The ability of LLMs to detect and fix logical errors

Paper 1

LLMs cannot find reasoning errors, but can correct them given the error location

1. Study Overview

Title: LLMs Cannot Find Reasoning Errors, but Can Correct Them Given the Error LocationAuthors: Gladys Tyen, Hassan Mansoor, Victor Cărbune, Peter Chen, Tony MakAffiliations: University of Cambridge, Google ResearchPublication Year: 2024Objective: To investigate whether Large Language Models (LLMs) can effectively detect logical errors in their outputs and, if given the correct mistake location, correct them accurately.

Key Question: Why do LLMs fail in self-correction for reasoning tasks?

2. Research Idea & Hypothesis

2.1 Core Hypothesis.

LLMs are weak at identifying logical mistakes in reasoning-based tasks. However, they are strong at correcting mistakes when given precise error locations. A small mistake-finding classifier trained on out-of-domain data can outperform LLMs in mistake detection.

2.2 Importance of Study

Self-correction methods often degrade performance because LLMs fail to accurately locate reasoning mistakes. Improving mistake detection could significantly enhance the reliability of AI-driven reasoning tasks. Developing a dedicated mistake-classifier may be an effective alternative to fine-tuning large models.

3. Methodology

3.1 Study Approach: Separates self-correction into two distinct tasks:

1. Mistake Finding – Evaluates LLMs' ability to detect logical reasoning errors.
2. Error Correction – Tests whether LLMs can correct errors once the mistake location is provided.

Uses a backtracking method to correct errors by regenerating incorrect.

3.2 Benchmark Models

Tested LLMs include:

. GPT-4 Turbo.

. GPT-4.

. GPT-3.5 Turbo.

. Gemini Pro.

. PaLM 2 Unicorn (also used for dataset generation).

3.3 Experimental Setup

Mistake Finding Approaches.

1- Direct Trace-Level Prompting – Model is asked to find the mistake in the entire reasoning trace.

2- Direct Step-Level Prompting – Model is prompted at each step for correctness.

3- Chain-of-Thought (CoT) Step-Level Prompting – Model reasons step-by-step before determining correctness.

Error Correction Approach.

Uses a backtracking method where incorrect steps are regenerated until a different step is produced.

Performance is compared against random resampling to ensure improvements are meaningful.

4. Dataset: BIG-Bench Mistake

4.1 Overview

Dataset Name: BIG-Bench Mistake.

Size: 2186 Chain-of-Thought (CoT) traces.

Source: Generated using PaLM 2 Unicorn.

Annotations: Each trace is labeled with the first logical mistake.

Availability: Publicly released at **GitHub:** BIG-Bench Mistake

4.2 Tasks Covered

Tasks:

. Word sorting: Arranging words in alphabetical order.

. Tracking shuffled objects: Following objects through shuffled positions.

. Logical deduction : Solving step-by-step logical reasoning problems.

. Multistep arithmetic: Performing calculations across multiple steps.

. Dyck languages: Identifying valid nested bracket sequences.

4.3 Annotation Process.

Human Annotators: Labeled mistakes for 4 out of 5 tasks.

Automatic Annotation: Used for Dyck languages due to limited variation in responses.

Inter-Rater Agreement: Measured using Krippendorff's alpha (>0.95 across tasks).

5. Evaluation Metrics

5.1 Mistake Finding Accuracy.

Evaluates whether models correctly identify the first logical mistake in a trace.

Performance compared across three prompting strategies (trace-level, step-level, CoT-step-level).

