Fine-Tuning LLMs for Code Grading

PROJECT SUPERVISOR

Dr. Jawwad Shamsi

PROJECT TEAM

Muhammad Moaaz bin Sajjad K20-0154
Saad bin Khalid K20-0161
Syed Abdul Rehman K20-0239

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

FAST SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

KARACHI CAMPUS

May 2024

Project Supervisor	Dr. Jawwad Shamsi	
Project Team	Muhammad Moaaz bin Sajjad	K20-0154
	Saad bin Khalid	K20-0161
	Syed Abdul Rehman	K20-0239
Submission Date	May 16, 2024	

Dr. Jawwad Shamsi _____

Supervisor

Dr. Zulfiqar Ali Memon _____

Head of Department

FAST SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

KARACHI CAMPUS

Acknowledgement

We are thankful to Allah and His blessings. We would like to express our utmost gratitude to our supervisor, Dr. Jawwad Shamsi, for his enthusiasm, patience, insightful comments, helpful information, practical advice, and unceasing ideas that have helped us tremendously at all times especially during the research and development of this project. His immense knowledge, profound experience, and professional expertise in Artificial Intelligence and Machine Learning has enabled us to complete this project successfully. Without his support and guidance, this project would not have been possible.

We are also thankful to our FYP jury and FYP committee for their guidance at each evaluation.

Abstract

The ever-increasing popularity of computer science programs is placing a significant strain on traditional methods of grading programming assignments. Manual grading by professors and teaching assistants is cumbersome, time-consuming, and prone to errors. This can lead to inconsistency in marking, frustration for students, and in worst cases, accusations of bias levied against the teacher. The growing number of computer science students exacerbates this problem, creating a pressing need for a more efficient and reliable solution. We propose an innovative automatic AI-based code grader that addresses these challenges head-on.

To implement this code grader, we first generated a large dataset using Gemini. The dataset comprised C/C++ codes (both correct and incorrect) of questions common in introductory programming courses. We then fine-tuned CodeGemma on this dataset.

Instead of manual checking, all the teachers will have to do is visit our website and provide a model answer. The student code will be compared with this model answer and graded on the basis of its logical and semantic similarity with the model solution.

We hope that our program eases the task of teachers and goes on to revolutionize the checking of programming assignments.

Table Of Contents

List of Figures

List of Tables

Introduction

Programming assignments are common in any Computer Science course. Most of the time, teachers check these assignments manually or employ a Teacher Assistant (TA) to do so. This is not only cumbersome and time consuming, but also prone to errors. Moreover, there is often a lack of consistency in marking which leads to frustration, complaints, and, in worst cases, accusations of bias levied against the teacher.

With the number of students pursuing Computer Science increasing every year, this challenge continues to get worse. To address this growing problem, we have devised an automatic AI based code grader. This will not only save time, but also produce consistent marking and reduce the need for Teacher Assistants.

To implement this code grader, we first generated a large dataset using Gemini. The dataset comprised C/C++ codes (both correct and incorrect) of questions common in introductory programming courses such as Programming Fundamentals, Object Oriented Programming, and Data Structures. We then fine-tuned the state-of-the-art LLM, CodeGemma on this dataset.

Once the model was trained, we developed a web application to act as an interface for teachers to access our system.

Instead of manual checking, all the teachers will have to do is visit our website and provide a model answer. The student code will be compared with this model answer and graded on the basis of its logical and semantic similarity with the model solution.

We hope that our program eases the task of teachers and goes on to revolutionize the checking of programming assignments.

Related Work

To better design and implement our solution, we began by first studying some already existing solutions related to computing the similarity between a pair of code snippets. Given below is a list of related work we studied.

1. Contrastive Code Representation Learning: Paras Jain [1]

Summary

The research paper titled "Contrastive Code Representation Learning" introduces ContraCode, a groundbreaking contrastive pre-training task designed to learn code functionality rather than its form.

Addressing limitations observed in reconstruction-based models like RoBERTa, which exhibit sensitivity to source code edits even when preserving semantics, ContraCode employs a neural network trained to identify functionally similar variants of a program among non-equivalent distractors. To enhance data diversity, the authors utilize an automated source-to-source compiler for generating these variants.

The paper demonstrates the superior performance of contrastive pre-training over RoBERTa in an adversarial code clone detection benchmark, achieving a remarkable 39% improvement in AUROC. Interestingly, this enhanced adversarial robustness also translates into improved accuracy over natural code, with ContraCode showcasing a 2 to 13 percentage point increase in accuracy for tasks such as summarization and TypeScript type inference over competitive baselines.

The paper meticulously details ContraCode's architecture, code transforms, and pre-training strategies, providing comprehensive analysis and highlighting novel contributions such as compiler-based transformations as data augmentations and program representation learning based on functional equivalence, resulting in significant advancements across various code understanding tasks.

Limitations and Future Work

While ContraCode shows promising results in improving code representation learning and adversarial robustness, there are some limitations and avenues for future work.

Firstly, the current implementation heavily relies on the availability of an automated source-to-source compiler for generating functionally equivalent program variants. This may limit the applicability of ContraCode to programming languages for which such compilers are not readily available. Future research could explore alternative methods for generating diverse program variants without relying on specific compilers.

Additionally, our evaluation primarily focuses on specific code understanding tasks such as clone detection, type inference, and summarization. Further investigation is needed to assess the generalizability of ContraCode across a wider range of code analysis tasks and programming paradigms.

Finally, while ContraCode improves adversarial robustness, there is still room for exploring more sophisticated adversarial attack scenarios to further enhance the resilience of code representation models.

2. Code2Vec: Learning Distributed Representations of Code: Uri Alon [2]

Summary

The document titled "Code2Vec: Learning Distributed Representations of Code" introduces a smart way to turn pieces of code into continuous distributed vectors called "code embeddings." These vectors help predict the meaning of code by breaking it down into paths in its abstract syntax tree and learning how to represent and combine these paths.

The method is proven effective by training on a big dataset of 14 million methods and successfully predicting method names in new files. Compared to older techniques, this model shows a big improvement of over 75% in predicting method names across different projects. The authors stress the importance of good method names for understanding and maintaining code, especially in public APIs. They reveal that the learned code vectors capture similarities and relationships between method names, offering valuable insights.

The paper also discusses how these code embeddings can be used in various tasks like automatic code review, finding and understanding APIs, code classification, and measuring similarity for ranking and clone detection. The challenges addressed include finding a way to represent code that captures its meaning and can be used in different programs, along with using attention mechanisms to focus on important parts of the code during learning.

Overall, the Code2Vec model presents a promising way to create useful representations of code that can be used in many programming tasks, improving our understanding and analysis of code.

Limitations and Future Work

While the Code2Vec model discussed in this paper has shown positive outcomes in learning code representations and predicting method names, there are a few areas that need attention for future improvement.

Firstly, the model's performance may vary when used with smaller or specific datasets, as it heavily relies on having a large amount of training data.

Additionally, there's room to enhance the model's ability to understand complex relationships between different pieces of code. Future research could explore adding more details like program context or variable connections to make predictions more accurate.

Moreover, evaluating the model on various programming language tasks, such as summarizing code or detecting bugs, would offer a broader view of its capabilities. In summary, while Code2Vec looks promising, more research is necessary to address these areas and make it more useful in the field of programming languages and software engineering.

3. Learning Program Semantics with Code Representations: An Empirical Study: J.K.Siow [3]

Summary

The paper titled "Learning Program Semantics with Code Representations: An Empirical Study" systematically explores various program representation techniques for code intelligence tasks. These techniques, categorized into Feature-based, Sequence-based, Tree-based, and Graph-based, are evaluated across three prominent code intelligence tasks: Code Classification, Vulnerability Detection, and Clone Detection.

The study addresses key research questions involving a comparison of code representation techniques, understanding the impact of node embedding information in tree-based and graph-based representations, and assessing the efficacy of various graph representations.

To answer these questions, the authors utilize a range of machine learning algorithms, tailoring models to each representation technique for comprehensive analysis. The experimental outcomes highlight the consistent superiority of graph-based representations across the selected tasks. The study underscores the importance of textual information in capturing program semantics over node type information. Furthermore, it notes that while different tasks may require task-specific semantics for optimal performance, combining program semantics from diverse dimensions, such as control dependency and data dependency, can yield promising results.

The contributions of this paper lie in providing a systematic evaluation of diverse code representation techniques, offering insights into the impact of node embedding information, and evaluating the effectiveness of different graph representations. These findings hold valuable implications for researchers and practitioners engaged in addressing code-related challenges, enhancing the understanding and application of program semantics for various tasks.

Limitations and Future Work

While the empirical study outlined in this paper contributes valuable insights into the efficiency of diverse program representation techniques for code intelligent tasks, it is essential to acknowledge certain limitations.

The study focuses on a specific set of representation techniques and tasks that may potentially exclude other approaches and applications. Additionally, the evaluation relies on publicly available benchmark datasets, which may not fully encompass the complexity and diversity of real-world code scenarios.

Future endeavors could explore the amalgamation of multiple representation techniques to capitalize on their respective strengths for enhanced performance. Furthermore, delving into the impact of varied code representations on emerging code-related challenges, such as code generation or program synthesis, holds the potential for deeper understanding and advancements in the realm of program semantics learning.

4. Learning Program Embeddings to Propagate Feedback on Student code: C.Piech [4]

Summary

The paper titled "Learning Program Embeddings to Propagate Feedback on Student Code" by Chris Piech et al. confronts the challenge of providing scalable feedback in the context of massive online computer science courses.

Recognizing the limitations of traditional machine learning approaches when applied directly to program data, the authors introduce a novel neural network method for encoding student programs. This method proves successful in practical application, particularly in assessments from Code.org Hour of Code and Stanford University's CS1 course.

The paper underscores the significance of optimizing program and memory-state embedding, introducing a neural network architecture tailored to achieve this objective. Notably, the authors leverage the executability of programs to gather training data, demonstrating the effectiveness of their approach through the amplification of teacher feedback using real student data. In doing so, the paper not only contributes insights into scalable feedback mechanisms for online programming courses but also raises awareness about the potential impact of program embeddings on the landscape of computer science education. The authors advocate for further exploration in this emerging field, emphasizing its importance in shaping the future of online programming education.

Limitations and Future Work

In this study, the model presented was a neural network method for encoding programs and propagating feedback on student code in massive online classes. Even though the approach shows promising results in capturing both functional and stylistic elements of programs, it has certain limitations.

Firstly, it relies on a fixed-dimension real-valued space representation, which may not capture the full complexity of programs. Additionally, the model does not consider user-defined variables, limiting its applicability to certain programming languages.

Furthermore, the training data used in the experiments is primarily from computer science courses, and future work should explore the generalizability of the approach to other domains.

Additionally, incorporating more sophisticated techniques, such as incorporating natural language processing methods for analyzing code comments, could enhance the feedback generation process.

Overall, the work lays the foundation for further research in learning program embeddings and opens up avenues for improving feedback mechanisms in large-scale computer science education.

