Course Project [18 Points]

Generative AI

Saarland University – Winter Semester 2024/25

Part 1 Submission due: 20 January 2025 by 6pm CET
Part 2 Submission due: 3 February 2025 by 6pm CET

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; Released: 7 January 2025; Due: 20 January 2025, 6 PM CET]. 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; Released: 21 January 2025; Due: 3 February 2025, 6 PM CET]. 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 metric *TrainingTime* that captures the training time of the fine-tuning process.

2 Project Part#1 [10 Points]

As mentioned above, the first part of the project focuses on evaluating the baseline models and exploring prompt engineering techniques for program repair and hint generation.

2.1 Provided Datasets and Scripts

We will use the INTROPYNUS dataset [3, 4] for evaluation, consisting of tuples with a programming task T, a buggy program P_b, repaired program P_r, and test suite Ω_T. There are 5 tasks, each with 5 buggy programs. To facilitate the experiments, we have prepared a set of initial scripts and data files. More concretely, we have provided the following files in `project_part1.zip`:

- `project_part1_evaluate.py`: This script coordinates the evaluation process for program repair and hint generation. It includes two key classes. The `ProgramEvaluation` class manages the evaluation of individual programs, with methods such as `get_and_evaluate_repair` for generating and evaluating repairs, and `get_hint` for generating hints. The `DatasetEvaluation` class handles dataset-wide processing, with methods like `evaluate_programs_in_folder` for evaluating all programs in a task folder, `save_evaluation_results` for storing results in JSON format, and `evaluate_dataset` for orchestrating the entire evaluation process. Upon running, the final results are stored in `project_part1_evaluation_results/`. For debugging purposes, all prompts and responses are saved in `project_part1_transcripts/`.
- `project_part1_repair.py`: This script focuses on repairing buggy programs. It defines the `Repair` class, which provides several methods, including `generate_repair` to create prompts and query the model for repairs, `extract_fixed_code` to parse and extract repaired programs from model responses, `call_llm_openai` to query OpenAI models, `call_llm_huggingface` to query HuggingFace models, `call_llm` as a wrapper for the previous two functions, and `save_transcript_to_json` for recording input prompts and the generations.
- `project_part1_hint.py`: This script manages hint generation. It includes the `Hint` class, which has methods like `generate_hint` to create prompts and query the model for hints, `extract_hint` to parse and extract hints from model responses, textttcall_llm_openai to query OpenAI models, `call_llm_huggingface` to query HuggingFace models, `call_llm` as a wrapper for the previous two functions, and `save_transcript_to_json` for recording input prompts and the generations.
- `project_part1_utils.py`: This utility script provides essential functions for program execution and evaluation. The `Compiler` class includes `run_program` to execute Python code and return outputs or error messages, and `extract_error_message` to extract relevant errors. The `Distance` class offers methods such as `get_edit_distance` to compute token-level differences between programs and `program_to_essential_tokens` to preprocess programs by removing unnecessary tokens.
- `project_part1_prompts.py`: This script defines the prompts used for interacting with generative models. It includes `system_message_nus` for configuring the model as a tutor, `user_message_nus_repair_basic` for creating repair-generation prompts, and `user_message_nus_hint_basic` for generating hint prompts.
- `project_part1_datasets.zip`: This file will be unzipped into a directory `project_part1_datasets/` containing two directories, namely `problems/` and `evaluation_data/`.[1] The `problems/` directory contains JSON files, each representing an individual programming problem. Each JSON file includes a textual problem description, the buggy program, the correct repaired program, and a set of test cases with inputs and expected outputs. The `evaluation_data/` directory provides the detailed evaluation setup for each problem. It is organized into problem-based folders, where each folder corresponds to a specific programming problem (e.g., `1_sequential-search`). Inside each problem folder, subfolders (e.g., `prog_1`) contain the buggy and repaired Python files (`buggy.py` and `fixed.py`) for the individual problems.

To run the scripts with GPT-4o-mini, you will first need to upload the file containing your OpenAI API key to Colab, then change the `«OPENAI_API_KEY_FILE»` placeholder in `project_part1_repair.py` and `project_part1_hint.py` with its actual name.

[1] Note that you can upload and unzip the file directly on Colab – see Section 2.7.

2.2 Part 1.a: Basic Prompts [2 Points]

For these questions, you will work on understanding the provided scripts, data, and evaluation criteria. We will use GPT-4o-mini[2] and Phi-3-mini (4-bit quantized version)[3]. Note that we use Unsloth libraries [5] for Phi-3-mini, which provide better inference for quantized models. Report results for the following four questions in the PDF submission file – see instructions in Section 2.6.

(I.1) Modify the parameters in the `project_part1_evaluate.py` script to query GPT-4o-mini with the basic prompt to generate program repairs and run the script. Report *RPass* and *REdit*.[4]

(I.2) Modify the parameters in the `project_part1_evaluate.py` script to query GPT-4o-mini with the basic prompt to generate hints and run the script. Evaluate the generated hints using the *HGood* metric [3] and report your results.