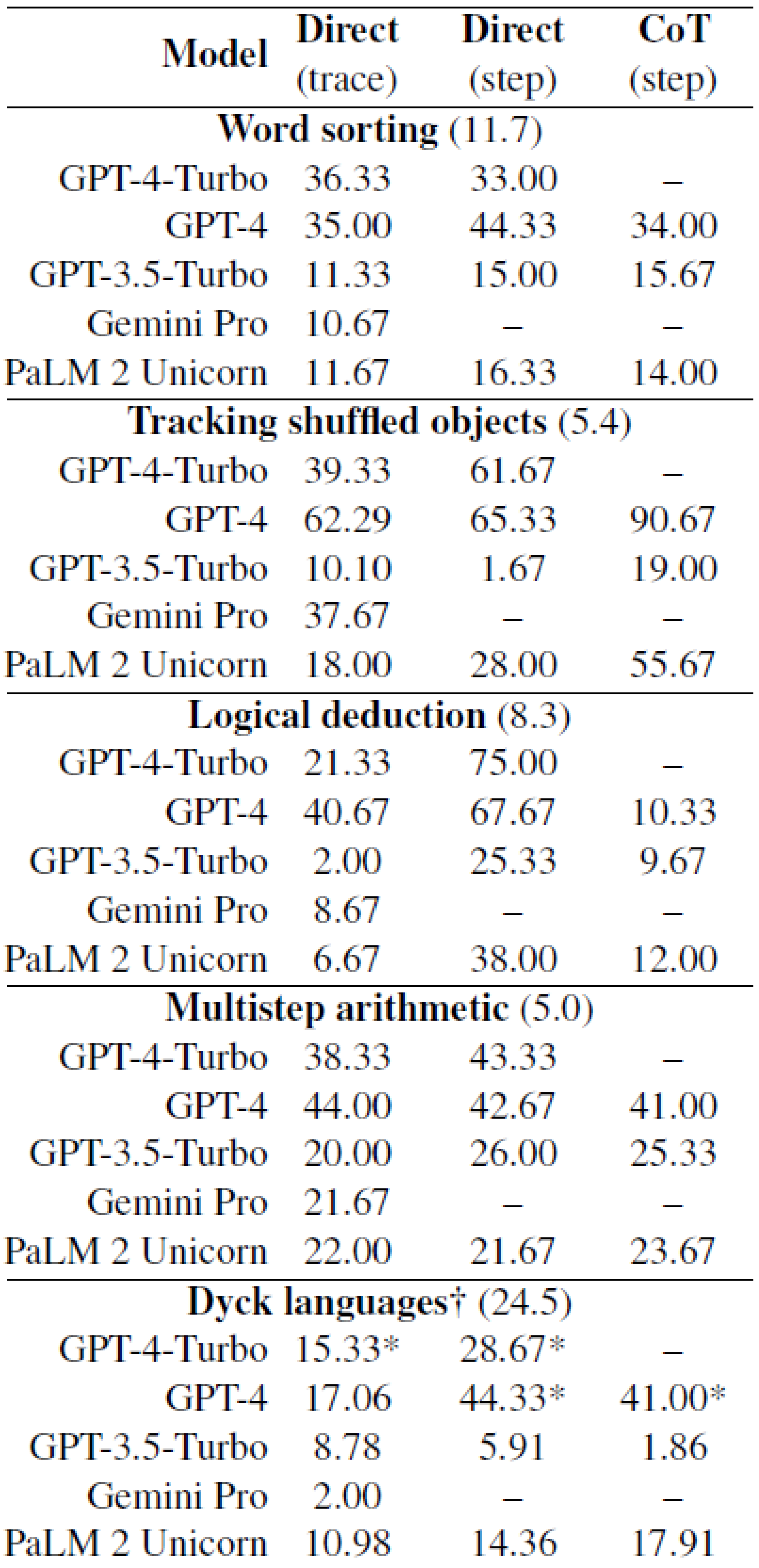
5.2 Error Correction Performance.

Measures accuracy improvement when correct mistake locations are provided.

Uses backtracking correction and compares against random resampling.

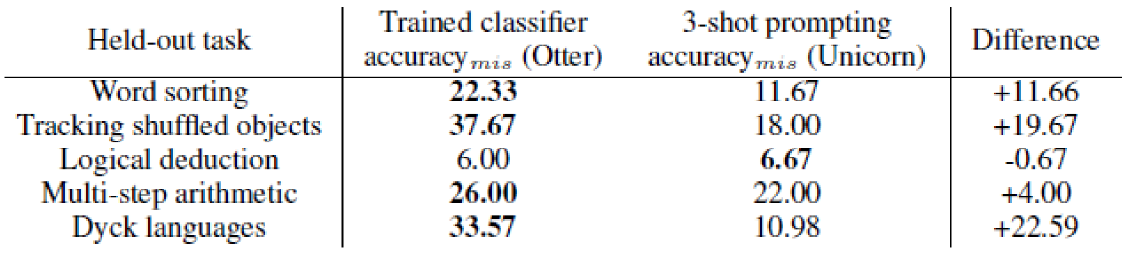
6. Results

6.1 Mistake Finding Accuracy (Best Model Results)



Model	Direct (trace)	Direct (step)	CoT (step)
Word sorting (11.7)			
GPT-4-Turbo	36.33	33.00	–
GPT-4	35.00	44.33	34.00
GPT-3.5-Turbo	11.33	15.00	15.67
Gemini Pro	10.67	–	–
PaLM 2 Unicorn	11.67	16.33	14.00
Tracking shuffled objects (5.4)			
GPT-4-Turbo	39.33	61.67	–
GPT-4	62.29	65.33	90.67
GPT-3.5-Turbo	10.10	1.67	19.00
Gemini Pro	37.67	–	–
PaLM 2 Unicorn	18.00	28.00	55.67
Logical deduction (8.3)			
GPT-4-Turbo	21.33	75.00	–
GPT-4	40.67	67.67	10.33
GPT-3.5-Turbo	2.00	25.33	9.67
Gemini Pro	8.67	–	–
PaLM 2 Unicorn	6.67	38.00	12.00
Multistep arithmetic (5.0)			
GPT-4-Turbo	38.33	43.33	–
GPT-4	44.00	42.67	41.00
GPT-3.5-Turbo	20.00	26.00	25.33
Gemini Pro	21.67	–	–
PaLM 2 Unicorn	22.00	21.67	23.67
Dyck languages† (24.5)			
GPT-4-Turbo	15.33*	28.67*	–
GPT-4	17.06	44.33*	41.00*
GPT-3.5-Turbo	8.78	5.91	1.86
Gemini Pro	2.00	–	–
PaLM 2 Unicorn	10.98	14.36	17.91

6.2 Accuracy Gains After Providing Mistake Location



Held-out task	Trained classifier accuracy$_{mis}$ (Otter)	3-shot prompting accuracy$_{mis}$ (Unicorn)	Difference
Word sorting	**22.33**	11.67	+11.66
Tracking shuffled objects	**37.67**	18.00	+19.67
Logical deduction	6.00	**6.67**	-0.67
Multi-step arithmetic	**26.00**	22.00	+4.00
Dyck languages	**33.57**	10.98	+22.59

6.3 Key Observations.

GPT-4 performed best overall in mistake detection.