5. SCG_FBS: A Code Grading Model for Students' Program in Programming Education: Y.Qin [5]

Summary

In the paper titled "SCG_FBS: A Code Grading Model for Students' Program in Programming Education" by Yuze Qin et al., the authors tackle the prominent challenge of efficiently evaluating students' programming assignments at scale. Their proposed grading model, SCG_FBS, presents an innovative combination of semantic feature analysis and black-box testing, leveraging a select function to extract crucial semantic features, evaluate code, and reduce input sequence length.

Employing a pre-trained instruction embedding to convert source code into a vector sequence, the model utilizes a neural network with attention mechanisms to extract semantic features effectively. To validate their approach, the authors collect data sets from a widely used Online Judge platform for black-box testing in college settings.

The experimental results showcase the SCG_FBS model's remarkable accuracy in grading, accompanied by reduced training time compared to baseline models, thereby enhancing efficiency in the evaluation process. The paper also delves into related works in feature extraction and program grading, emphasizing the significance of their approach in

addressing the intricate task of evaluating both correct and incorrect student programs within the realm of programming education.

Limitations and Future Work

While the SCG_FBS model shows promise in grading students' programs, addressing certain limitations is crucial.

Enhancements could involve incorporating additional techniques like static analysis, accommodating diverse coding styles and languages, exploring advanced neural network architectures, and conducting extensive experiments on larger datasets.

Furthermore, assessing usability and scalability in real educational settings, alongside gathering feedback from teachers and students, would contribute to refining the grading system for practical implementation.

6. A Similarity based Assisted Grading for Introductory Programming Course: K.Fukushima [6]

Summary

The paper titled "A Similarity-based Assisted Grading for Introductory Programming Course" introduces a method to simplify the grading process in introductory programming courses.

Recognizing the challenges faced by teachers in manually grading numerous submissions, the proposed approach automates the classification of student submissions based on source code similarity. It compares new submissions with a set of manually graded ones, assigning the same grade if a similar submission is found. If no match is identified, the teacher manually grades the submission, and the assigned grade is used for future similar submissions.

This method leverages the assumption that submissions with high source code similarity deserve the same grade, reducing the grading effort. The study implements a straightforward algorithm based on this principle and evaluates its effectiveness using student submissions from a programming course.

The paper also discusses the method's limitations and its applicability to other programming courses, providing valuable insights into using source code similarity as a grading criterion and developing an iterative algorithm for automated grading in educational settings.

Limitations and Future Work

While the similarity-based assisted grading method proposed for introductory programming courses shows promise, it comes with certain limitations and potential areas for future improvement.

One limitation is its assumption of a single Python file for each submission, making it less applicable to courses involving interlinked projects or multiple programming languages. Additionally, the current binary grading system (Accept/Reject) might not adequately capture the nuances of partial credit.

To address these limitations, future work could explore extending the method to accommodate diverse programming assignments and incorporate more fine-grained grading criteria. Moreover, there is a need to further evaluate and optimize the algorithm's performance and scalability, ensuring its effectiveness with larger datasets and enhancing efficiency in real-world educational settings.

7. Semantic Approach for Increasing Test Case Coverage in Automated Grading of Programming Exercise: M. R. I. Bariansyah [7]

Summary

The paper titled "Semantic Approach for Increasing Test Case Coverage in Automated Grading of Programming Exercise" by M. Rifky I. Bariansyah, Satrio Adi Rukmono, and Riza Satria Perdana addresses the challenge of manual test case creation in automated programming assignment grading.

The authors propose an innovative solution using white-box testing and semantic analysis to automatically generate test cases with improved coverage. In this method, the evaluator provides a reference code representing the correct implementation, and the system analyzes semantic differences between this code and the student's solution to generate relevant test cases.

The implementation utilizes concolic execution, combining symbolic and concrete execution, to record and analyze execution paths. Experiments demonstrate the approach's effectiveness, showcasing its ability to generate meaningful test cases and enhance coverage compared to random test case generators. The conclusion highlights the potential of white-box testing and semantic analysis in test case generation for programming exercises, suggesting avenues for future research in this domain.

Limitations and Future Work

While the proposed approach shows promise in enhancing test case coverage for automated programming exercise grading, it's crucial to acknowledge some limitations and things needed to be done in future.

The system's reliance on a reference code may pose challenges in obtaining it or introduce biases in grading. Additionally, the effectiveness hinges on precise semantic analysis and accurate detection of path deviations and equivalences, areas that warrant further enhancement for increased reliability.

For future research, avenues include broadening language support, integrating machine learning for automated reference code generation, and optimizing test case generation. Further experiments across diverse programming assignments would offer valuable insights. Implementing feedback mechanisms for students to understand grade rationale and improve programming skills stands as a valuable extension.

Objectives

The overall project objective is to develop and implement an automated AI-based code grader system to streamline the grading process for programming assignments in computer science courses. Specifically, we aim to achieve the following objectives in this project:

- Increase grading efficiency.
- Reduce the time and effort required for grading, freeing up valuable time for professors and teaching assistants.
- Streamline the grading process, allowing professors to focus on providing more personalized feedback and guidance to their students
- Enhance student learning experience.
- Provide students with immediate feedback on their submissions, allowing for faster identification and correction of errors.
- Reduce student frustration and anxiety.
- Eliminate the possibility of human errors, inconsistent or biased grading to ensure fairness and transparency in the grading process.

Requirements

Operating Environment

The target website portal will operate in any standard website browser like Chrome, Microsoft Edge etc. The host can have any common operating system, including Windows, Linux, and macOS. Overall, the software will be designed to operate in a secure environment, with measures in place to protect sensitive data and ensure the integrity of the software.

System Constraints

1. **Software constraints**

   The system will be developed using Python libraries, and Flutter. The website portal imposes no additional constraints on the user other than a modern and standard, compatible web browser. However, for the backend server, the host machine should have Python and its updated libraries installed.

2. **Hardware constraints**

   The system will be designed to work with standard hardware platforms.

3. **Cultural constraints**

   The system will be designed to support English as the primary language.

4. **Legal constraints**

   The system will comply with all relevant legal requirements and will not support use of any illegal or unethical features or functionalities.

5. **Environmental constraints**

   The system will be designed to operate in a standard computing environment.

6. **User constraints**

   The system will be designed to be user-friendly and easily accessible to computer science professors and teacher assistants.

Assumptions and Dependencies

The deliverable system assumes the following regarding the usable environment:

- The teachers have a standard web browser installed on the machine.
- The teacher has a correct model answer available for the assignment question.

- Students have submitted their assignments and their code files are present on the teacher's machine.

External Interface Requirements

The project system primarily consists of the following three external interfaces:

1. Web Portal

The web interface is intended to be used by the teacher to input the model & student answers and visualize the results generated on the server by the AI model. The server will return the graded similarity scores for each student answer, and the portal will display the results in a user friendly way.

2. Code Files

All code files need to be present on the teacher's machine. The files will be read from the folder specified by the teacher in the web portal. The browser needs to support file reading functionality and permissions in order for the system to read them.

3. Server

The server will be responsible for listening to the web portal requests and use the deep learning model to calculate the similarity between provided code files. The server will then return the request with answer scores in JSON format.

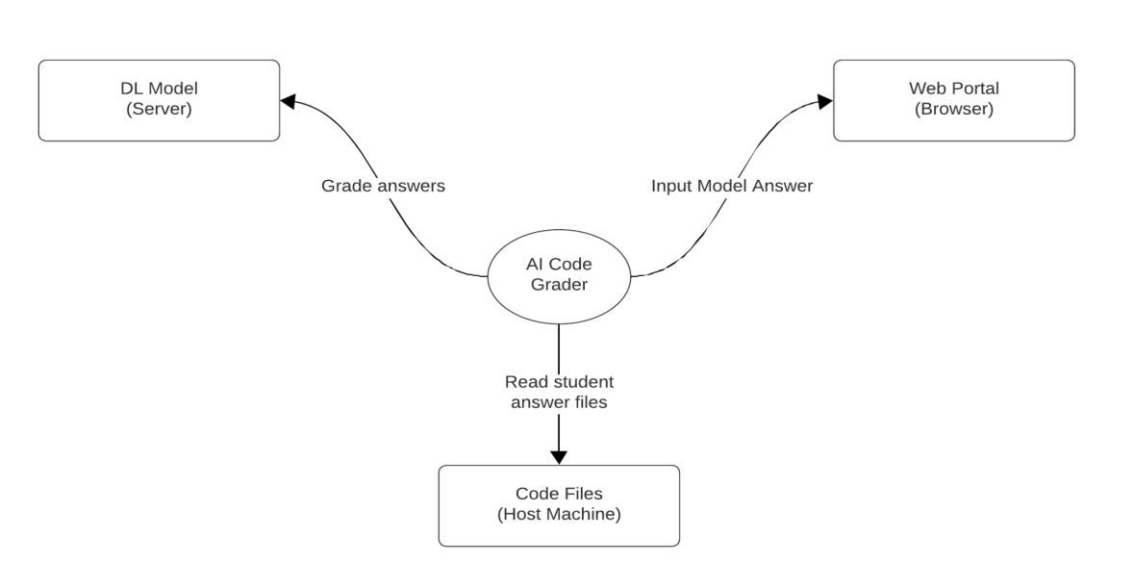


Figure 1: Context diagram of the system

Hardware Interfaces

There are no specific hardware interfaces for the web portal. The only necessity is to be able to read the code files stored on the machine.

Software Interfaces

Web Portal

The communication channel will be established only with the user browser. The website will be compiled into native browser level code i.e. HTML, CSS, and Javascript and then deployed on the user machine. Therefore, the only level of requirement is a standard operating web browser. There is no such requirement for a specific operating system. However, the browser should be able to communicate with the underlying OS in order to read the code files from the teacher's machine.

Server

The server machine needs to be able to support the latest versions of Python and its libraries. These libraries will be used to execute the AI model and generate graded scores for student submissions. Python is also used to start a listening server to be able to remotely accept requests from the web portal. These tools and libraries are better supported and accessible on a Linux platform; hence, we intend to use that for our server environment.

Communications Interfaces

Communication Channel

The only form of inter-component communication required is from the web portal to the backend server. This channel is implemented by using REST APIs through HTTPS protocol calls. This ensures that the data exchanged between the web portal and backend is transmitted over a secure channel, minimizing the risk of data interception or tampering.

Message Format

Data exchanged between the frontend and backend is in JSON format. This standardized format streamlines communication, providing a clear structure for data transmission. JSON is lightweight and easy to parse, enhancing the efficiency of information exchange.

Security

Utilizing HTTPS adds an extra layer of security by encrypting the data during transmission by TLS protocol.

