Course Project [18 Points]

Generative AI

Saarland University – Winter Semester 2024/25

Part 1 Submission due: 30 January 2025 by 6pm CET (extended)
Part 2 Submission due: 6 February 2025 by 6pm CET (extended)

genai-w24-tutors
genai-w24-tutors@mpi-sws.org

1 Introduction

In the course project, we will explore how to leverage generative AI models for enhancing programming education. Earlier in Week 6 assignment, we explored the abilities of generative AI models in the domain of visual programming [1]. In this project, we will consider the scenarios involving program repair and hint generation for introductory Python programming [2, 3]. As part of the project, we aim to evaluate and improve these models systematically by establishing the baseline performance using basic prompts, enhancing generation quality through advanced prompting techniques, and fine-tuning the models to further improve their performance.

REMARK: The course project draws inspiration from a recent paper [3]. To get a better understanding about the problem setting and performance metrics introduced below, we recommend you to read this paper before starting to work on the project.

1.1 Project Parts

We have divided the projects into two parts as described below.

Project Part#1 [10 Points; Due: 30 January 2025, 6 PM CET (extended)]. The first part of the project focuses on evaluating the baseline models and exploring prompt engineering techniques for program repair and hint generation. You will use the provided datasets and scripts to analyze the performance of models such as GPT-4o-mini and Phi-3-mini. We will explore advanced prompting techniques to improve the generation quality.

Project Part#2 [8 Points; Due: 6 February 2025, 6 PM CET (extended)]. The second part of the project focuses on fine-tuning base models for program repair and hint generation. In particular, we will consider how to boost the generation quality of small models that are more suitable for training and deployment in resource-constrained settings. We will explore parameter-efficient training (LoRA) and experiment with varying hyperparameters, analyze the trade-offs between model quality and training efficiency.

1.2 Problem Setting

We will consider two scenarios involving program repair and hint generation, as formalized below [3]. Figure 1 shows an example of program repair and hint generation.

Program Repair. Given a programming task T and a learner's buggy program P_b, the goal is to generate a repaired program P_r with minimal edits w.r.t. P_b. The buggy program P_b fails to pass at least one test case in the task T's test suite Ω_T and may contain various types of errors, including syntax and semantic errors. The repaired program P_r is expected to fix these errors and pass all the test cases.

Problem Description

Sequential Search: Implement a search function that performs a sequential search. It takes in a value x and a sorted sequence seq, and returns the position that x should go to such that the sequence remains sorted. If the same value already exists, then place x before any previous occurrence. The sequence is sorted in ascending order. If the sequence is empty, return 0. The sequence can be a list or a tuple of integers.

(a) Programming task T.

Buggy Program

```
def search(x, seq):
    for i in range(len(seq)):
        if x <= seq[i]:
            return i
    return i + 1
```

(b) Learner's buggy program P_b.

Repaired Program

```
def search(x, seq):
    for i in range(len(seq)):
        if x <= seq[i]:
            return i
    return len(seq)
```

(c) Repaired program P_r.

Hint

Consider what happens when the input sequence is empty. How does your current implementation behave, and what variable might not be set in this specific scenario?

(d) Generated hint H.

Figure 1: Example of program repair and hint generation.

Hint Generation. Given a programming task T and a learner's buggy program P_b, the goal is to generate a natural language hint H as feedback to assist the learner in resolving errors and making progress. The primary objective is to provide the learner with a concise hint without explicitly revealing the solution or necessary fixes.

1.3 Performance Metrics

We consider several metrics to assess the overall performance of a model, including quality metrics for program repair and hint generation as well as usability metrics related to deployment in educational settings.

Quality metrics for program repair. The quality of program repair will be evaluated using two metrics that focus on accuracy and efficiency. *RPass* (binary) examines whether the repaired program passes all test cases, signifying a successful fix. *REdit* (non-negative number) captures the token-level edit distance between the buggy and the repaired program, with a smaller edit distance reflecting that the repaired program is better aligned with the learner's buggy program. When reporting results, these metrics will be computed automatically.