PaLM 2 Unicorn struggled significantly with mistake finding.

Providing explicit mistake locations improved accuracy by up to +43.92%.

Step-level prompting outperformed trace-level prompting.

Paper 2

Decoding Logic Errors: A Comparative Study on Bug Detection by Students and Large Language Models

1. Idea

The study investigates the ability of Large Language Models (LLMs), specifically GPT-3 and GPT-4, to detect logic errors in C programs and compares their performance to 964 introductory programming students. This is important because logic errors are difficult to detect as they do not trigger compiler errors but instead cause unintended behavior. Students often struggle with debugging due to a lack of experience, making it a challenging task in programming education. While LLMs have demonstrated success in code generation and explanation, their ability to detect logic errors without executing the code remains unclear. The study aims to evaluate whether LLMs can be used as educational tools to help students detect and fix logic errors more effectively.

Key points:

- Logic errors are particularly challenging for novice programmers because they do not produce syntax errors but rather incorrect program behavior.
- LLMs have shown promise in various programming-related tasks, but their ability to detect and explain logic errors (without executing code) is unclear.
- The study compares GPT-3, GPT-4, and 964 students from an introductory C programming course at the University of Auckland in identifying three types of logic errors:
    - Operator error
    - Expression error
    - Out-of-bounds error
- The results provide insights into how LLMs could be integrated into educational tools to support learning.

2. Methodology

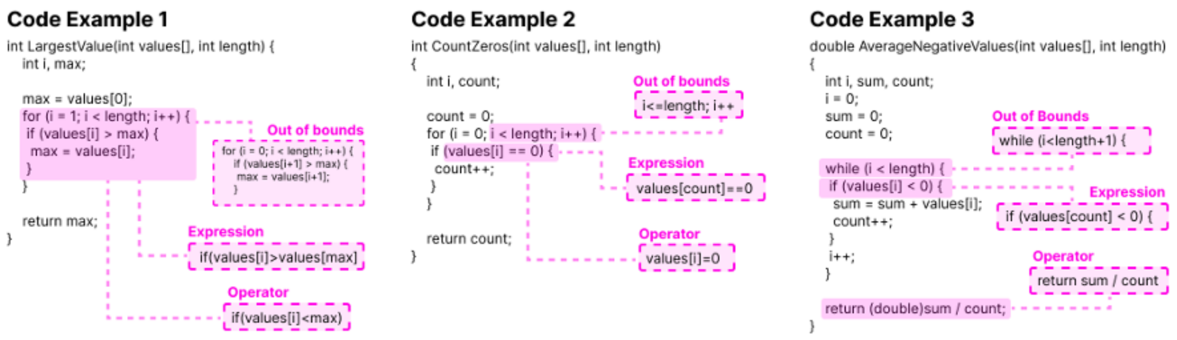
2.1 Research Questions

The study aims to answer:

1. How do students and LLMs compare in identifying logic errors in faulty code?

2. Which types of logic errors are easiest for students and LLMs to identify?

3. How many bugs or issues do students and LLMs identify when reviewing faulty and correct code?



Code Example 1

```
int LargestValue(int values[], int length) {
   int i, max;

   max = values[0];
   for (i = 1; i < length; i++) {
    if (values[i] > max) {
     max = values[i];
    }
   }

   return max;
}
```

Out of bounds

```
for (i = 0; i < length; i++) {
   if (values[i+1] > max) {
    max = values[i+1];
   }
```

Expression

```
if(values[i]>values[max]
```

Operator

```
if(values[i]<max)
```

Code Example 2

```
int CountZeros(int values[], int length)
{
   int i, count;

   count = 0;
   for (i = 0; i < length; i++) {
    if (values[i] == 0) {
     count++;
    }
   }

   return count;
}
```

Out of bounds

```
i<=length; i++
```

Expression

```
values[count]==0
```

Operator

```
values[i]=0
```

Code Example 3

```
double AverageNegativeValues(int values[], int length)
{
   int i, sum, count;
   i = 0;
   sum = 0;
   count = 0;

   while (i < length) {
    if (values[i] < 0) {
     sum = sum + values[i];
     count++;
    }
    i++;
   }

   return (double)sum / count;
}
```

Out of Bounds

```
while (i<length+1) {
```

Expression

```
if (values[count] < 0) {
```

Operator

```
return sum / count
```

2.2 Study Design

- Participants: 964 first-year C programming students
- Each participant reviewed three C code snippets, each containing either:
    - Correct code
    - One of three buggy variants
- They were asked to:
    - Identify any errors or bugs
    - Describe the function's intended purpose
- LLMs (GPT-3 & GPT-4) were tested on the same tasks using multiple prompts and response temperatures to account for variability.