Functional Requirements

Use Cases

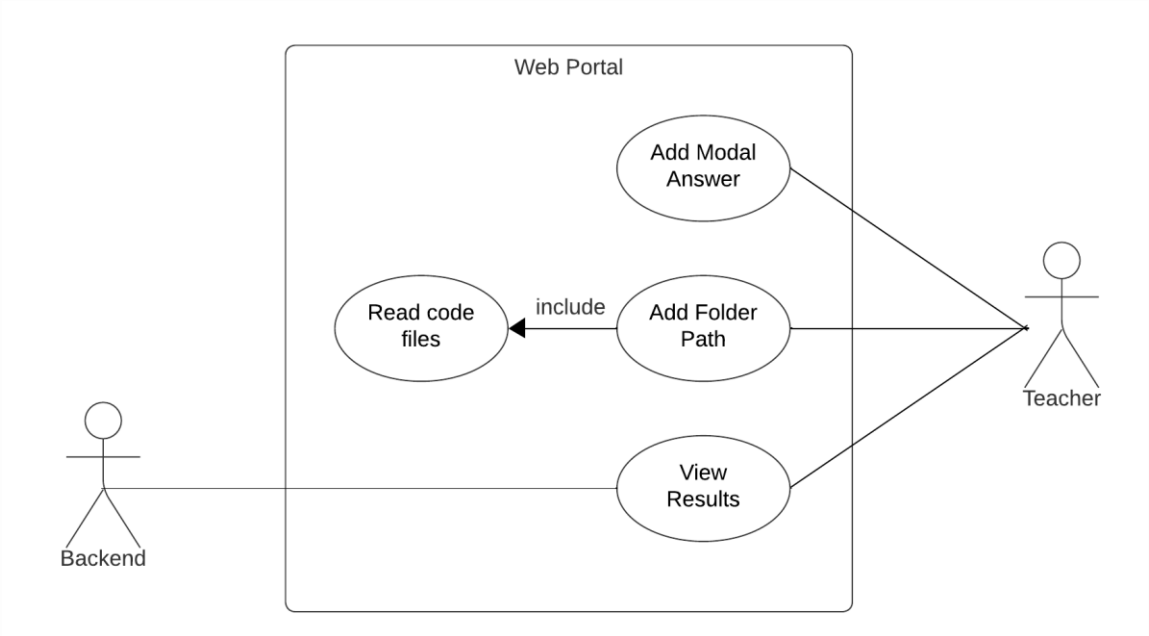


Figure 2: Use case diagram of the web portal

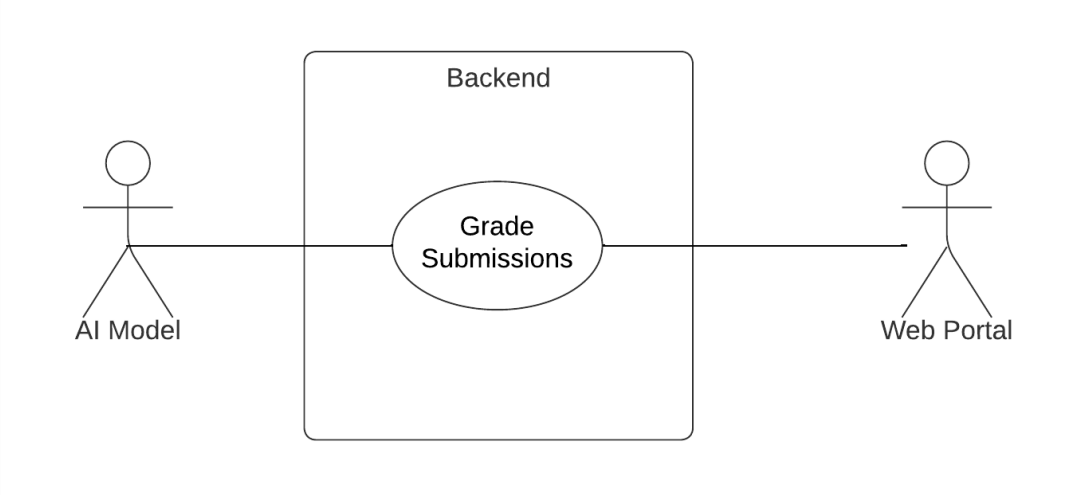


Figure 3: Use case diagram of the backend server

Add Model Answer

The teacher will be allowed to add a model code file into the system against which the students' submissions will be graded.

Table 1: Add Model Answer

1: Add Model Answer		
Use case ID	1	
Actors	Teacher	
Feature	Allow the teacher to add a model answer.	
Pre-condition	The teacher has finalized the question and its correct solution	
Scenarios		
Step	**Action**	**Software Reaction**
1.	Teacher opens the website	Model answer input field is shown
2.	Teacher clicks on pick file	The file selector is shown
Alternate Scenarios		
Step	**Action**	**Software Reaction**
1.	Invalid file extension is selected	Website does not allow selection
Post Conditions		
Step	**Description**	
1.	The code input / file is submitted saved in the memory	
Use Case Cross Referenced	None	

Add Folder Path

The teacher will select a folder where student submissions of code files are stored.

Table 2: Add Folder Path

2: Add Folder Path		
Use case ID	2	
Actors	Teacher	
Feature	Allow the teacher to input student submissions	
Pre-condition	The teacher has to submit the model answer. All solutions of assignments have been gathered and stored in the folder on the teacher's machine. Nested folders are ignored.	
Scenarios		
Step	**Action**	**Software Reaction**
1.	Teacher clicks on select folder	A folder selector is shown
Alternate Scenarios		
Step	**Action**	**Software Reaction**
1.	Teacher picks an empty folder	An error message is displayed
Post Conditions		
Step	**Description**	
1.	The selected folder is traversed to read all files	
Use Case Cross Referenced	3	

Read Code Files

The website will request the browser to read all code files from the selected folder.

Table 3: Read Code Files

3: Read Code Files		
Use case ID	3	
Actors	Teacher	
Feature	Read all code files from selected folder	
Pre-condition	The teacher has to select the folder	
Scenarios		
Step	**Action**	**Software Reaction**
1.	Teacher selects the folder	The file reading module is started
Post Conditions		
Step	**Description**	
1.	All files are stored in memory for the time being	
Use Case Cross Referenced	2	

View Results

The teacher will be able to see the graded results of each student submission.

Table 4: View Results

4: View Results		
Use case ID	4	
Actors	Teacher, Backend	
Feature	Allow the teacher to visualize and see the grades	
Pre-condition	The teacher has to select the folder. Backend has to be ready for operation	
Scenarios		
Step	**Action**	**Software Reaction**
1.	Teacher clicks on ‘Grade’ button	All data is sent to the backend for processing and loading indicator is displayed
2.	The backend responds	The graded results and metrics are displayed
Alternate Scenarios		
Step	**Action**	**Software Reaction**
1.	Internet gets disconnected	An error message is displayed
2.	Backend responds with an error	Received error is displayed
Post Conditions		
Step	**Description**	
1.	An option to grade again is shown	
Use Case Cross Referenced	None	

Grade Submissions

The backend will grade the provided code submissions against the provided model answer.

Table 5: Grade Submissions

5: Grade Submissions		
Use case ID	5	
Actors	Web Portal, AI Model	
Feature	Grade the received code submissions.	
Pre-condition	The web portal needs to send the request with data. The AI Model file needs to be present at the server.	
Scenarios		
Step	**Action**	**Software Reaction**
1.	Web portal sends grading request	The system receives the request.
2.	AI Model is invoked with code files	The grading results are computed
3.	AI Model computes the results.	The response is sent with computed grades
Alternate Scenarios		
Step	**Action**	**Software Reaction**
1.	An error occurs while processing.	The response is sent with errors.
Post Conditions		
Step	**Description**	
1.	The grading results are sent back to the web portal	
Use Case Cross Referenced	None	

Non Functional Requirements

Performance Requirements

1. **Speed:** The system must be able to grade assignments extremely fast within a maximum time frame of 1 minute per submission.
2. **Precision:** The grading accuracy of the system must be affordable with minimal false positives and false negatives.
3. **Concurrency:** The system must be able to handle simultaneous grading requests from multiple users without impacting performance.
4. **Capacity:** There is no requirement for scaling the system to a very large audience. However, the system needs to have the potential to be scaled in future if required.
5. **Safety:** The system must be able to recover from crashes and prevent data loss.
6. **Reliability:** The system must be available during working hours on all days of week.

Safety Requirements

The system must ensure the safety of the communication channel by implementing secure protocols and encryptions.

Security Requirements

The system does not need to deal with malicious activities or unauthorized access because no data of the user is stored at any stage.

Methodology

Previous Approaches

As mentioned earlier, we commenced by studying some already existing solutions related to computing the similarity between a pair of code snippets. We studied several state-of-the-art research papers (these have been discussed in detail in the related work section) and gained valuable insights. This enabled us to have a better understanding of our task, and the challenges that we may encounter.

As per the requirements of the FYP committee, we then prepared a proposal and defended our idea in front of the jury. The jury accepted our proposal without recommending any changes.

We then began to implement our project, starting off with data collection. We contacted several instructors of our university, and requested them to give us access to past Google Classrooms of Programming Fundamentals. We were successful in attaining access to 6 Google Classrooms.

However, the data in these Classrooms was messy, and required a lot of cleaning. It was a mixture of different files (source files, zips, PDFs, Word documents etc) whereas we wanted only the source files. Moreover, the source files were grouped student wise, whereas we wanted them to be grouped task wise.

Some of the cleaning was done manually whereas the rest was done using Python and BASH scripts. The cleaning involved removal of files other than source files, as well as grouping of files task wise rather than student wise. Additionally, we renamed each file to a unique ID to assist with grading. We would append underscore followed by the grade out of five to the file name after grading.

Once we began grading, we quickly realized that manual grading is too time consuming and cumbersome. To ease our task, we developed a grading app in Flutter. The grading app allowed us to iterate through the files with the mere press of the left and right arrow keys; a big improvement over manually opening and closing files one by one. It also auto formatted the code. To grade the file, we no longer had to rename it; we just pressed a number from one to five (denoting the grade) and then hit Enter, and the file automatically got graded (and renamed accordingly).

Next, we cloned ContraCode's repository on GitHub, and checked its performance on training and testing data. The results are illustrated on the next page:

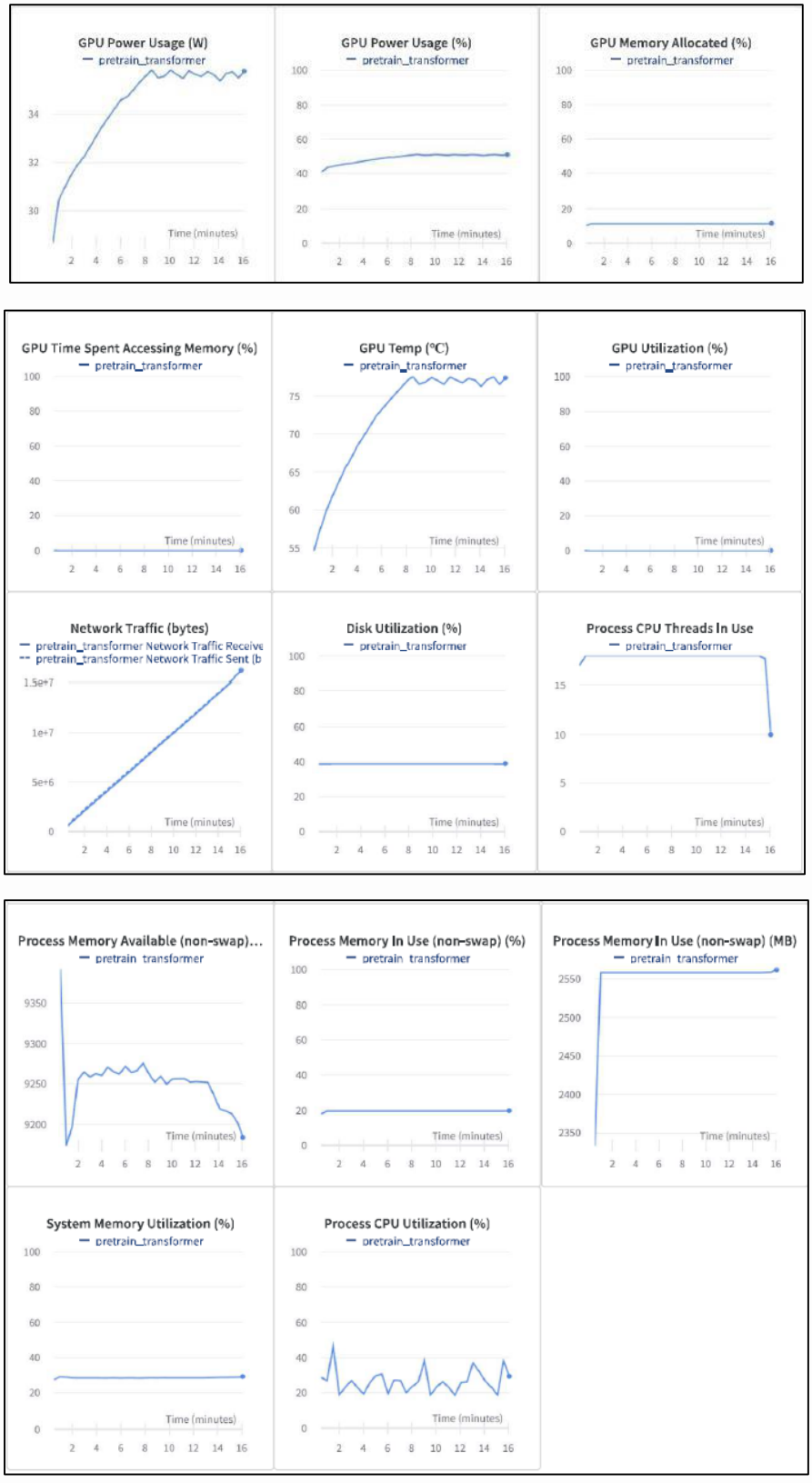


Figure 4: Model's performance on training data (before GPUs)

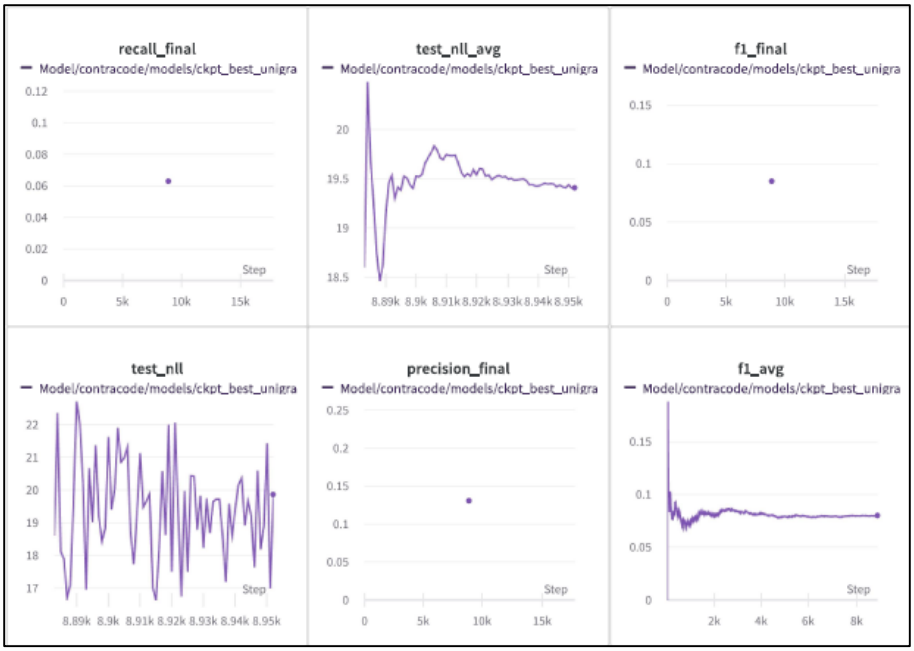


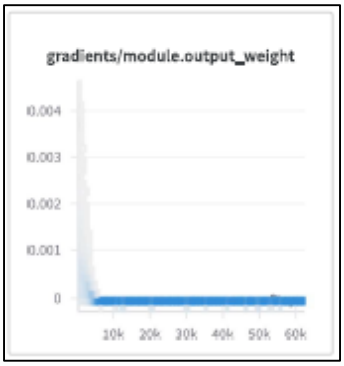
Figure 5: Model's performance on test data (before GPUs)

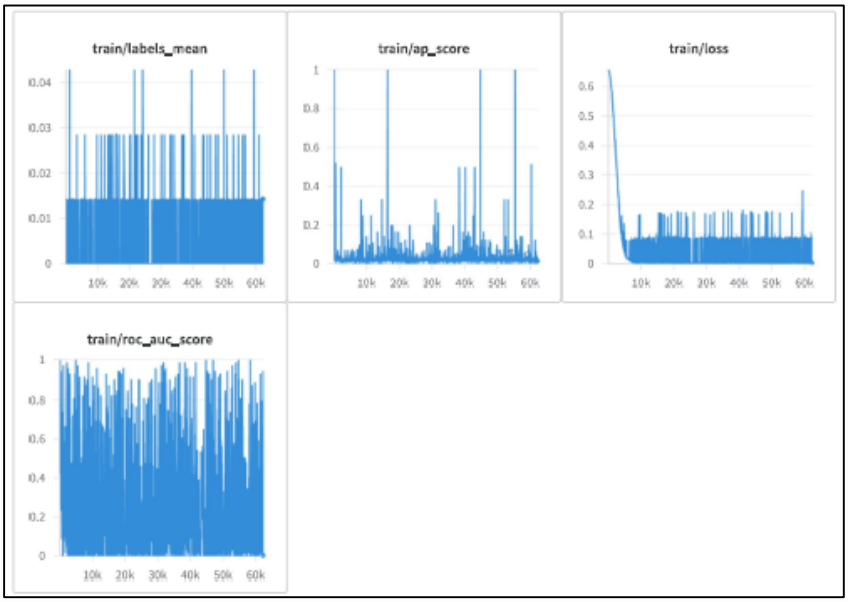
As can be clearly seen from the above graphs, the results were far from satisfactory.

After consultation with our supervisor, we decided to improve our model's accuracy by training it on a larger dataset. However, when we tried to train on larger datasets, plenty of time was being taken which was not feasible for us. So, we decided to train using GPUs of Google Cloud Platform.

When we tried to create a VM instance on Google Cloud Platform using our Student's NU account, we were unsuccessful. Therefore, we decided to buy Colab Pro Subscription for GPUs. However, even with Colab Pro, training was taking a lot of time. Moreover, the Colab session was non persistent, and there were limited compute units. Therefore, we decided to use the GPUs in our university's FYP lab. However, the PCs in the FYP lab were rarely available, and there was little reduction in the time taken for training.

As a last resort, we asked our supervisor, Dr. Jawwad, for his opinion. He advised us to use Google Cloud Platform using Teacher's NU account. We contacted IT department who created a teacher's NU account and gave us the credentials. Using this account, we were finally successful in creating a VM instance, and training was done successfully. The results are shown on the next page:





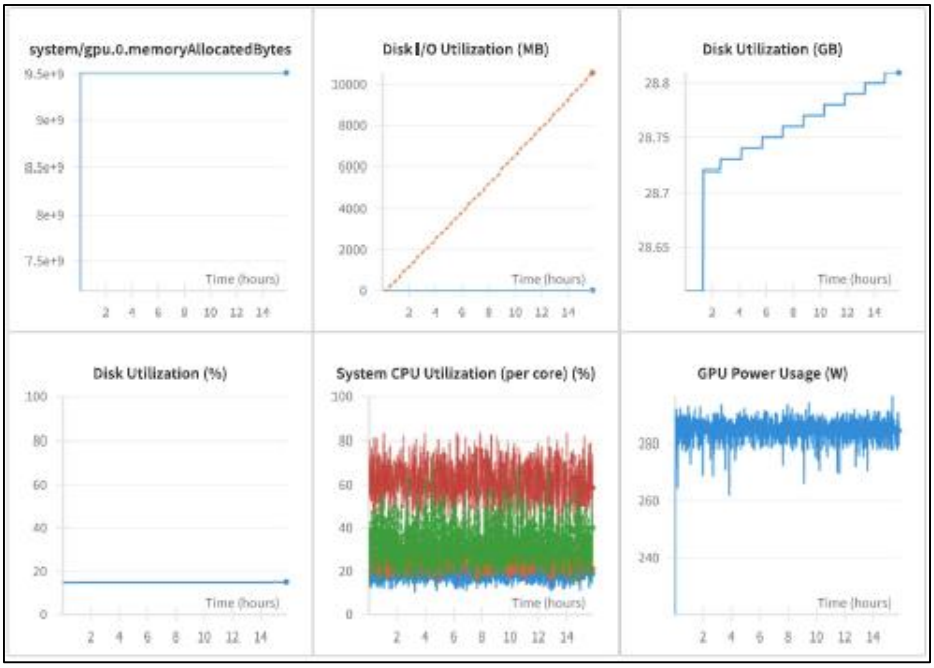


Figure 6: Model's performance after acquiring GPUs

As can be deduced from the above graphs, the results remained poor. After many attempts at fine-tuning failed to improve results, we decided to alter our approach, and turned to GraphCodeBERT — a multi-programming-lingual model pre-trained on NL-PL pairs in 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go) that considers the inherent structure of code i.e. data flow instead of taking syntactic-level structure of code like abstract syntax tree (AST). We fine-tuned GraphCodeBERT on BigCloneBench dataset. The results were given in form of 0 and 1, with 0 denoting dissimilar pairs and 1 denoting similar pairs.

Next, we decided to extend the model from binary classification to regression, so that it would output the similarity between 1 and 0 depending on the level of logical similarity between student's code and the model answer, with 1 denoting complete similarity and 0 denoting no similarity.

Even though, the results improved, they were still far from satisfactory. Our model was not sensitive enough to detect small changes in code like addition being replaced by subtraction, increment operator being replaced by decrement, the sense of an inequality being reversed etc. Moreover, the model was prone to giving high scores even for incorrect codes. For instance, if we made a change to make the student code incorrect, its score did go down but not by as much as it should have e.g the score decreased to 0.6 when a score of around 0.2 would represent a more accurate grading.

Current Approach

After the failure of GraphCodeBERT, we decided to fine-tune an LLM instead. We also decided to find a new dataset since BigCloneBench has binary class labels and is in Java while our work focuses on C, whereas the data in Google Classrooms was not of a good quality.

Dataset

Taking inspiration from the GPTCloneBench paper, we decided to generate our own dataset from Gemini. After much experimentation, we settled on the below prompt:

```
final prompt = '''
Task:
Generate a student solution codes in C of the given programming problem.
Use the provided acceptable model solution to ensure the generated codes follows a similar approach.

Problem Description:
${problem.title}. ${problem.description}.
${problem.complexity}.
```

Acceptable Model Solution:
${run.solution.title}. ${run.solution.description}.