Quality metrics for hint generation. The quality of hint generation will be evaluated based on several binary metrics that define its pedagogical utility. *HCorrect* (binary) captures whether the generated hint provides correct information for resolving issues in the buggy program. *HInformative* (binary) captures whether the generated hint provides useful information to help the learner resolve bug(s). *HConceal* (binary) captures that the information in the generated hint is not too detailed, so the learner would also have to reason about implementing the fixes. *HComprehensible* (binary) captures whether the generated hint is easy to understand, presented in a readable format, and does not contain redundant information. We measure the overall quality of the generated hint by *HGood* (binary) that takes the value of 1 (good quality) when all the four attributes are rated as 1. When reporting results, these metrics would require manual annotations.

Usability metrics. Beyond the quality metrics, there are several important usability metrics for deployment in educational settings, related to model size, user privacy aspects, and training/inference time. To account for these usability metrics, we will consider a small model Phi-3-mini and further load it as a 4-bit quantized version for inference/training. In Part#2 of the project, we will explore parameter-efficient training (LoRA) and analyze the trade-offs between model quality and training efficiency. Here, we will use the metrics *TrainingTime* and *TrainingMemory* that capture the time and the memory used during the fine-tuning process.

2 Project Part#1 [10 Points]

The first part of the project focuses on evaluating the baseline models and exploring prompt engineering techniques for program repair and hint generation. The relevant files for this part have been released earlier.

3 Project Part#2 [8 Points]

The second part of the project focuses on fine-tuning language models for program repair and hint generation, exploring parameter-efficient training with Low-Rank Adaptation (LoRA) and multi-task fine-tuning.

3.1 Provided Datasets and Scripts

Similar to Part#1, we will use the IntroPyNUS dataset [3, 4]. In addition to the scripts and datasets provided for Part#1, new resources have been included for Part#2. More concretely, we have provided the following files in `project_part2.zip`:

- `project_part1_evaluate.py`, `project_part1_repair.py`, `project_part1_hint.py`, `project_part1_utils.py`, `project_part1_prompts.py`, `project_part1_datasets.zip`: These files are same as those provided in Part#1.
- `project_part2_evaluate.py`: This script is similar to `project_part1_evaluate.py`, with minor adaptations for better focusing on evaluating fine-tuned models.
- `project_part2_finetuning.py`: This script manages supervised fine-tuning (SFT) of Phi-3-mini using LoRA. It initializes the model and its tokenizer with specific configurations such as sequence length and LoRA hyperparameters (`r`, `lora_alpha`, `lora_dropout`). The `get_custom_dataset` function prepares the dataset for fine-tuning by tokenizing and padding inputs and labels. In `__main__`, the external `SFTTrainer` class is used to perform fine-tuning with customizable training arguments, including batch size, learning rate, and gradient accumulation. The fine-tuned model and tokenizer are saved at `project_part2_models/` for future inference.
- `project_part2_assemble_dataset.py`: This script processes the raw dataset to generate JSONL files for fine-tuning. It contains a `get_problem_data` function to help construct the input prompts in the dataset. The script uses templates from `prompts.py` and extracts buggy code and corresponding outputs from `project_part2_dataset_training_raw.json`. The `generate_train_dataset` function assembles the training dataset. The contents of the resulting JSONL files are shuffled to ensure robust training.
- `project_part2_dataset_training_raw.json`: This file contains the raw data for program repair and hint generation. Each entry includes a unique problem ID, buggy code, repaired code, and hint text. It serves as the foundation for creating training datasets tailored to specific tasks, such as program repair or feedback generation.

Useful Tips for Training in Colab

The training time to fine-tune a model can take up to 2 hours.

If you would like to reuse a fine-tuned model later in a future Colab session, you can zip and then download it (see Section 3.7).

Note that Colab provides limited GPU resources for each user account. To ensure that your account does not run out of GPU resources, we recommend you close your Colab instance when not using GPU resources (e.g., you could download generated repairs/hints locally for annotations).