2.3 Models

- Two models: GPT-3 (text-davinci-003) & GPT-4 (gpt-4-0314).
- Three prompts were used, such as:
    1. "List all errors and bugs, if any, found in the following C code: <code>"
    2. "List any issues, including bugs, errors, or potential problems..."
    3. "Assume the role of a highly intelligent computer scientist..."
- Two temperature settings:
    - 0.7 (default) for balanced responses
    - 0.3 (lower) for more deterministic answers

Each combination of prompt & temperature was tested five times per code snippet, totaling 720 LLM responses.

3. Datasets

- 2980 total student responses collected from a C programming lab session.
- 720 LLM responses generated for analysis.
- 30 responses per form were used for each code for each error type, ensuring repeatability and analysis of the results of the analyses.

4. Evaluation Metrics

The study used both quantitative and qualitative evaluation metrics to assess how well students and LLMs (GPT-3 and GPT-4) detected logic errors in C programs. Below is a more detailed breakdown of each metric:

4.1 Bug Detection Accuracy

Definition:

- Measures how well students and LLMs correctly identified logic errors in faulty code.
- Expressed as a percentage of correct responses for each group (students, GPT-3, GPT-4).

How it was measured:

- Each participant (student or LLM) was given three C programs, each with different bug variants.
- Their responses were checked to see if they correctly identified the actual error in the code.

If a participant correctly identified the bug, it was counted as a correct detection.

4.2 False Positive Rate

Definition:

- Measures how often students or LLMs incorrectly flagged correct code as containing an error.
- Expressed as a percentage of incorrect error identifications for each group.

How it was measured:

- Participants were also given a correct version of the code with no errors.
- If they mistakenly reported that the correct code contained an error, it was counted as a false positive.

4.3 Number of Bugs Identified

Definition:

- Measures how many bugs each participant (student or LLM) identified in the faulty code.
- Used to compare whether LLMs detected more bugs than students and if they identified valid or additional unintended issues.

How it was measured:

- Each participant reviewed three C code examples, each containing a specific bug (operator error, out-of-bounds error, or expression error).
- Researchers counted how many bugs each participant identified and whether they correctly matched the intended bug in the code.

4.4 Response Verbosity

Definition:

- Measures the length and detail of responses provided by students, GPT-3, and GPT-4.
- Expressed as the average word count per response.

How it was measured:

- The total number of words used in each response was counted.
- The average word count was calculated for each participant group (students, GPT-3, GPT-4).

4.5 Qualitative Analysis

Definition:

- A manual review of responses to check whether the detected bugs were valid, useful, and correctly explained.

How it was measured:

- A team of four researchers manually reviewed responses from students and LLMs.
- Responses were categorized as:
  - Correctly identifying the bug
  - Incorrectly identifying a non-existent bug
  - Providing unnecessary suggestions (e.g., style improvements instead of logic errors)
- Researchers also noted whether responses contained valid explanations of the detected bugs.

4.6 Statistical Analysis

Definition:

- A statistical model was used to compare the performance differences between students, GPT-3, and GPT-4 across different error types.

How it was measured:

- The study used a linear mixed-effects model, which is a statistical technique that accounts for variability across different conditions (e.g., different types of bugs, different participants).
- The model helped compare how well students, GPT-3, and GPT-4 performed relative to each other while controlling for variations in the dataset.

5. Results

5.1 Bug Detection Accuracy

- LLMs significantly outperformed students in detecting faulty code:
    - GPT-3: 87.3% accuracy
    - GPT-4: 99.2% accuracy
    - Students: 34.5% accuracy

However, LLMs were prone to over-reporting errors in correct code:

- Students correctly identified 92.8% of correct code.
- GPT-3: 79.4% accuracy in detecting correct code.
- GPT-4: Only 42.2% accuracy in detecting correct code.
- This suggests that GPT-4 was more "aggressive" in identifying potential issues, even in error-free code.

5.2 Number of Bugs Identified

- GPT-4 identified significantly more bugs than both GPT-3 and students.
- However, GPT-4 often reported minor style issues (e.g., missing function prototypes, naming conventions) as "bugs."
- Students sometimes identified incorrect fixes or misinterpreted bugs.

5.3 Verbosity of Responses

- GPT-4 was significantly more verbose than both GPT-3 and students.
    - GPT-4 responses averaged 129 words.
    - GPT-3 responses averaged 54 words.
    - Student responses averaged 38 words.

GPT-4 often included explanations and suggested solutions, making its responses more detailed but sometimes overwhelming.

6. Discussion & Implications

6.1 Strengths of LLMs

- Superior error detection: Both GPT-3 and GPT-4 identified logic errors more accurately than students.
- Potential for integration: LLMs could be used in IDEs or debugging tools to assist students.

6.2 Weaknesses of LLMs

- Over-detection in correct code: LLMs flagged stylistic choices as "errors."
- Excess verbosity: GPT-4 provided lengthy explanations, which could overwhelm students.
- Potential for misleading students: If students blindly trust LLM feedback, they may fix "non-bugs" or become over-reliant.

6.3 Future Work

- Investigating more complex logic errors.
- Studying how expert programmers use LLMs.