Model Solution C Code:
${run.solution.code}

Grade:
The grade assigned to generated codes is ${run.correctness} out of 10.
For low scores, you can include semantic and logical mistakes or misunderstandings or blunders made by a real student.
For higher scores, generate correct solution codes without any mistakes.
For medium scores, include both mistakes and correct codes.
You are free to change the code to make it fit for a grade of ${run.correctness} out of 10.

Output JSON Format:

```
[
  {
    "code": string: generated students code,
    "explanation": string: very concise and brief explanation of the code,
  }
]
```

Important Points:

1. The number of JSON objects in the output list has to be exactly ${run.count}.
2. The generated student's solution should closely resemble the approach of the model solution, allowing room for creative variants.
3. Strictly follow the output format.
'';

We made a web app using Flutter for data generation. The app generates student codes for any given problem. First, you specify the problem title and description, and select its complexity (low, medium or high). Once the problem is created, you add a model solution (multiple model solutions can be added to highlight different approaches to solving the same problem), and give it a title and a description. You can then make as many runs as you want. In each run, you specify the correctness level on a scale of 1 to 10 where 1 denotes a completely incorrect student code, and 10 denotes a completely correct student

code. Additionally, you enter the number of codes you want to be generated and select a model solution.

All of this data is fed to Gemini, as can be seen in the above prompt. Gemini then generates variants of the model solution. Once the run is complete, you can view the generated codes and alter the assigned grade, if needed.

Using this web app, we generated over 500 codes for 7 problems that are common in introductory programming courses. To ensure quality, we also manually validated the generated codes, and reassigned the grade wherever necessary.

Model and its architecture

After exploring several state-of-the-art LLMs, we decided to use CodeGemma, a collection of lightweight open code models built on top of Gemma. CodeGemma models are text-to-text and text-to-code decoder-only models and are available as a 7 billion pretrained variant that specializes in code completion and code generation tasks (codegemma-7b), a 7 billion parameter instruction-tuned variant for code chat and instruction following (codegemma-7b-it) and a 2 billion parameter pretrained variant for fast code completion (codegemma-2b). We used codegemma-7b-it.

The fine-tuned CodeGemma Instruct model is an advanced and specialized neural network structure, specifically designed for instruction-following. Building on the pre-trained CodeGemma model, CodeGemma has been optimized using PEFT (Parameter Efficient Fine Tuning) and LoRA (Low-Rank Adaptation) techniques to enhance its performance and efficiency. Additionally, 4-bit quantization has been applied to reduce memory usage and improve computational speed. Key Components of our CodeGemma model are:

1. **Embedding Layer (embed_tokens):**
    - This layer converts input token IDs which are codes into dense vector representations.
    - The embedding size is 4096, and it's designed to work with a vast vocabulary of 128,256 tokens.
    - A padding index is set to 2.
2. **Decoder Layers (layers):**
    - The model contains 32 decoder layers (indexed from 0 to 31), indicating a deep network capable of handling different coding related patterns.
    - Each GemmaDecoderLayer consists of:
        - **i.** GemmaSdpaAttention: A specialized attention mechanism with:
            - Quantized linear layers (Linear4bit) for query (q_proj), key (k_proj), and value (v_proj) projections, reducing memory usage.

- The output projection (o_proj) is also a quantized linear layer.
- GemmaRotaryEmbedding: A variant of rotary position embeddings to capture sequence position information.

ii. GeMLP: A multi-layer perceptron with:
- Quantized linear layers (gate_proj, up_proj, down_proj) using 4-bit precision.
- An activation function, likely scaled exponential linear units (SiLU), for introducing non-linearity.

iii. Two instances of GeRMSNorm (Root Mean Square Layer Normalization) for normalizing the inputs to the self-attention and after the attention block, to improve training stability.

3. **Final Normalization Layer (norm):**
   - A GeRMSNorm layer is used to normalize the final output of the decoder stack.
4. **Language Model Head (lm_head):**
   - A linear layer mapping the decoder output back to our coding tokens (4096 features to 128,256 output tokens).
   - This layer is used for generating the next token predictions .

This fine-tuned architecture uses the strengths of PEFT and LoRA to enhance the pretrained codegemma-7b-it model performance and its efficiency. 4-bit quantization is used to reduce memory usage and improve computational speed. The result is a highly optimized model capable of handling complex coding related tasks with precision and speed.

Fine-tuning

Finally, we used the data generated from Gemini to fine-tune CodeGemma. Several libraries were used for fine-tuning including:

1. **bitsandbytes:** Used to accelerate CUDA operations.
2. **PyTorch:** Used for building and training neural networks.
3. **HuggingFace's transformers library:** Used to import the pre-trained CodeGemma model.
4. **trl (Transformer Reinforcement Learning):** Used to enable reinforcement learning techniques to improve the model's decision-making processes.
5. **peft (Python Efficient Finetuning)**: Used to streamline fine-tuning and optimization of current model and to ensure maximum efficiency.
6. **Hugging Face's accelerate library:** Used to accelerate the fine-tuning process.
7. **auto-gptq and loralib:** Used for automated quantization and optimization.

This carefully curated ensemble of libraries enables CodeGemma to improve its performance and help teachers in code grading.

Once again, prompt engineering was crucial to get good results. After lots of experimentation, this is the prompt we finally used:

You are equipped with a comprehensive understanding of various programming fundamentals topics in C language.

Carefully analyze the given pair of C codes, representing a correct solution and a student's submission to the same programming problem.

Rigorously assess the logical/semantic similarity between the two code snippets, considering the following aspects:

-Functional Equivalence: Determine whether the code snippets produce the similar results for a given set of inputs.

- Consider handling potential edge cases or unusual inputs to ensure thorough evaluation.

- Recognize differences in algorithmic approaches or problem-solving techniques.

- Be highly sensitive to small changes like:

Equality operators (e.g., ==, !=, <=, >=)

Relational operators (e.g., <, >)

Assignment operators (e.g., =, +=, -=, *=, /=, %=)

Increment/decrement operators (e.g., ++, --)

Logical operators (e.g., &&, ||, !)

Bitwise operators (e.g., &, |, ^, ~)

Deduct points for even minute errors, such as:

Using <= instead of <

Using != instead of ==

Forgetting to increment a loop counter

Changing the loop counter's initialization or increment/decrement logic, leading to infinite loops or skipped iterations

Modifying the conditional statement in a way that alters the control flow

Understand the impact of mistakes on the solution, such as:

Infinite loops causing the program to run indefinitely

Skipped iterations or incorrect results due to modified loop logic

Incorrect conditional statements leading to wrong program execution paths

Your task is to grade the student's code on the basis of similarity with the given solution code.

For you, the solution code is always 100% correct. Judge student's code according to solution.

No grade should be given on any syntactical similarity, only on logical/semantic similarity as described above.

You have to try your best to give a low grade score.

Output Format:

Strictly adhere to the following standardized JSON format.

Do not output anything other than the specified JSON format.

{"similarity": "Similarity score ranging from 0 to 10"}

Example:

Solution: for (int i = 1; i<=10;i++)

Student Submission: for (int i = 1; i<=10;i--)

Similarity Score: 0 (Due to infinite loop and completely different logic

We made the prompt as detailed as possible, and explicitly addressed the previous problems by instructing the model to be highly sensitive to even small changes and giving it some examples of them, and instructing the model to give a low score.

Other work

Additionally, we also designed the web portal using Flutter. This web portal will serve as an interface where teachers can upload the model answer and the student files, and, in return, obtain the graded files.

Lastly, we created APIs to integrate our system with our supervisor’s education portal, iParhai — an Artificial Intelligence based solution that aims to offer personalized education for everyone.

Testing and Results

We tested our model on GPTCloneBench, a state-of-the-art comprehensive benchmark of semantic clones and cross-language clones created using GPT-3 model and SemanticCloneBench. It contains 79,928 clone pairs, with 37,149 true semantic clone pairs (Type-3/Type-4), 19,288 false semantic pairs (Type-1/Type-2), and 20,770 cross-language clones across four languages (Java, C, C#, and Python).

Following is a classification of the definition of clone that has found widespread acceptance in the scientific literature: [34]

- **Type-1 (T1):** Code segments that are identical except whitespace differences (and sometimes layout differences) and comment differences [6].
- **Type-2 (T2):** Fragments that are structurally and syntactically identical to one another, with the exception of differences in identifiers, literals, types, layout, and comments. [6]
- **Type-3 (T3):** Fragments that were copied with additional alterations made. Changes can also be made to literals, types, layouts, and comments, in addition to identifiers, which can be renamed or removed entirely. Statements can be modified, added to, or removed entirely [44].
- **Type-4 (T4):** Two or more snippets of code that, when combined, carry out the same computation but do so in accordance with distinct syntactic variations [44].

We chose GPTCloneBench as our benchmark because it is 15-fold larger than SemanticCloneBench, has more functional code examples for software systems and programming language support than CLCDSA, and overcomes BigCloneBench's qualities, quantification, and language variety limitations.

Since we have focused on C, we only took code pairs in C into consideration. Since GPTCloneBench is very big and we had limited computational power, we randomly sampled 1200 code pairs, divided as shown in Table 6 below. Note that positive pairs means that the two codes are clones whereas negative means that they are not. Negative pairs were created by taking one code from one file and another code from another file. Care was taken to ensure that they are not clones. GPTCloneBench only contains clones (or positive pairs as we call them), but we decided to test our model on negative pairs as well to ensure that it works well for not just correct student submissions but incorrect ones as well.

Table 6: Division of our testing dataset

	Positive pairs	Negative pairs	Total
Type-1/Type-2	200	200	400
Type-3	200	200	400
Type-4	200	200	400
Total	600	600	1200

We ran the model on the pairs in the test dataset. For each pair, we recorded the grade given by the model. All these results were stored in a jsonl file.

Since our testing data is binary while our model gives the score in a range of 0 to 10 depending on the level of correctness; for positive pairs, we considered a score of 8 or above to be correct and any score below that to be incorrect grading, whereas for negative pairs, we considered a score of 2 or below to be correct grading, and higher scores to be incorrect grading.

