Fine-tuning LLM for Multilingual Code Generation

Link to Github Repo: LLM-for-code-intelligence

Link to Google Doc to add comment and see latest version: click here

OVERVIEW

- **Very** Large code LLMs (> 1B params) often have good performance in a variety of languages (**Salesforce/codegen-2B-multi -> C, C++, Go, Java, JavaScript, and Python**)
- **Small** code LLMs (< 1B params) are usually trained to work on one programming language to have good performance (**Salesforce/codegen-350M-mono -> Python**)
- Small code LLMs can be extended to cover more languages by a variety of methods:
    a. Full Fine-tuning:
        - Pros: Best performance in fine-tuned model
        - Cons: High computational requirements, catastrophic forgetting
    b. LoRa Fine-tuning:
        - Pros: Efficacy, Scalable, No catastrophic forgetting
        - Cons: Lower performance than full fine-tuning
    c. Knowledge Distillation: tbd

GOALS

1. Optimally extend Small code LLMs to new programing language considering:
    a. Performance: in terms of Perplexity and HumanEval
    b. Efficiency: in terms of Memory and Time
2. Study the effect of different factors on the training, evaluation and inference of models:
    a. Training Data: Selection of github repos and how to filter them
    b. Training Strategies: learning rate, batch size, gradient accumulation
    c. Evaluation: Stride and stop tokens
    d. Inference: Sampling strategy and temperature
3. Qualitative Analysis of the results:
    a. Failure modes comparison in HumanEval between Python and Java (Exceptions, runtime errors and not passing test cases)
    b. Correlation between Perplexity and HumanEval

4. Further objectives:
    a. Relation between Different Programming Languages
    b. Knowledge Distillation

Timeline:

1. Read papers on Code LLM and select appropriate baseline -> **Salesforce/codegen-350M-mono**
2. Read papers on Code datasets and select perplexity evaluation data -> **bigcode/the-stack-dedup**
3. Read papers on evaluation strategy to implement HumanEavl in python and Java -> follow standards of **CodeX** and **Multipl-E**
4. Setting up Evaluation:
    - Done: Perplexity Evaluation on Google-Colab and HumanEval on Google-Colan and Virtual Containers (for Java)
    - Challenges: Perplexity Dataset, Humaneval Stop tokens for java
5. Generate Baseline Results using baseline model
6. Setting up Training and Evaluation environments:
    - Done: Configure Full & LoRa fine-tuning on Google-Colab, Track training progress on WandB, Save checkpoints on HuggingFace.
    - Challenges: Resuming Training reset some variables(LR, number of steps), Memory to 15GB and duration to 3 hours
7. Generate Results for the full tuned model and LoRa tuned model.
8. Edit the training setup (learning rate , gradient accumulation and batch size) based on the literature (**bigcode/starcoder**) and the results of the first finetuning
9. Train the model again and notice the effect of modification on two models:
    - Full fine tuning: *Significant improvement* in Perplexity and HumanEval for **Java** and *significant drop* for **Python**
    - LoRa fine tuning: *Significant improvement* in Perplexity and *Slight* in HumanEval for **Java** and *no change* for **Python**
10. Plan Next steps: further modifications, larger models, inference and distillation
11. Setting up live inference environment:
    - Done: Live inference configured on HuggingFace spaces to generate code from tuned models base on given prompt -> Code Inference
    - Challenges: CPU limitation
12. Got the best parameters for each model to train and evaluate effectively, and generated final results of main experiment
13. New releases suggests smaller models needs much more data CodeGen2.5: Small, but mighty (salesforceairesearch.com)

Results

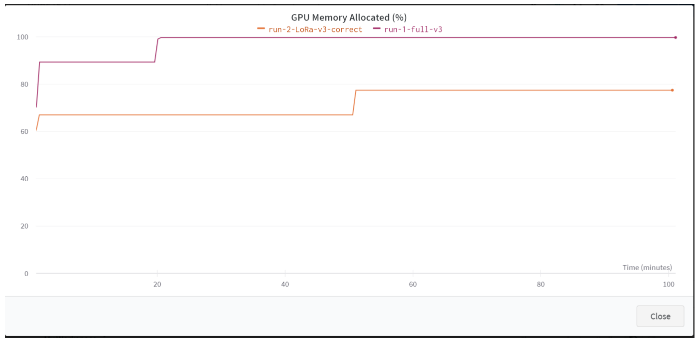
1. Setup:

a. Objective:

- We want to adopt a monolingual code LLM from one programming language (Python) to another (Java)
- In order to do so , we start with monolingual baseline model : **Salesforce/codegen-350M-mono(Python)**
- This is an autoregressive code completion model pretrained on The mono-lingual dataset BIG PYTHON
- To verify our claim, i.e, you can efficiently adopt a small LLM to work on a variety of programming languages, we will need to fine-tune this baseline on a Java corpus.
- For this we will use two fine tuning methods: Full fine-tuning (100% of the model parameters) and LoRa finetuning (5% of the model parameters)

b. Fine Tuning Hyper-Parameters:

- The Latest fine-tuning hyper-parameters were selected based on empirical experiments with the objective of maximizing the Pass@100 score on Java while simultaneously using no more than 15GB of Memory and minimum training time.
- ***Batch Size***: For both LoRa and Full fine-tuning we ended up using an effective batch size of **32** (Normal batch size = 16, and gradient accumulation = 2). The use of gradient accumulation however slowed the training step in both LoRa and Full from **0.2step/second** to **0.09step/second**. This was an acceptable increase given that we can use a higher batch-size with no extra memory requirements. Finally, a batch size of 32 utilizes all of the **15GB** of memory in full fine-tuning training, but only **9GB** in LoRa.