Refining LLM responses to reduce false positives in correct code.

7. Limitations

- Student responses were open-ended, making it difficult to determine intent when responses were blank.
- Students might perform better on their own code than on unfamiliar snippets.
- Only three types of bugs were tested—future work could explore broader categories.

8. Conclusion

- GPT-4 demonstrated near-perfect detection of logic errors in buggy code but was too aggressive in identifying minor issues in correct code.
- Students were better at recognizing correct code but struggled to detect subtle logic errors.

LLMs could be useful debugging assistants, but need calibration to reduce false positives and verbosity.

9. Practical Implications

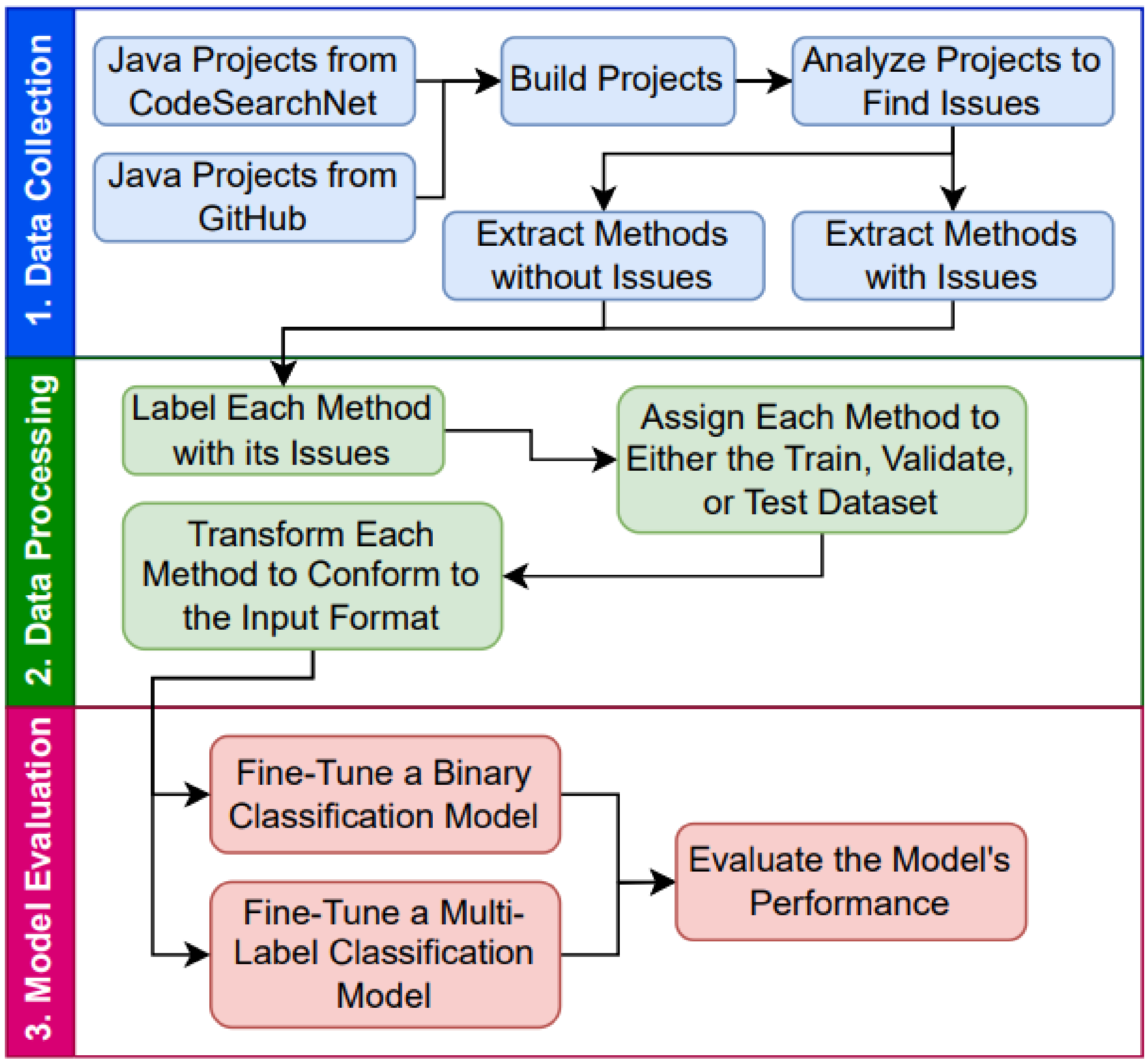
- For educators: LLMs could be integrated into learning environments but require fine-tuning.
- For students: LLMs can be helpful but should not be blindly trusted.

For developers: Future AI-assisted debugging tools should balance precision & recall.

Paper 3
Code Linting using Language Models" (June 2024)

Abstract:

The study investigates the application of Large Language Models (LLMs) in developing a versatile code linter capable of detecting potential issues in source code across various programming languages. Traditional linters are often language-specific and focus on certain types of issues, potentially leading to false positives. This research aims to create a language-independent linter that covers a variety of issue types while maintaining high speed.