Attached below are the results of our model. We have compared it with LLaMA-3-8b:

CodeGemma

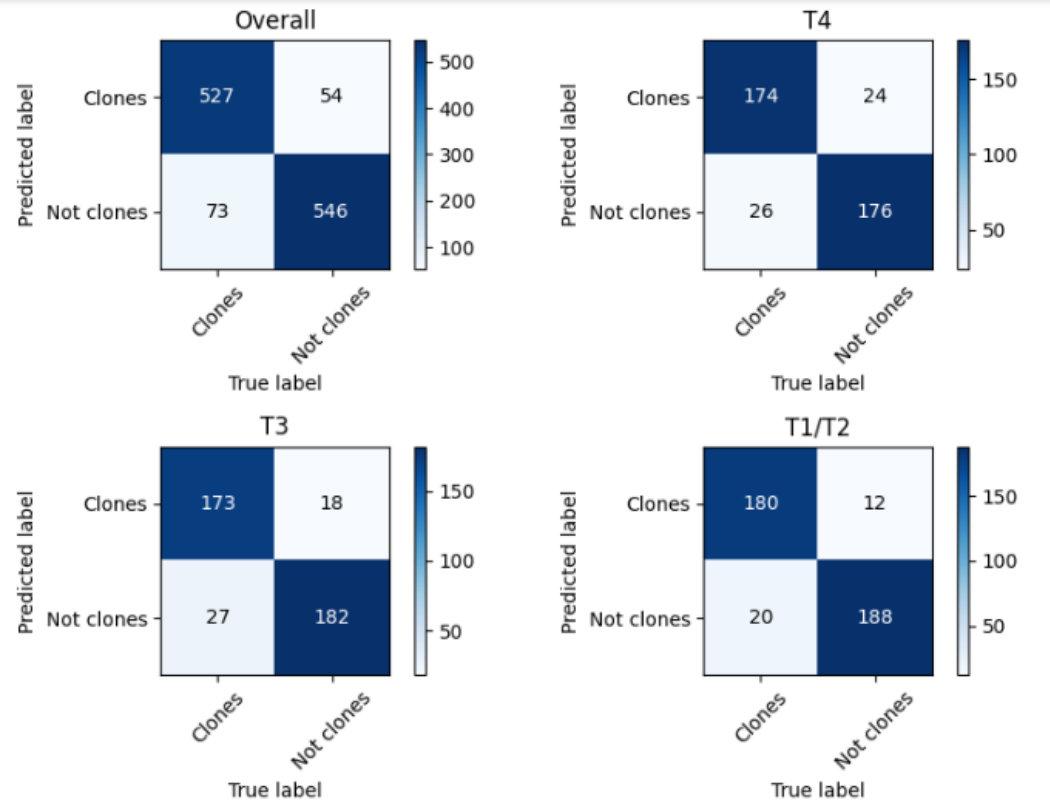


Figure 7: Confusion matrices showing performance of CodeGemma for Type-1/Type-2, Type-3, Type-4 and overall

Table 7: Results of CodeGemma for Type-1/Type-2, Type-3, Type-4 and overall

	T1/T2	T3	T4	Total
Correct	200	200	200	600
Correct Predicted	180	173	174	527
Recall	90%	86.5%	87%	87.83%
Incorrect	200	200	200	600
Incorrect Predicted	188	182	176	546
Specificity	94%	91%	88%	91%

LLaMA-3-8b

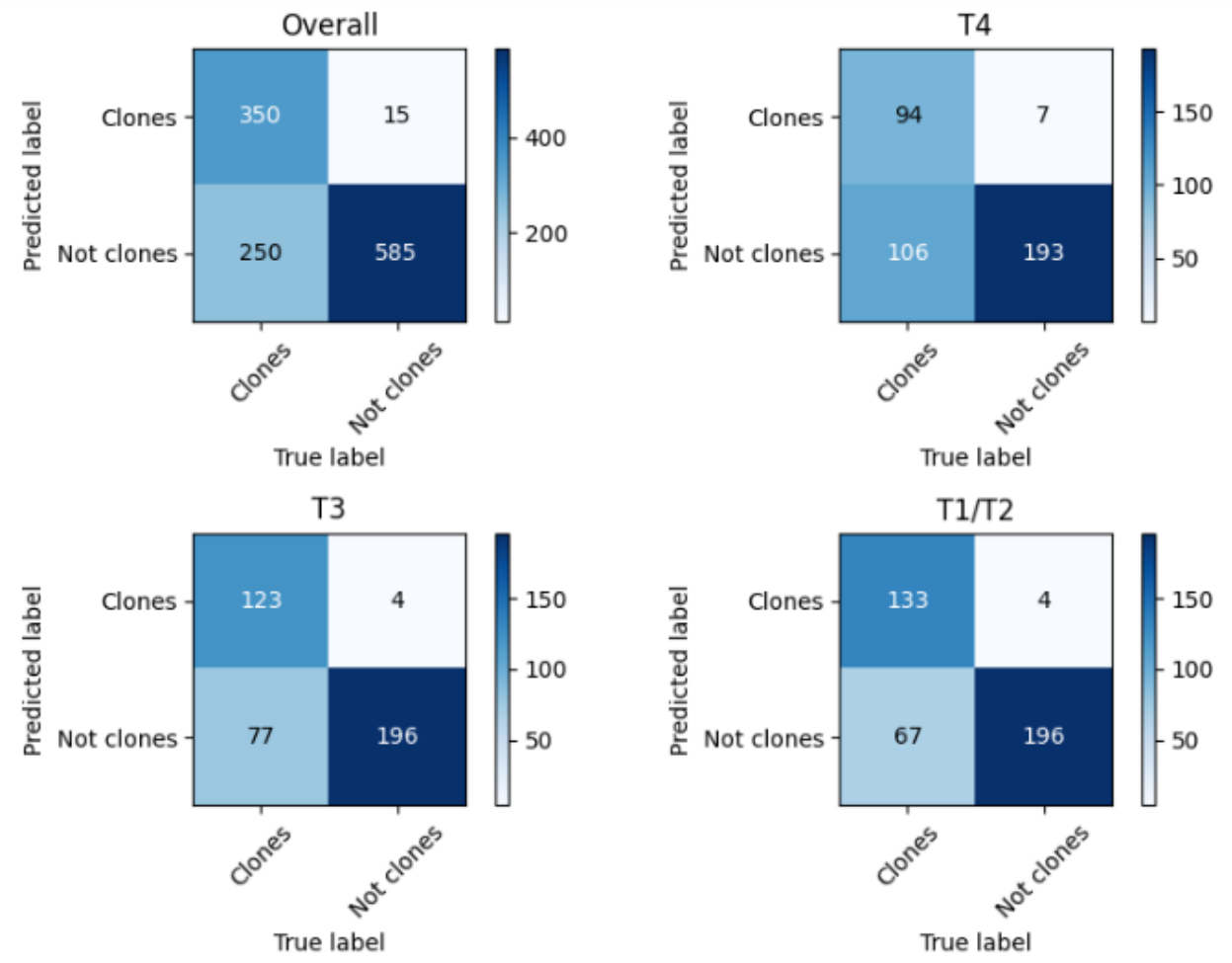


Figure 8: Confusion matrices showing performance of LLaMA-3-8b for Type-1/Type-2, Type-3, Type-4 and overall

Table 8: Results of LLaMA-3-8b for Type-1/Type-2, Type-3, Type-4 and overall

	T1/T2	T3	T4	Total
Correct	200	200	200	600
Correct Predicted	133	123	94	350
Recall	66.5%	61.5%	47%	58.33%
Incorrect	200	200	200	600
Incorrect Predicted	196	196	193	585
Specificity	98%	98%	96.5%	97.5%

As can be clearly seen, CodeGemma performs much better. LLaMA-3-8b has a high specificity but very low recall whereas CodeGemma is good at both of these metrics.

For evaluation, we used these metrics: accuracy, precision, recall, and F1 score. The models' performances on these metrics are illustrated in Table 9 below:

Table 9: Accuracy, Precision, Recall and F1 score of both models for Type-1/Type-2, Type-3, Type-4 and overall

Metric	Model	T1/T2	T3	T4	Total
Accuracy	**CodeGemma**	92%	88.75%	87.5%	89.42%
	LLaMA-3-8b	82.25%	79.75%	71.75%	77.92%
Precision	**CodeGemma**	93.75%	90.58%	87.88%	90.71%
	LLaMA-3-8b	97.08%	96.85%	93.07%	95.89%
Recall	**CodeGemma**	90%	86.5%	87%	87.83%
	LLaMA-3-8b	66.5%	61.5%	47%	58.33%
F1 Score	**CodeGemma**	0.92	0.88	0.87	0.89
	LLaMA-3-8b	0.79	0.75	0.62	0.73

These results are visualized on the next page:

CodeGemma

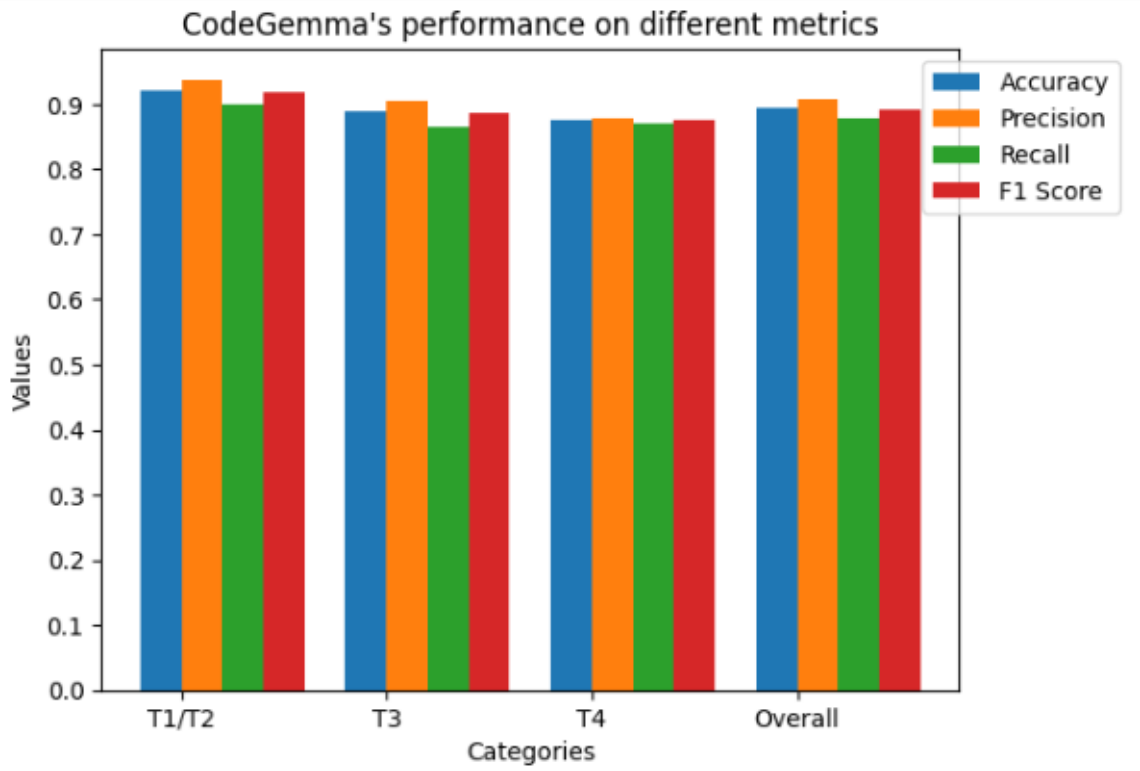


Figure 9: CodeGemma's performance on different metrics

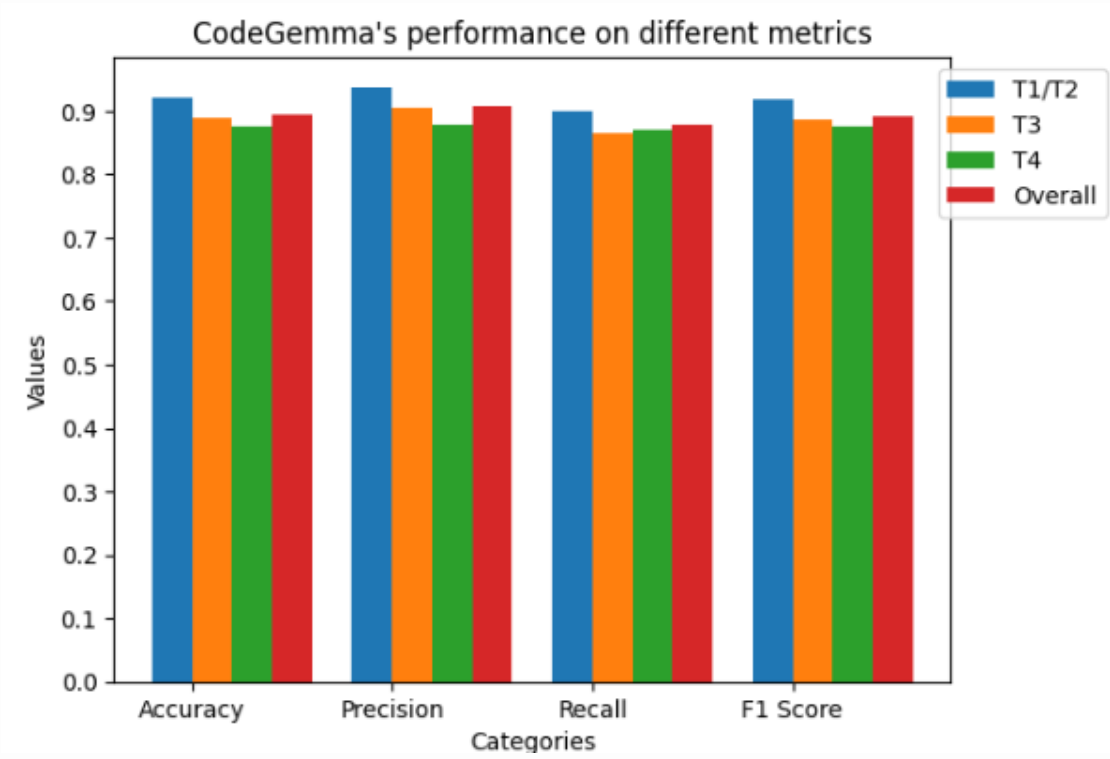


Figure 10: CodeGemma's performance on different metrics

LLaMA-3-8b

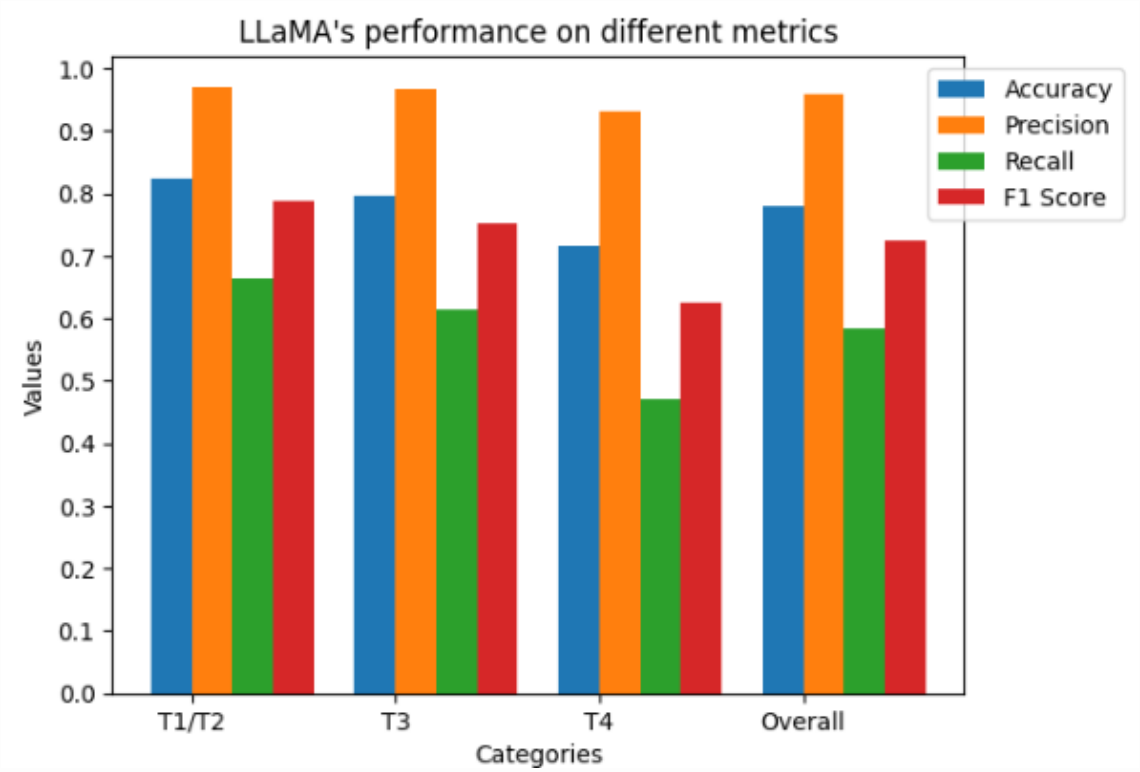


Figure 11: LLaMA's performance on different metrics

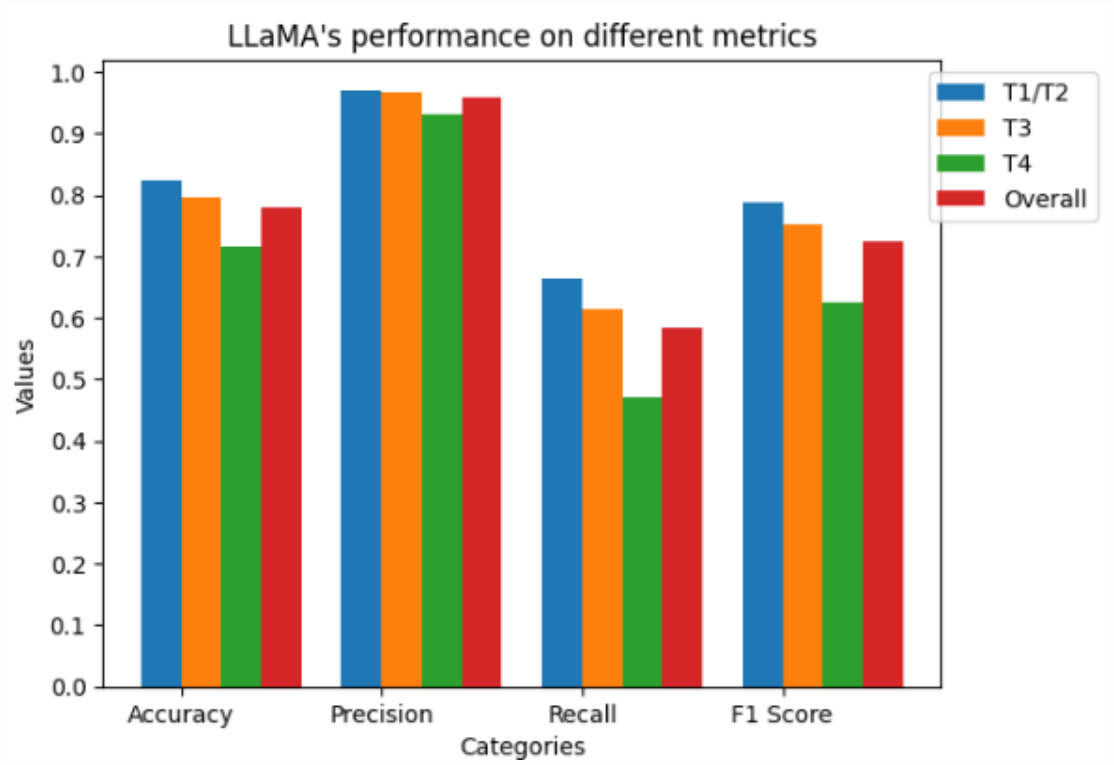


Figure 12: LLaMA's performance on different metrics

Side-by-side comparisons

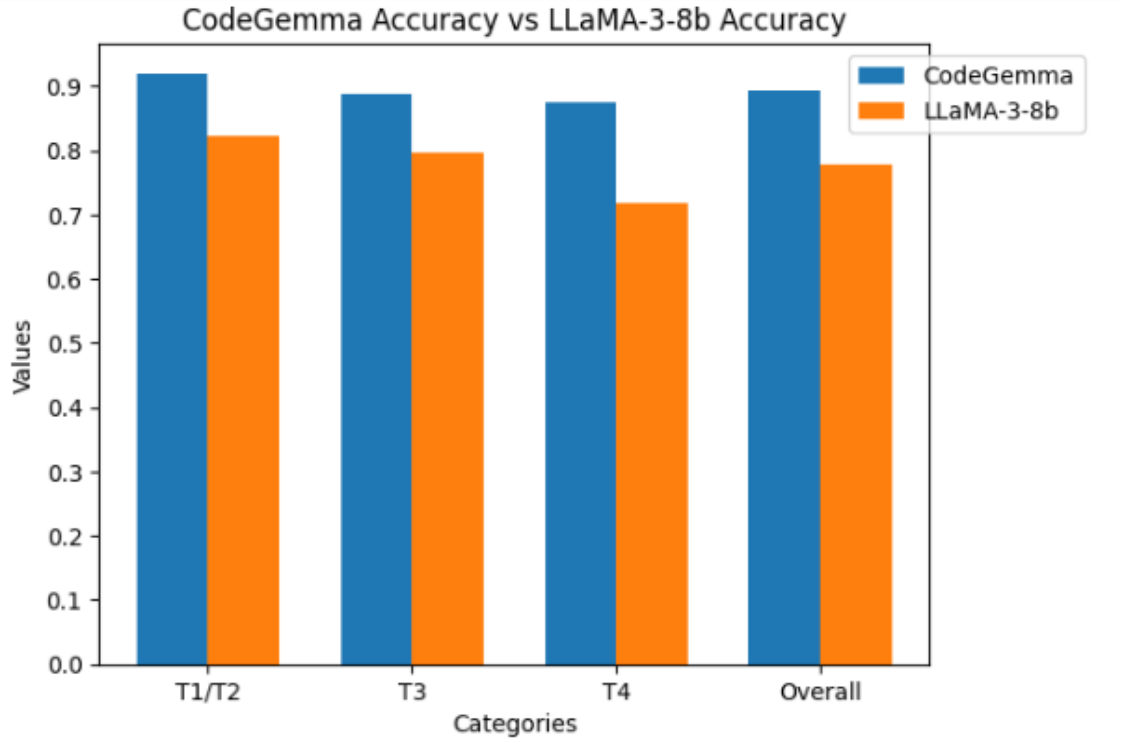


Figure 13: Comparison of accuracies of both models

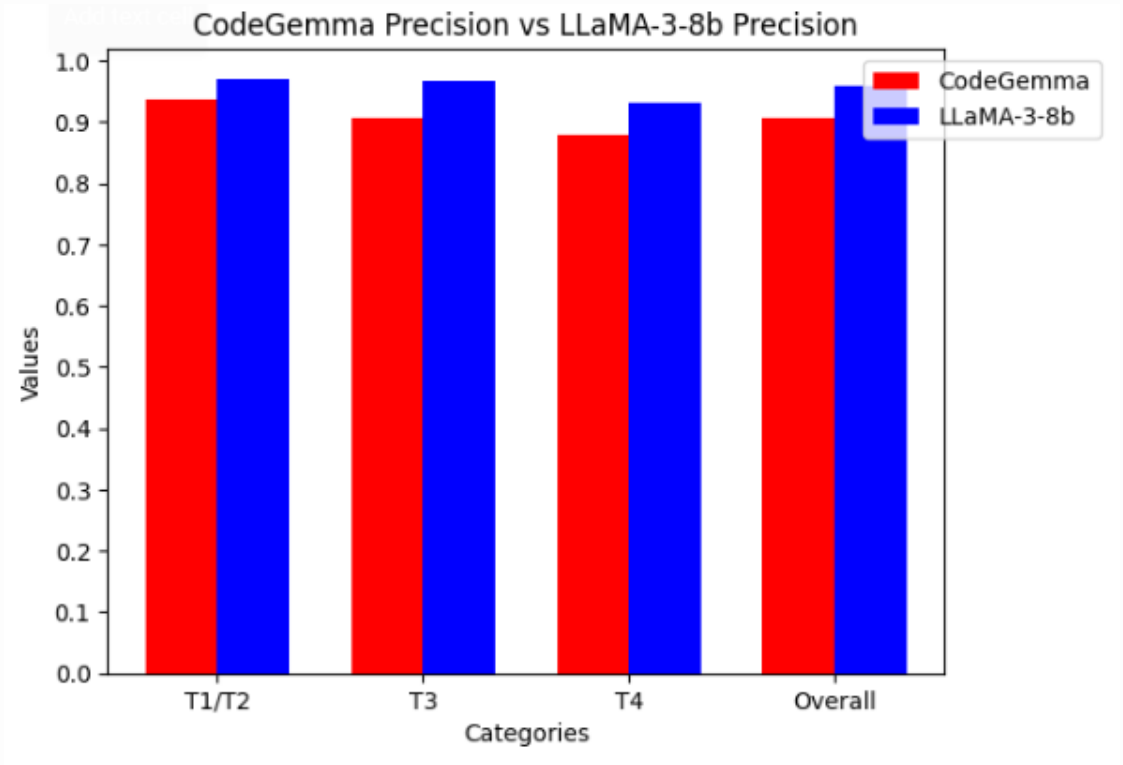


Figure 14: Comparison of precisions of both models

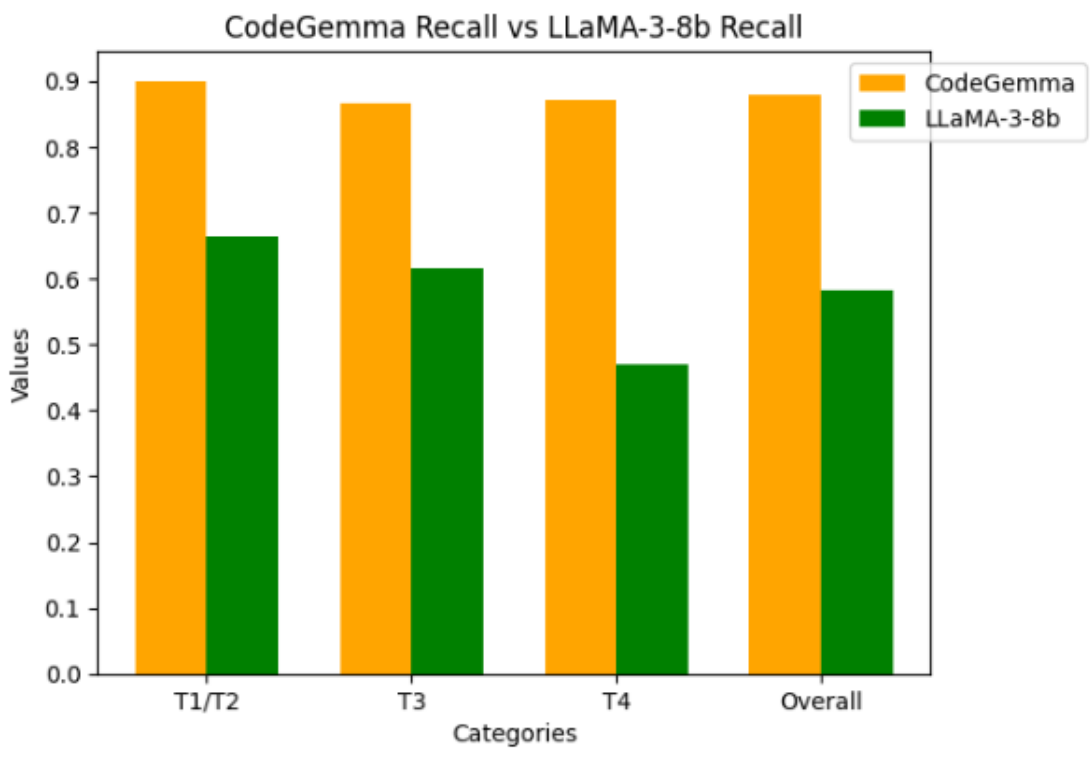


Figure 15: Comparison of recalls of both models

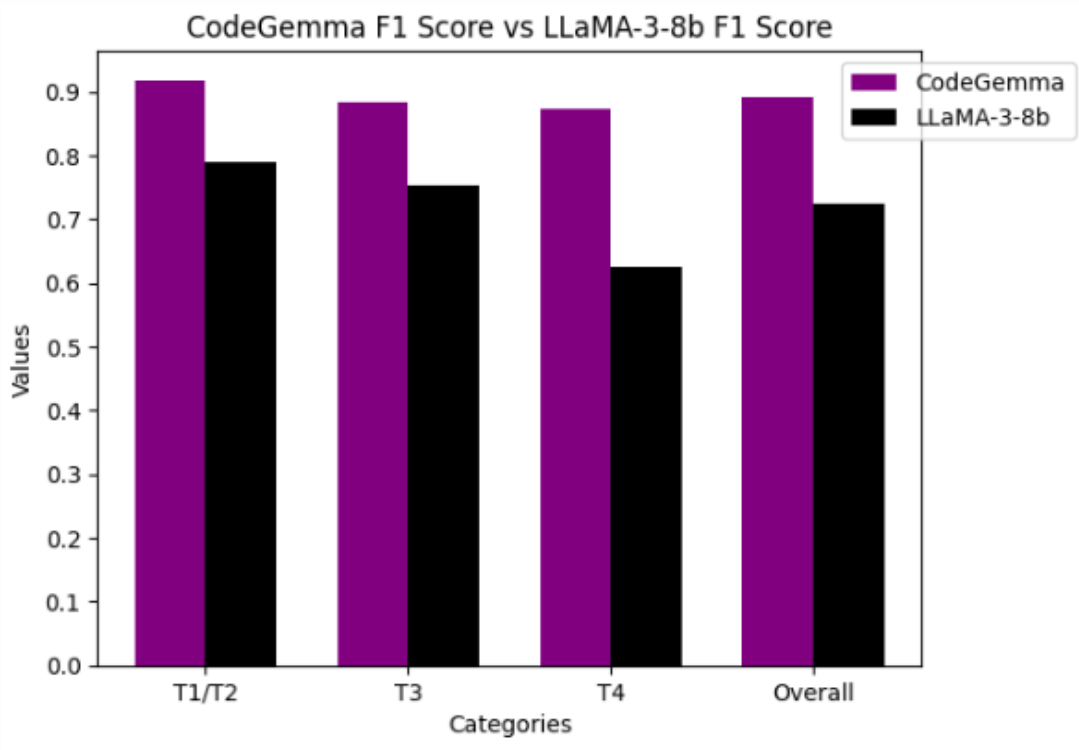


Figure 16: Comparison of F1 scores of both models

From the data illustrated above, we can clearly notice that the performance of both models, on all metrics, is best on Type-1/Type-2 clones and worst on Type-4 clones. This makes sense since Type-4 clones are the most challenging to detect unlike Type-1/Type-2 clones which have large degree of syntactic similarity.

Moreover, even though LLaMA-3-8b has great precision, it does not do as well on other metrics unlike CodeGemma which performs well on all metrics making it significantly better than its counterpart.

Recommendations for Future Work

Even though our model shows great results, it should be noted that GPTCloneBench is originally made for code clone detection. As such, the testing dataset we have derived from it is also binary.

Our model, however, is not made for code clone detection (which is a binary classification task), but rather for code grading, where the grade can be any value between 0 and 10, making this a regression/multi-classification task.

Unfortunately, there is no standard dataset available right now that grades code pairs on the basis of percentage of logical/semantic similarity; instead, all the standard datasets are binary, much like GPTCloneBench.

We believe that testing can be improved significantly by using a dataset that grades code pairs on a scale of 0 to 10 on the basis of their logical/semantic similarity with one another instead of a binary dataset.

Additionally, improvements can also be made by using a larger dataset (be it for testing or for fine-tuning) and by using a more powerful system with greater computing power.