- ***Learning Rate***: For both LoRa and Full fine-tuning we ended up using a learning rate of **0.00005** with **100 steps warmup** and **AdamW optimizer.** From our experiments , we noticed as the learning rate, the loss curves of the full fine-tuning setup fluctuates, while the LoRa is more robust. So we added a cosine scheduler to stabilize the full-fine tuning loss as training progressed. Finally we opted for AdamW optimizer instead of the more memory efficient Adagrad as we concluded that the percentage of memory saved (1%) does not make up for the inherent instability in Adagrad.

c. Training Dataset:

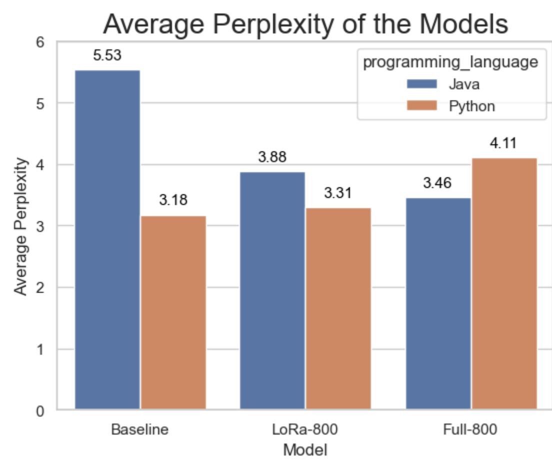
- For training we used the java subset of THE STACK datasets ammarnasr/bigcode-the-stack-dedup-java-small-subset

- One important factor here is that we did not use any pre-processing such as: (1) filtering, (2) deduplication, (3) tokenization, (4) shuffling, and (5) concatenation. As is in the case in most of the pretraining setups.

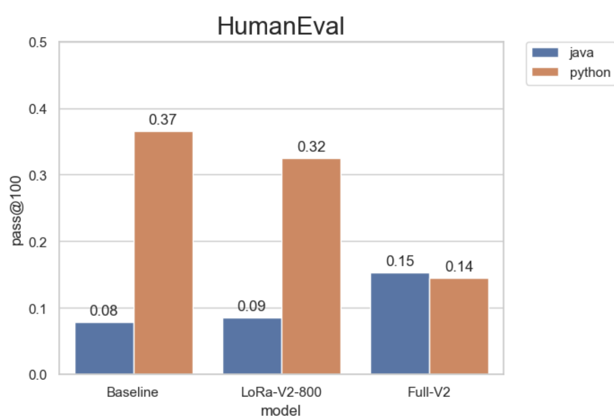
2. Comparisons

a. Comparing the models in terms of:

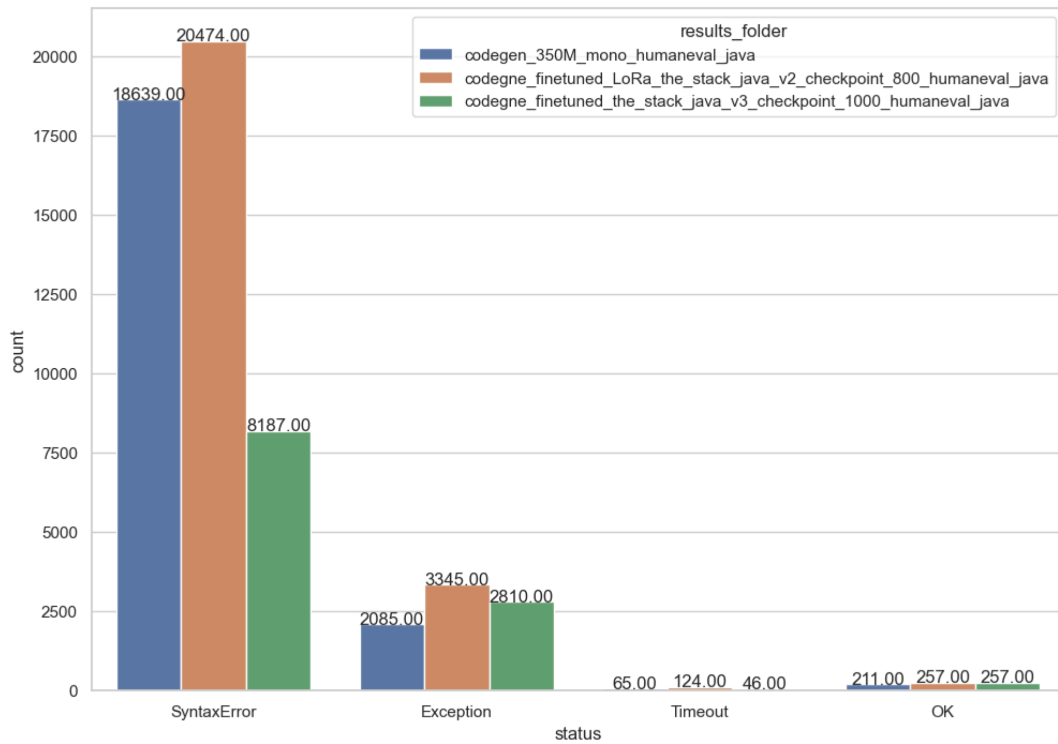
- Perplexity: The Lower the better

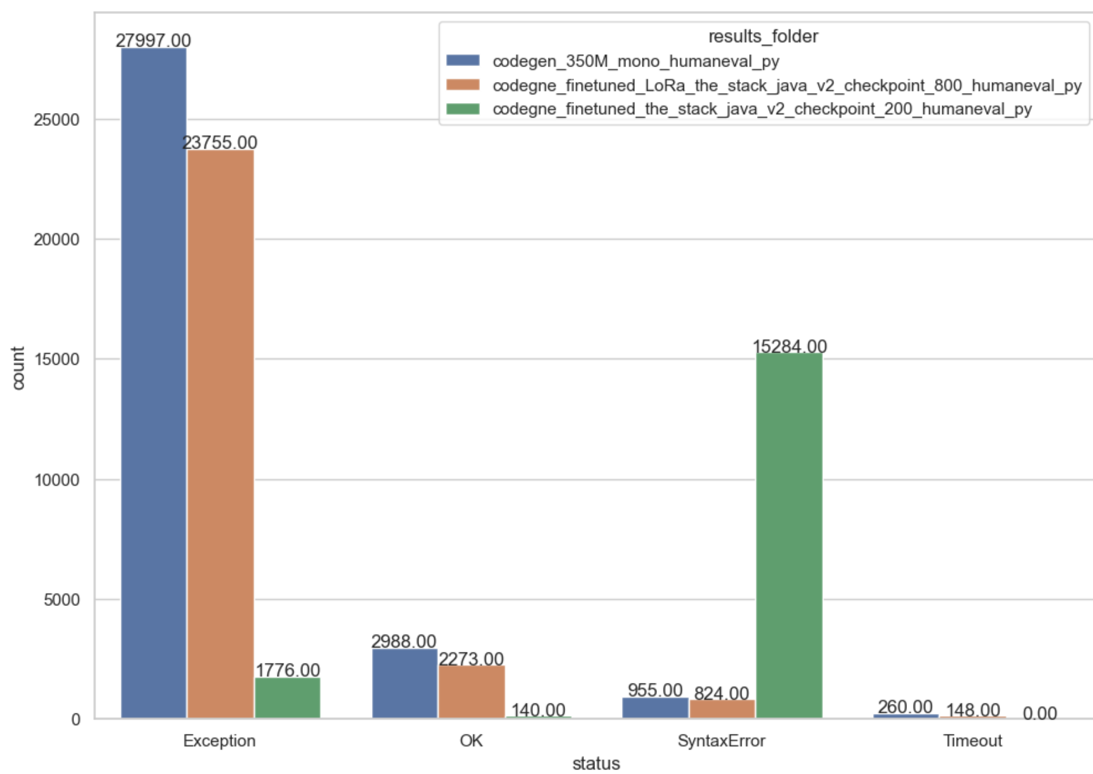


- HumanEval: The Higher the better



- Error Analysis


results_folder
codegen_350M_mono_humaneval_java
codegne_finetuned_LoRa_the_stack_java_v2_checkpoint_800_humaneval_java
codegne_finetuned_the_stack_java_v3_checkpoint_1000_humaneval_java
20474.00
18639.00
8187.00
3345.00
2810.00
2085.00
65.00
124.00
46.00
211.00
257.00
257.00
count
status
SyntaxError
Exception
Timeout
OK


results_folder
codegen_350M_mono_humaneval_py
codegne_finetuned_LoRa_the_stack_java_v2_checkpoint_800_humaneval_py
codegne_finetuned_the_stack_java_v2_checkpoint_200_humaneval_py
27997.00
23755.00
1776.00
2988.00
2273.00
140.00
15284.00
955.00
824.00
260.00
148.00
0.00
count
status
Exception
OK
SyntaxError
Timeout

3. Experiments

Link to Google Sheets full table and latest version: [sheet]

Code Name	Dataset Split	Seed	Seq Length	max_steps	eval_steps	optimizer	warm up steps	Learning Rate
full-v3-1000	0.0001	None	512	1000	50	adamw_hf	100	5.00E-05
full-v3-2000	0.0001	None	512	2000	50	adamw_hf	0	5.00E-05
LoRa-v3-1000	0.0001	None	512	1000	50	adamw_hf	100	5.00E-05
LoRa-v3-2000	0.0001	None	512	1000	50	adamw_hf	0	5.00E-05