(I.3) Repeat the steps in I.1 for Phi-3-mini and report your results.

(I.4) Repeat the steps in I.2 for Phi-3-mini and report your results.

Useful Tips for Part 1.a

`project_part1_evaluate.py`: Inside `__main__`, you can change between `gpt-4o-mini` and `phi-3-mini` models by modifying the `model_selected` variable. In the same file, you can change the `mode` variable to do either repair or hint generation.

For I.1 and I.3, use the automatically computed *RPass* and *REdit* metrics. You should report the averages per problem and overall.

For I.2 and I.4, you need to manually annotate *HCorrect*, *HInformative*, *HConceal*, and *HComprehensible* in order to obtain *HGood* [3]. Include all of them in your report. You should report the averages per problem and overall.

2.3 Part 1.b: Improved Program Repairs [2 Point]

For these questions, you will focus on improving the repair rate by using multiple queries and picking the best repair. Report results for the following two questions in the PDF submission file – see instructions in Section 2.6.

(I.5) Modify the scripts to query GPT-4o-mini for generating $n = 3$ repair candidates with a temperature of 0.7 and select the best candidate. Next, run the scripts and report *RPass* and *REdit*.

(I.6) Repeat the steps in I.5 for Phi-3-mini and report your results.

Useful Tips for Part 1.b

`project_part1_repair.py`: You need to modify the `generate_repair` function to generate 3 repairs and pick the closest correct repair. Finally, the modified `generate_repair` function should return the closest correct repair. You should change the temperature parameter when querying GPT-4o-mini to 0.7 in the `call_llm_openai` function. Similarly, when querying Phi-3-mini, you should enable sampling and set the temperature to 0.7 in the `call_llm_huggingface` function.

For checking whether a repair is successful and compute its edit distance, you can use the `run_program_with_testcases` and the `get_edit_distance` functions from the `utils.py` file.

Carefully debug your code before running as any OpenAI query incurs costs!

[2] GPT-4o-mini: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

[3] Phi-3-mini: https://huggingface.co/unsloth/Phi-3-mini-4k-instruct. Note that the model will loaded as a 4-bit quantized version for inference/training using the flag `load_in_4bit = True` [5].

[4] You can report your results using tables in LaTeX: https://www.overleaf.com/learn/latex/Tables.

2.4 Part 1.c: Advanced Workflow [6 Points]

Next, we aim to further improve the quality of hints by combining techniques from the previous questions. First, we can use $n = 3$ repairs and get the best repair. Then, we can use the repair to generate an explanation along with a hint, where the explanation will act as Chain-of-Thought. Report results for I.7, I.8, and I.9 in the PDF submission file – see instructions in Section 2.6.

(I.7) Modify the required scripts to improve the quality of the hint generation process of GPT-4o-mini. Evaluate the generated hints using the *HGood* metric [3] and report your results.

(I.8) Repeat the steps in I.7 for Phi-3-mini and report your results.

(I.9) Explain the approach you took to improve the results of the models in detail. Mention some of the limitations of this approach and ideas of how it could be further improved it future works.

(I.10) Provide the modified code files (i.e., `project_part1_evaluate.py`, `project_part1_repair.py`, `project_part1_hint.py`, `project_part1_utils.py`, and `project_part1_prompts.py`) in the ZIP file, following the instructions in Section 2.6.

Useful Tips for Part 1.c

`project_part1_prompts.py`: You can create a variable with a new prompt with a repair placeholder, and use additional prompt engineering to generate explanations and better hints.

`project_part1_hint.py`: You should change the `__init__` function of the `Hint` class to use your new prompt. Additionally, modify the `generate_hint` function to make use of your previous code for generating repairs. Don't forget to add the newly added repaired program to the prompt when calling the `call_llm` function.

Carefully debug your code before running as any OpenAI query incurs costs!

2.5 Grading Instructions

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

- Obtaining expected results for the quality metrics on the provided datasets (exercises I.1–I.8).
- Describing the approach taken with full details in a clear way (exercise I.9).
- Submitting the complete version of the modified code (exercise I.10).
- 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).

2.6 Submission Instructions

Please submit the following files using your personal upload link:

- **<Lastname>_<Matriculation>_project1.pdf**. This PDF file will report answers to the following questions: I.1, I.2, I.3, I.4, I.5, I.6, I.7, I.8, and I.9. 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_project1.pdf).
- **<Lastname>_<Matriculation>_project1.zip**. This ZIP file will report answers to question I.10. Importantly, the ZIP file should unzip to a folder named **<Lastname>_<Matriculation>_project1**. 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_part1.zip).

2.7 Technical Instructions

Please refer to the following technical instructions:

- You can use the `!unzip project_part1_datasets.zip` command to unzip datasets in Colab.
- 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.
- To use OpenAI API key, refer to the technical instructions in Week 6.

3 Project Part 2

Part 2 of the project will be released on 21 January 2025, after the submission deadline for Part 1.

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.

Based on template from https://media.neurips.cc/Conferences/NeurIPS2024/Styles.zip