;
 - Whether the prediction passed all test cases (true/false);
 - Predicted implementation;

Name the json files as `smollm-135-testset.json`, and `smollm-360-testset.json`.

A Appendix

MultiPL-E is a benchmark to evaluate code generation models across different programming languages. In this project, you will use MultiPL-E to assess the performance of your fine-tuned models on the Julia language. You can find more information about this tool in the official repository and in this article. Below, we list some useful information for using this tool.

In the `benchmark` folder, we have included the following files:

- `results`, a folder containing the results of GitHub Copilot on the MultiPL-E benchmark and which will also include the prediction results of your evaluated models.
- `generate.sh`, a bash script to generate predictions from your fine-tuned model on MultiPL-E prompts. This script takes as first argument the path of the model checkpoint to evaluate or the name of a model from HuggingFace. Remember to specify the `CUDA_VISIBLE_DEVICES` variable present within the script before launching it!
- `evaluate.sh`, a bash script to run test cases on your model predictions. Before launching this script, it is necessary to download and configure Docker on your machine. Again, the script takes as an argument the name or path of the checkpoint to be evaluated. In output, it returns a JSON file containing the model predictions and a CSV file containing the *pass@1* value (under the 'estimate' column).

Below is an example of using MultiPL-E on *DeepSeek Coder 1.3B* model:

```
# Install the required Python packages
pip install -r requirements.txt

# Generate predictions from DeepSeek Coder 1.3B model
./generate.sh deepseek-ai/deepseek-coder-1.3b-base

# Evaluate the predictions (verify that Docker is running)
./evaluate.sh deepseek-ai/deepseek-coder-1.3b-base
```

Please check that you are able to run these scripts as soon as possible so that we can intervene if there are any issues. Feel free to customize the scripts to better suit your needs.

Inside the `humaneval-jl-reworded.json` file you can find the prompts on which the model is evaluated. You must remove duplicates within your train set by extracting the function name from these prompts and ensuring that no function in the train set features this name.

Before running the `evaluate.sh` script make sure to "clean" your predictions. Indeed, these models may output more than just the function implementation, adding e.g., a textual description of what they implemented. Make sure that you pass to the `evaluate` script only the implementation (body) of the required function.