1. Data Collection
Java Projects from CodeSearchNet
Java Projects from GitHub
Build Projects
Analyze Projects to Find Issues
Extract Methods without Issues
Extract Methods with Issues
2. Data Processing
Label Each Method with its Issues
Assign Each Method to Either the Train, Validate, or Test Dataset
Transform Each Method to Conform to the Input Format
3. Model Evaluation
Fine-Tune a Binary Classification Model
Fine-Tune a Multi-Label Classification Model
Evaluate the Model's Performance

Methodology:

1. **Dataset Collection:** The authors compiled a substantial dataset of code snippets accompanied by their associated issues. While specific details about the dataset's composition are not provided in the abstract, it is implied that the dataset encompasses multiple programming languages and a diverse range of code issues.
2. **Model Selection and Training:** A language model was selected to train two classifiers:
    a. **Binary Classifier:** Determines whether a code snippet contains issues.
    b. **Multi-Label Classifier:** Identifies the specific types of issues present in the code.

The training process involved feeding the collected dataset into the language model to enable it to learn patterns associated with code issues.

Evaluation Metrics:

The performance of the LLM-based linter was assessed using accuracy metrics:

- **Binary Classifier Accuracy:** Measures the proportion of correctly identified code snippets as either containing issues or being issue-free.
- **Multi-Label Classifier Accuracy:** Evaluates the model's ability to correctly identify multiple types of issues within a single code snippet.

Results:

The experimental studies demonstrated that the LLM-based linter achieved:

- **Binary Classifier Accuracy:** 84.9%
- **Multi-Label Classifier Accuracy:** 83.6%

These results indicate that the proposed linter is effective in detecting a wide range of code issues across different programming languages.

Conclusion:

The research presents a significant advancement in code analysis by leveraging LLMs to develop a language-independent linter capable of identifying various code issues with high accuracy. This approach addresses limitations of traditional linters, such as language specificity and a narrow focus on certain issue types, thereby contributing to the development of high-quality software systems.

Paper 4

Debug Bench: Evaluating Debugging Capability of Large Language Models

Introduction

Large Language Models (LLMs) have demonstrated exceptional coding capabilities, but their ability to debug and fix logical errors remains relatively underexplored. Logical errors, unlike syntax errors, require a deeper understanding of program behavior, making them particularly challenging for automated systems. This literature review examines the findings from the study *DebugBench: Evaluating Debugging Capability of Large Language Models* by Tian et al. (2024).

Study Overview

Dataset

- **Source:** DebugBench dataset was constructed using LeetCode programming challenge solutions.
- **Size:** 4,253 instances covering various bug categories.
- **Languages:** C++, Java, and Python.
- **Bug Implantation:** Bugs were artificially introduced using GPT-4 to ensure diversity and controlled evaluation.

- **Idea**

  The study "DebugBench: Evaluating Debugging Capability of Large Language Models" by Tian et al. (2024) introduces a novel framework to systematically assess how LLMs identify and resolve software bugs, with a focus on logical errors—a category of defects that are notoriously difficult to automate due to their dependence on contextual reasoning and program semantics. The core idea of DebugBench is rooted in addressing two critical gaps in existing research:

  1. Lack of Standardized Benchmarks: Prior evaluations of LLM debugging capabilities often relied on small, ad-hoc datasets or focused on syntax errors, which do not reflect the complexity of real-world debugging tasks. DebugBench fills this void by providing a large-scale, controlled benchmark that categorizes errors into distinct types (syntax, reference, logic, and multiple errors) and spans multiple programming languages (Python, Java, C++).
  2. Human vs. AI Debugging Comparison: While LLMs have shown promise in code generation, their ability to emulate human debugging intuition—particularly for subtle logical flaws—remains poorly understood. DebugBench directly compares LLM performance to human developers, offering insights into the strengths and limitations of current models.

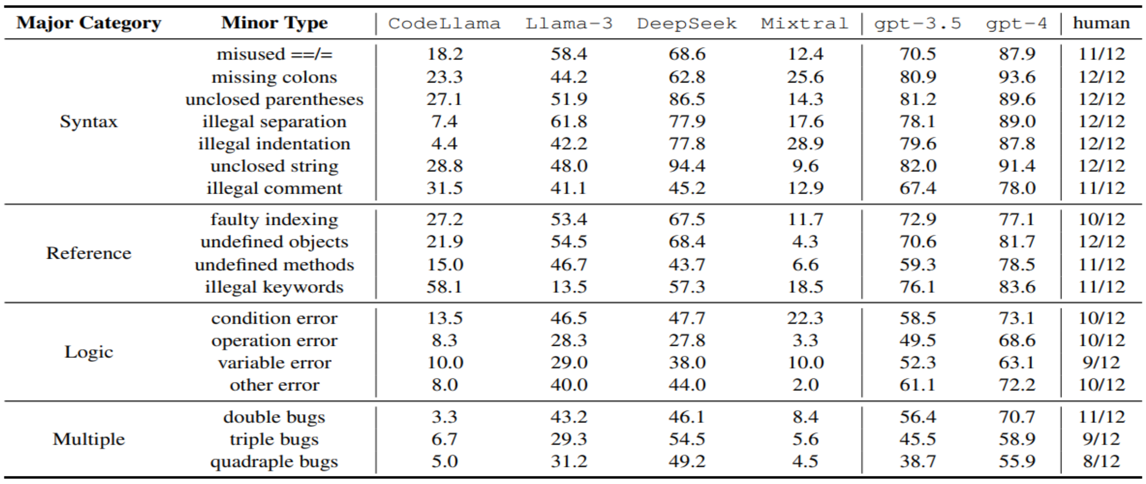
- **Method:** The researchers created DebugBench by implanting bugs into code solutions using LLMs, ensuring a diverse range of errors. These bugs were then manually validated to maintain dataset quality. The study assessed the debugging performance of six LLMs through structured experiments, analyzing how well they could identify and fix different types of bugs.