3.2 Part 2.a: Fine-tuning Baseline Models [1 Point]

For the following questions, first modify and run the script to assemble data for fine-tuning, creating separate datasets for program repair and feedback generation. Next, fine-tune the Phi-3-mini (4-bit

quantized version) model using the program repair dataset to obtain Phi-3-SFT-Repair-r16-alpha32 and the feedback generation dataset to obtain Phi-3-SFT-Hints-r16-alpha32. Similarly to Part#1, we use Phi-3-mini (4-bit quantized version) with Unsloth libraries [5]. Report results for the following two questions in the PDF submission file – see instructions in Section 3.6.

(I.11) Modify the parameters in the `project_part2_evaluate.py` script to generate program repairs using each of the two models (i.e., Phi-3-SFT-Repair-r16-alpha32 and Phi-3-SFT-Hints-r16-alpha32). Report *RPass* and *REdit*[1].

(I.12) Modify the parameters in the `project_part2_evaluate.py` script to generate hints using each of the two models (i.e., Phi-3-SFT-Repair-r16-alpha32 and Phi-3-SFT-Hints-r16-alpha32). Evaluate the generated hints using the *HGood* metric [3] and report your results.

Useful Tips for Part 2.a

`project_part2_assemble_dataset.py`: You can modify the mode variable inside the `main` function to obtain the dataset for either program repair or hint generation.

`project_part2_finetuning.py`: You can modify the dataset used for training in the `main` function of the script, by replacing `«PATH_TO_THE_DATASET»`. Save your fine-tuned model with appropriate naming by changing the `output_dir` variable.

`project_part2_evaluate.py`: You should modify the `model_selected` variable inside the `main` function to point to the directory where your newly fine-tuned model is stored. You should also change the `mode` variable to do either repair or hint generation

When reporting quality metrics for hint generation, you should report all of *HCorrect*, *HInformative*, *HConceal*, *HComprehensible*, and *HGood*.

3.3 Part 2.b: Fine-tuning for Program Repair with Varying LoRA Parameters [1 Point]

For the next questions, you will fine-tune the Phi-3-mini for program repair using different LoRA parameter configurations: $(r, \alpha) = \{(4, 8), (16, 32), (64, 128)\}$. Note that $(r, \alpha) = (16, 32)$ model setting is the same as that used in Part 2.a. Use the provided script to train models with each configuration and evaluate their performance on program repair. Report results for the following two questions in the PDF submission file – see instructions in Section 3.6.

(I.13) Evaluate the three different fine-tuned models with varying (r, α) on program repair. Report program repair quality metrics (*RPass*, *REdit*), *TrainingTime*, and *TrainingMemory*. In terms of quality metrics, compare the results of different fine-tuned models with those of the base Phi-3-mini in I.3 of Project Part#1.

(I.14) Analyze the tradeoffs between LoRA parameter configurations in terms of program repair quality and *TrainingMemory*. Include a discussion about the *TrainingMemory* and the number of trained parameters for each configuration (printed in the console when starting training).

Useful Tips for Part 2.b

`project_part2_finetuning.py`: You can modify the LoRA parameters `lora_r` and `lora_alpha` in the `main` function of the script. Upon completion, the script prints the total training memory used among other useful details.

3.4 Part 2.c: Fine-tuning on Merged Datasets [6 Points]

Next, you will create a unified dataset and fine-tune a single model capable of performing both program repair and feedback generation. Select the same LoRA parameter configuration as in Part

[1]You can report your results using tables. More details about how to create tables in LaTeX: https://www.overleaf.com/learn/latex/Tables

2.a (i.e., $(r, \alpha) = (16, 32)$). Use the provided script to fine-tune the multi-task model and evaluate its performance. Report results for I.15, I.16, and I.17 in the PDF submission file – see instructions in Section 3.6.

(I.15) Evaluate the multi-task model on program repair. Report results based on the program repair quality metrics as done in I.11.

(I.16) Evaluate the multi-task model on hint generation. Report results based on the hint generation quality metrics as done in I.12.