Conclusion

We have integrated our system with iParhai, an Artificial Intelligence based solution that aims to offer personalized education for everyone. We hope that it will be successful, and we soon aim to integrate it with other educational platforms as well.

We hope that our program eases the task of teachers and goes on to revolutionize the checking of programming assignments as it will not only save time, but also produce consistent marking and reduce the need for Teacher Assistants. It will eliminate the possibility of human errors, inconsistent or biased grading to ensure fairness and transparency in the grading process. Furthermore, it will enhance students' learning experience by providing them immediate feedback on their submissions, allowing for faster identification and correction of errors.

References

[1] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, “Contrastive Code Representation Learning,” arXiv.org, 2021. https://arxiv.org/abs/2007.04973

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: learning distributed representations of code,” Proceedings of the ACM on Programming Languages, vol. 3, no. POPL, pp. 1–29, Jan. 2019, doi: https://doi.org/10.1145/3290353

[3] J. K. Siow, S. Liu, X. Xie, G. Meng and Y. Liu, "Learning Program Semantics with Code Representations: An Empirical Study," 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 2022, pp. 554-565, doi: 10.1109/SANER53432.2022.00073

[4] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, “Learning Program Embeddings to Propagate Feedback on Student Code,” proceedings.mlr.press, Jun. 01, 2015. https://proceedings.mlr.press/v37/piech15.html

[5] Y. Qin, G. Sun, J. Li, T. Hu, and Y. He, “SCG_FBS: A Code Grading Model for Students’ Program in Programming Education,” Feb. 2021, doi: https://doi.org/10.1145/3457682.3457714

[6] K. Fukushima, T. Ishio, K. Shimari and K. Matsumoto, "A Similarity-based Assisted Grading for Introductory Programming Course," 2022 IEEE 16th International Workshop on Software Clones (IWSC), Limassol, Cyprus, 2022, pp. 23-24, doi: 10.1109/IWSC55060.2022.00012.

[7] M. R. I. Bariansyah, S. A. Rukmono and R. S. Perdana, "Semantic Approach for Increasing Test Case Coverage in Automated Grading of Programming Exercise," 2021 International Conference on Data and Software Engineering (ICoDSE), Bandung, Indonesia, 2021, pp. 1-6, doi: 10.1109/ICoDSE53690.2021.9648439.