- **Metrics Used:**
  - **Pass Rate:** The percentage of instances where the model-generated fix successfully resolved the bug.
  - **Error Type Performance:** Performance was categorized based on bug types: Syntax, Reference, Logic, and Multiple.

    - **Comparison to Humans:** LLMs were evaluated against human performance to gauge their effectiveness.
- **Pass Criteria:**
    - **Test Case Validation:** Each buggy program has predefined test cases.
    - **Success Definition:** A debugged solution is considered successful if it passes **all** test cases without errors.
    - **Binary Evaluation:**
        - If the model-generated fix passes **all** test cases, it is marked as a **successful debug**.
        - If it fails **any** test case, it is considered an unsuccessful attempt.
    - **Impact of Bug Type on Pass Rate:**
        - Syntax and reference errors have a higher pass rate because they involve fixing structural issues.
        - Logical and multiple errors have a lower pass rate, as they require deeper reasoning and understanding.
    - **Comparison with Human Debugging:**
        - Human developers consistently achieve a higher pass rate than LLMs.
        - LLM performance varies significantly depending on error complexity.

Results

- **Performance Comparison:**
    - Closed-source models (GPT-4, GPT-3.5) outperform open-source models (CodeLlama, Llama-3, DeepSeek-Coder, Mixtral) but still lag behind human debugging capabilities.
    - Logical errors and multiple-error scenarios were significantly more challenging to fix than syntax and reference errors.
    - Incorporating runtime feedback improved performance for syntax and reference errors but did not significantly help with logical errors.
- **Impact of Debugging Complexity:**
    - Debugging performance varied by error type; syntax errors were easiest to fix, while logical and multiple errors were the hardest.
    - Debugging and code generation performance were correlated, indicating that LLMs' ability to generate code influences their debugging skills.



Major Category	Minor Type	CodeLlama	Llama-3	DeepSeek	Mixtral	gpt-3.5	gpt-4	human
Syntax	misused ==/=	18.2	58.4	68.6	12.4	70.5	87.9	11/12
	missing colons	23.3	44.2	62.8	25.6	80.9	93.6	12/12
	unclosed parentheses	27.1	51.9	86.5	14.3	81.2	89.6	12/12
	illegal separation	7.4	61.8	77.9	17.6	78.1	89.0	12/12
	illegal indentation	4.4	42.2	77.8	28.9	79.6	87.8	12/12
	unclosed string	28.8	48.0	94.4	9.6	82.0	91.4	12/12
	illegal comment	31.5	41.1	45.2	12.9	67.4	78.0	11/12
Reference	faulty indexing	27.2	53.4	67.5	11.7	72.9	77.1	10/12
	undefined objects	21.9	54.5	68.4	4.3	70.6	81.7	12/12
	undefined methods	15.0	46.7	43.7	6.6	59.3	78.5	11/12
	illegal keywords	58.1	13.5	57.3	18.5	76.1	83.6	11/12
Logic	condition error	13.5	46.5	47.7	22.3	58.5	73.1	10/12
	operation error	8.3	28.3	27.8	3.3	49.5	68.6	10/12
	variable error	10.0	29.0	38.0	10.0	52.3	63.1	9/12
	other error	8.0	40.0	44.0	2.0	61.1	72.2	10/12
Multiple	double bugs	3.3	43.2	46.1	8.4	56.4	70.7	11/12
	triple bugs	6.7	29.3	54.5	5.6	45.5	58.9	9/12
	quadraple bugs	5.0	31.2	49.2	4.5	38.7	55.9	8/12

The table presents debugging performance across different error categories (Syntax, Reference, Logic, and Multiple) for various models, including open-source (CodeLlama, Llama-3, DeepSeek, Mixtral) and closed-source (GPT-3.5, GPT-4), compared to human performance. The results indicate that:

- **Syntax errors** were the easiest to fix, with GPT-4 achieving the highest success rate, closely approaching human-level performance.
- **Reference errors** were more challenging, but GPT-4 and GPT-3.5 still outperformed open-source models.
- **Logical errors** showed the most significant performance gap, with all models struggling, reinforcing the need for improved reasoning capabilities in LLMs.
- **Multiple errors** were the hardest to debug, further emphasizing the compounding difficulty of complex bug scenarios.

  These findings highlight the superiority of closed-source models over open-source alternatives, but also illustrate that LLMs still lag behind human debugging capabilities, particularly for complex errors.

Reflection

Contributions

- DebugBench provides a large-scale and diverse dataset for evaluating LLM debugging performance.
- The study highlights key challenges faced by LLMs in logical error detection and correction.