(I.17) Report and analyze the performance w.r.t. *TrainingTime* and *TrainingMemory*. Explain the approach you used to create the dataset and fine-tune the multi-task model, noting challenges and key observations. Discuss the performance changes compared to specialized models in Part 2.a and the possible reasons behind them.

(I.18) Provide the complete code used for fine-tuning in the ZIP file, following the instructions in Section 3.6 (include the following files: `project_part1_evaluate.py`, `project_part1_repair.py`, `project_part1_hint.py`, `project_part1_utils.py`, `project_part1_prompts.py`, `project_part2_evaluate.py`, `project_part2_assemble_dataset.py`, and `project_part2_finetuning.py`).

Useful Tips for Part 2.c

`project_part2_assemble_dataset.py`: You can add a new function here that prepares a combined dataset for both program repair and hint generation.

You should evaluate the same multi-task model for both program repair and hint generation.

When reporting quality metrics for hint generation, you should report all of *HCorrect*, *HInformative*, *HConceal*, *HComprehensible*, and *HGood*.

3.5 Grading Instructions

The grading of the project will account for the following criteria:

- Obtaining expected results for the quality metrics after fine-tuning the models on the provided datasets (exercises I.11, I.12, I.13, I.15, and I.16).
- Analyzing and discussing the tradeoffs between LoRA configurations in detail, e.g., using a plot (exercise I.14).
- Describing the approach taken and possible reasons for performance changes with full details in a clear way (exercise I.17).
- Submitting the complete version of the modified code (exercise I.18).
- Using the required styling files for preparing the report PDF, appropriately structuring the report, and submitting the files in the right format (see submission instructions).

3.6 Submission Instructions

Please submit the following files using your personal upload link:

- **<Lastname>_<Matriculation>_project2.pdf**. This PDF file will report answers to the following questions: I.11, I.12, I.13, I.14, I.15, I.16, and I.17. The PDF file should be generated in LaTeX using NeurIPS 2024 style files. The PDF file size should **not exceed** 1**mb**. Please use the following naming convention: Replace <Lastname> and <Matriculation> with your respective Lastname and Matriculation number (e.g., Singla_1234567_project2.pdf).
- **<Lastname>_<Matriculation>_project2.zip**. This ZIP file will report answers to question I.18. Importantly, the ZIP file should unzip to a folder named **<Lastname>_<Matriculation>_project2**. Inside this folder, include the mentioned files. The ZIP file size should **not exceed** 5**mb**. Replace <Lastname> and <Matriculation> with your respective Lastname and Matriculation number (e.g., Singla_1234567_project2.zip).

3.7 Technical Instructions

Please refer to the following technical instructions:

- You can use the `!unzip project_part1_datasets.zip` command to unzip datasets in Colab.
- You can use the `!zip -r «output_file».zip «path_to_the_model_folder»` command to zip your model before downloading for future use.
- To use GPU-based resources, refer to the technical instructions in Week 3.
- To install specific requirements, refer to the technical instructions in Week 4.

References

[1] Victor-Alexandru Padurean and Adish Singla. Benchmarking Generative Models on Computational Thinking Tests in Elementary Visual Programming. In *NeurIPS (Datasets and Benchmarks Track)*, 2024. Paper link: https://openreview.net/pdf?id=q2WT19Ciad.

[2] Tung Phung, Victor-Alexandru Padurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation. In *LAK*, 2024. Paper link: https://dl.acm.org/doi/pdf/10.1145/3636555.3636846.

[3] Nachiket Kotalwar, Alkis Gotovos, and Adish Singla. Hints-In-Browser: Benchmarking Language Models for Programming Feedback Generation. In *NeurIPS (Datasets and Benchmarks Track)*, 2024. Paper link: https://openreview.net/pdf?id=JRMSC08gSF.

[4] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-Factoring Based Program Repair Applied to Programming Assignments. In *ASE*, 2019. Paper link: https://mechtaev.com/files/ase19.pdf.

[5] Daniel Han, Michael Han, and Unsloth team. Unsloth, 2023. http://github.com/unslothai/unsloth.