BTP-1 Report
Subjective Autograding of Programming Assignments

Margav Savsani
200050072

November 27, 2023

Contents

1 Introduction

Automated evaluations, commonly based on test cases, are widely employed to assess the functional correctness of programming assignments submitted by students in programming courses. While these evaluations ensure precise functionality assessment, there is a compelling need for qualitative feedback to enhance students' learning experiences while ensuring that the grading process remains automated. This includes feedback on the efficiency of the algorithm, code readability, modularity, robustness, etc.

This project aims to develop an AI-assisted feedback and evaluation system for programming assignments.

2 Background and Motivation

In our internal exploration, prior to my involvement, open-source models like CodeBERT and GraphCodeBERT were tested for tasks such as Algorithm Identification within code. However, the approach faced notable challenges. Specifically, separate models were necessary for each task, and these models struggled with very long inputs—a crucial requirement in our context.

To address these issues, we are thus considering the use of Large Language Models (LLMs). These models boast the ability to support extensive context lengths, proving beneficial in scenarios where lengthy inputs are necessary. Furthermore, their versatility allows a single LLM to potentially grade various types of problem statements. This shift towards LLMs presents an opportunity to overcome the limitations we faced with previous approaches, offering a more streamlined and efficient solution for our specific needs.

3 Key Components

- **Grading Submission**: Retrieve a 'suitable' set of (Problem Statement, Rubric Criterion, Submission, Grade) from an existing database. A 'suitable set' is defined as having the same rubric criterion and offering good diversity in grades. Utilize this set as few-shot examples for the Large Language Model (LLM) to generate grades for new submissions (Problem Statement, Rubric Criteria, Submission). Note that for entirely new programming assignments or criteria, the instructor can manually grade a few submissions to provide this set.
- **Rubric Criteria Suggestion**: Provide rubric criteria suggestions (such as well-commented code, code complexity, etc.) to instructors who input a Problem Statement into the model.

4 Model Survey

We conducted a comprehensive survey of all available open-source Large Language Models (LLMs) to determine the most suitable starting point for our project. Initially, we identified Llama2 as the optimal choice and commenced our work using it. However, during the project's course, Code Llama was released, prompting us to shift our focus to this newer model.

4.1 Llama 2: Open Foundation and Fine-Tuned Chat Models

Llama 2 represents an upgraded version of Llama 1, trained on a new mix of publicly available data. Notable improvements include a 40% increase in the pre-training corpus size, a doubled model context length, and the adoption of grouped-query attention. Various versions of Llama 2 with 7B, 13B, and 70B parameters are available.

Additionally, Llama 2-Chat is a fine-tuned variant optimized for dialogue use cases, featuring versions with 7B, 13B, and 70B parameters.

4.2 Code Llama: Open Foundation Models for Code

Code Llama, a family of large language models for code based on Llama 2, offers state-of-the-art performance among open models. It excels in infilling capabilities, supports large input contexts, and boasts zero-shot instruction-

following abilities for programming tasks. Variants include Code Llama, Code Llama - Python, and Code Llama - Instruct, each with 7B, 13B, and 34B parameters.

Code Llama - Instruct variants are further fine-tuned on a mix of proprietary instruction data to enhance safety and helpfulness. This refinement significantly improves performance on various benchmarks related to truthfulness, toxicity, and bias, with a moderate impact on code generation performance.

I reviewed the original paper and the corresponding presentation can be found **here**.

5 Code Generation

The tasks of code generation and grading programming assignments share similarities. It is reasonable to expect that a model capable of generating more accurate code would excel in grading tasks and vice versa.

5.1 Using Pseudo Codes

For this task, we identified relevant existing datasets, including:

- **NL2Bash**: Contains 12k one-line Linux shell commands and their natural language descriptions.
- **PseudoGen Django**: A Django-based website where each line of code has an associated comment/pseudocode.
- **WikiSQL**: Involves the generation of SQL queries from given natural language and table schema.
- **Spoc Dataset**: Provides 18k programs for Codeforces problems, complete with test cases and pseudocodes.

5.2 Using Research Paper Pseudo Codes

We also explored the innovative task of using pseudocodes from research papers to generate code implementing the techniques and algorithms described in those papers. In the absence of existing datasets, a feasibility analysis was conducted, involving:

- Reviewing various research papers related to Data Structures and Algorithms.
- Noted that while most papers include pseudocodes, they often lack links or references to corresponding code implementations.
- Identified that the pseudocode often lacks sufficient information, such as descriptions of notations used, for effective code generation.
- Found papers with both pseudocode and code implementation links, but not in easily extractable formats.
- Recognized that some code implementation repositories have numerous files and folders, presenting challenges in testing the correctness of the generated code.