- Findings suggest that runtime feedback can assist with debugging, though its effectiveness depends on error type.

Limitations

- LLMs still struggle with logical errors due to limited reasoning abilities.
- The dataset is synthetic, meaning real-world debugging challenges may not be fully captured.
- Open-source models perform significantly worse, likely due to data scarcity during training.

Future Research

- Improving LLM reasoning and contextual understanding to enhance logical error debugging.
- Developing hybrid human-AI debugging approaches for more effective solutions.
- Expanding DebugBench with real-world debugging scenarios to enhance evaluation robustness.

Paper 5

Improving LLM Classification of Logical Errors by Integrating Error Relationship into Prompts

Detecting Logical Errors in Programming Assignments Using Code2Seq

Idea

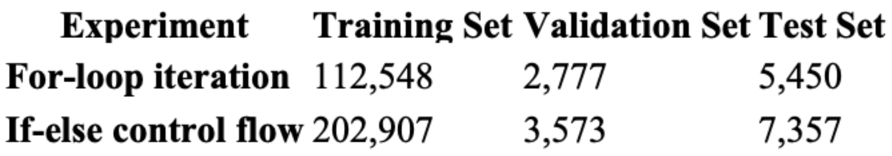
Evaluating code takes a significant amount of time, especially with the increasing number of students, making it essential for teachers to have tools that assist with the assessment process. Detecting logical errors is even more time-consuming compared to syntax errors, which compilers can easily identify. Given the importance of compilers in catching syntax errors, having an automated system capable of evaluating logical errors efficiently would be highly beneficial. To address this challenge, this experiment utilized a **machine learning model called Code2Seq**, which automatically analyzes code and detects **logical errors**. The goal is to **reduce the workload on teachers** by providing an efficient and automated way to evaluate students' code

Methodology

Two separate experiments were conducted on Java code, where logical errors were manually introduced. The goal was to test whether the model could detect errors after being trained on a dataset with deliberately embedded mistakes. Instead of using only correct code, **50% of the dataset was modified** to contain errors, ensuring that the model learns effectively during training

Dataset

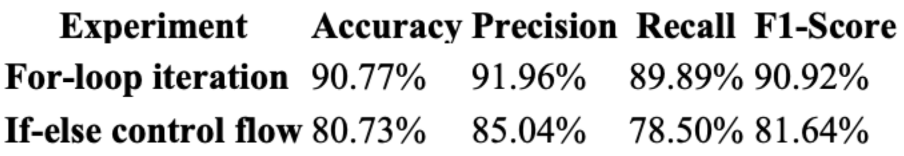
A total of 9,500 open-source Java projects from GitHub were used in this study. Real-world code was utilized to enhance the practicality and applicability of the model, ensuring it performs well in real coding environments. Instead of using artificially generated code, authentic code was chosen to improve the reliability of the results



Experiment	Training Set	Validation Set	Test Set
For-loop iteration	112,548	2,777	5,450
If-else control flow	202,907	3,573	7,357

Evaluation Metrics

The key evaluation metrics used were **accuracy, precision, recall, and F1-score**, each measuring a different aspect of the model's performance. Since relying solely on accuracy can be misleading, **four different metrics** were used to ensure a more **comprehensive and well-rounded evaluation** of the model's effectiveness



Experiment	Accuracy	Precision	Recall	F1-Score
For-loop iteration	90.77%	91.96%	89.89%	90.92%
If-else control flow	80.73%	85.04%	78.50%	81.64%

Results

The model demonstrated **high performance** in detecting logical errors, as shown in the results table, particularly in the **for-loop experiment**. This indicates that **Code2Seq** can be effectively utilized as an **automated tool** to assist in detecting students' logical errors, reducing the need for manual assessment

Limitations

The model was developed to detect only specific types of logical errors, such as for-loop initialization mistakes and incorrect if-else structures. While the model performs highly efficiently for these types of errors, its performance may not be as strong when dealing with more complex logical errors. Additionally, this study does not cover all possible logical errors, meaning that its applicability is limited to certain error types.

Suggestions for Future Research

One suggestion for future research is to enhance the model's capability to detect more complex logical errors beyond basic for-loop and if-else structure mistakes. This improvement can be achieved by training the model on more complex datasets while ensuring that the training data remains based on real-world project code rather than artificially generated examples. Using authentic code will help maintain practicality and reliability, ensuring that the model can effectively generalize to real-world programming scenarios.

References

1. LLMs cannot find reasoning errors, but can correct them given the error location
   https://arxiv.org/abs/2311.08516

   and BIG-Bench-Mistake Publicly available at https://github.com/WHGTyen/

2. Decoding Logic Errors: A Comparative Study on Bug Detection by Students and Large Language Models

   https://dl.acm.org/doi/abs/10.1145/3636243.3636245

3. Code Linting using Language Models

   https://arxiv.org/abs/2406.19508

4. Debug Bench: Evaluating Debugging Capability of Large Language Models

   https://arxiv.org/abs/2401.04621

5. Detecting Logical Errors in Programming Assignments Using Code2Seq
   https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1784369&dswid=-5789