6 Rubric Criteria Suggestion

- Searched for relevant existing datasets to aid in fine-tuning models for the task.
- Discovered a LeetCode dataset containing problem statements and corresponding tags relevant to their solutions. These tags align with keywords that could be used in a rubric criteria.

7 Grading Submission

7.1 Approach

We aim to fine-tune the Code Llama - Instruct model for our task, exploring Meta-Learning and approaches such as Lora and Qlora for the fine-tuning process. A presentation outlining potential fine-tuning approaches (and how to adapt them for low-memory servers) for Large Language Models (LLMs) is available **here**.

Subsequently, we will assess the performance of these enhanced models in comparison with various baselines, including ChatGPT and Code Llama - Instruct before fine-tuning, as well as other open-source LLMs. Additionally, our study will delve into the effects of Prompt Engineering and the provision of few-shot examples on model performance.

7.2 Implementation

Implemented and configured inference for Llama2 (HuggingFace model) on our local DGX servers. Upon transitioning to Code Llama, established its corresponding HuggingFace model, and configured the official Code Llama model released by Meta.

7.2.1 Challenges Faced

- Navigating code-related issues with LLMs proved challenging due to the relatively limited online resources and support available.
- The HuggingFace Code Llama model presented unexpected and peculiar outputs for longer contexts, leading to an exploration of the official model released by Meta.
- Finetuning the official Code Llama model poses difficulties, requiring the implementation of all explored approaches from scratch.
- While addressing issues with the Code Llama HuggingFace model, discovered that despite being Instruction Finetuned, there is a specific format for the overall prompt (not user prompt) crucial for the model to understand and perform effectively. This underscores the significance of prompts and prompt engineering in the realm of LLMs.

7.3 Results

In our pursuit of autograding with no manual intervention, it is imperative for LLMs to precisely adhere to the provided output format instructions. Moreover, they must have the capability to support and process extensive contexts or prompts, encompassing various elements such as Problem Statements, Instructor Model Solutions, Rubric Criteria, Student Submission code files, and Few Shot Examples.

- While LLMs have the capacity to handle very large prompts (context length), they tend to forget the context in the case of larger prompts. Consequently, they struggle to strictly adhere to output format instructions
- Fragmenting information into separate parts or components and presenting them conversationally proves more effective than providing the complete information in a single prompt.
- The structure of the prompt significantly influences the model's ability to follow instructions.
- Notably, the Code Llama - Instruct model shows a relatively lower proficiency in adhering to instructions compared to ChatGPT.

  For illustrative examples highlighting the aforementioned points, refer to the example conversations **here**.

7.4 Future Work

Our next objective involves refining the prompt to enhance the Code Llama - Instruct's adherence to the output format requirements. Following this optimization, we will proceed with the subsequent steps outlined in the

approach, including fine-tuning and performance evaluation.

8 CS 293 (Data Structures Lab) Dataset

A dataset was curated from the Lab Assignments of the CS293 course offering for the batch of 2025. This dataset encompasses the problem statement, instructor's model solution, student submissions, and corresponding grades for a total of 12 assignments, each with an average of around 180 submissions.

The primary task involved anonymizing the grades and student submissions, and organizing the files in a manner conducive to easy utilization as a dataset.

8.1 Implementation Details

- Utilized SHA-256 based HMAC to anonymize the roll numbers.
- Generated a comprehensive list of hashes derived from the data of all assignments and then assigned each hash a student ID from (1 to 196) randomly.
- For each student submission, compiled all relevant submitted files into a folder, ensuring nothing was left in compressed form.

8.2 Challenges Faced

- In some labs, the submission format was not provided, and even when available, most students did not adhere to it, complicating the automation of tasks.
- Dealing with a variety of compressed files submitted by students, including recursive extraction (compressed files within compressed files). Some files posed issues during automated extraction, requiring manual intervention, such as a true zip file named as .tar.gz.
- Instances were encountered where files could not be extracted programmatically but were extractable using the OS extract feature, or were extractable in Linux but not in Windows, or were entirely unextractable. These cases necessitated manual handling.
- Students submitted numerous extra and irrelevant files, such as executables and VSCode internal files, requiring removal.
- For files like .txt and README, manual checks were necessary to remove student credentials if present.
- Grades provided by the instructor, hashed for anonymity, contained discrepancies due to extra spaces and case variations in some roll numbers, resulting in multiple hashes for the same student. Detection and correction of such cases were essential.
- Inconsistencies between grades and submissions, such as some students receiving a grade without a corresponding submission or vice versa, had to be detected and rectified to ensure the dataset's consistent structure.