AI Bug Detector For Python

Team Members_ Nevin Alex Varghese
Snehit T Shiju
Aswin K Reji

College Mentor- DR. S.SHIRLY

Table of contents

AI Bug Detector For Python

Abstract

The AI Bug Detector is a smart debugging application that aims to assist programmers in locating and correcting errors in Python code in an efficient manner. It utilizes artificial intelligence (AI) and machine learning (ML) to inspect code, detect errors, and offer instant feedback. The system identifies several categories of errors, such as syntax errors, runtime errors, logic errors, and indentation errors, to provide developers with concise explanations and recommendations for correcting their code.

To improve performance, the tool has an LRU (Least Recently Used) Cache, which accelerates repeated debugging operations. It also has a simple interface constructed with Gradio, through which users can easily provide their code and get instant debugging output. The system also has visual reports in Plotly, which show patterns of errors and enable users to comprehend frequent errors.

Intended for students, new programmers, and professionals, the AI Bug Detector makes the debugging process easier, cuts down on time, and enhances coding capabilities. The application is scalable and may be extended to accommodate other programming languages in the future, making it an invaluable coding education and professional tool.

Problem Statement

This project, titled **"Bug Detection and Fixing,"** focuses on developing a machine learning system that can automatically detect errors in code and provide accurate suggestions to fix them. The main goal is to build or fine-tune a powerful AI model that can read and understand source code just like a human programmer, identify different types of bugs such as syntax errors, runtime issues, or logical flaws, and then recommend possible corrections.

The process begins with collecting a large set of code examples, labeling them as either **“buggy”** or **“bug-free,”** and including corrected versions of buggy code to train the model effectively. The project uses **Python** as the main development language, along with tools like **PyTorch** or **TensorFlow** for building and training the machine learning models.

The system is designed with a strong focus on **accuracy**, **performance**, and **user-friendliness**, so that even beginners can benefit from it. Once completed, this tool will serve as an intelligent debugging assistant for developers—helping them identify issues quickly, understand what went wrong, and apply the right fixes efficiently. It aims to save time, reduce frustration, and improve the overall quality of code.

Introduction

Debugging is a critical aspect of programming, but it is usually time-consuming and difficult, particularly for new programmers. The AI Bug Detector is an intelligent debugging tool that assists programmers in quickly detecting and correcting errors in Python code. Through the use of Artificial Intelligence (AI) and Machine Learning (ML), the tool examines code, identifies typical errors, and gives users extensive feedback.

The tool is developed with the Qwen2.5-Coder-1.5B-Instruct AI model, which smartly scans Python code to identify errors like syntax errors, logic errors, runtime errors, and type errors. For better user experience, the tool has an interactive interface with Gradio, enabling users to copy and paste their code, press a button, and get immediate feedback. It also uses an LRU (Least Recently Used) Cache for quicker processing and visual reports with Plotly to show error patterns in a better way.

Proposed Solution

To solve the issue of finding and correcting bugs in source code, we suggest a machine learning-based solution known as AI Bug Detector. The system utilizes a pre-trained large language model, Qwen2.5-Coder-1.5B-Instruct, that can comprehend and parse Python code. The model is embedded in a simple-to-use web interface where users can provide their code and receive immediate detailed feedback on errors.

The process works as follows:

Code Analysis with AI: When a user enters Python code, the AI model analyzes the input, identifies potential bugs, and offers debugging recommendations in natural language.

Error Categorization: The tool categorizes frequent error types like syntax errors, type errors, logic errors, and runtime errors using regular expressions. This informs users about the nature of the issue clearly.

LRU Cache for Performance: An LRU (Least Recently Used) cache is employed to cache and reuse the results of recently executed code, enhancing response time and efficiency.

Interactive UI with Gradio: The frontend is designed with Gradio, which has an interactive and simple interface where users can simply paste code, press a button, and see immediate results.

Visual Feedback with Plotly: Errors detected are displayed as bar charts with Plotly, presenting users with a clear indication of the frequency and nature of bugs in their code.

Extendable Framework: The system is designed in a modular way, so it can be extended in the future to support more programming languages or include custom error-fixing modules.

Methodology

The development of the AI Bug Detector was done using a systematic methodology involving machine learning model development, interface design for user-friendliness, and optimization of performance. The following are the steps we used to develop and test the system:

Step 1: Problem Understanding and Requirement Analysis

We began by examining the problem statement and appreciating the objectives—automatically identifying code bugs and providing potential solutions. The team collected technical requirements, researched available debuggers, and completed scope determination based on real-world applicability to developers.

Step 2: Data Collection and Preprocessing

We gathered Python code snippets from open-source repositories and online platforms. We marked each snippet as "buggy" or "bug-free," and if buggy, we added its corrected version. The data was cleaned, duplicates were eliminated, formatting was normalized, and the dataset was balanced to provide equal learning during model training.

Step 3: Model Selection and Fine-Tuning

We employed Qwen2.5-Coder-1.5B-Instruct, a transformer model that can comprehend code context and structure. The model was fine-tuned using our dataset to enhance the performance of bug detection and suggestion. It was trained to identify error patterns and produce corresponding fixes.

Step 4: Error Detection and Classification

We developed a classification system based on regular expressions (regex) to classify the errors into several categories like syntax errors, logical errors, runtime errors, and type errors. This made it possible for the tool to give precise and clear feedback depending on the type of bug found.

Step 5: User Interface Development

In order to provide ease of use, we designed an interactive interface with Gradio. Users can simply paste the code, press a button, and obtain real-time output from the model. The interface provides ease of debugging for developers and newcomers as well.

.

Step 6: Performance Optimization

To increase the speed and efficiency of the tool, we incorporated an LRU Cache that caches the recent outputs. This cache significantly reduces the processing time when there are repeated inputs and enhances the responsiveness of the tool.

Step 7: Visualization and Reporting

We employed Plotly for creating graphical visualizations to indicate the frequency and types of errors in the submissions. The visualization will give a clear view to the user regarding the most frequent mistakes and also monitor improvement with time.

Step 8: Testing and Evaluation

Standard ML evaluation metrics like precision, recall, and F1-score were used to test the model.
We compared the system with known and unseen sample code examples to check for its generalization, accuracy, and reliability in real-world scenarios

Technology Used

Programming Language: Python

❑ Used for its extensive libraries, ease of use, and strong support for AI and ML applications.

AI Model: Qwen2.5-Coder-1.5B-Instruct

❑ A Transformer-based AI model trained to analyze Python code, detect errors, and suggest fixes.

ML Frameworks: PyTorch / TensorFlow

❑ Used for training and fine-tuning the AI model, optimizing performance, and handling large datasets.

Frontend: Gradio

❑ Provides a simple, web-based interface where users can input code and receive debugging feedback.

Visualization: Plotly

❑ Generates visual reports on common error types and debugging trends for better insights.

Storage: SQLite / JSON / CSV *(If Applicable)*

❑ Stores debugging results, logs, and datasets for efficient processing and retrieval.

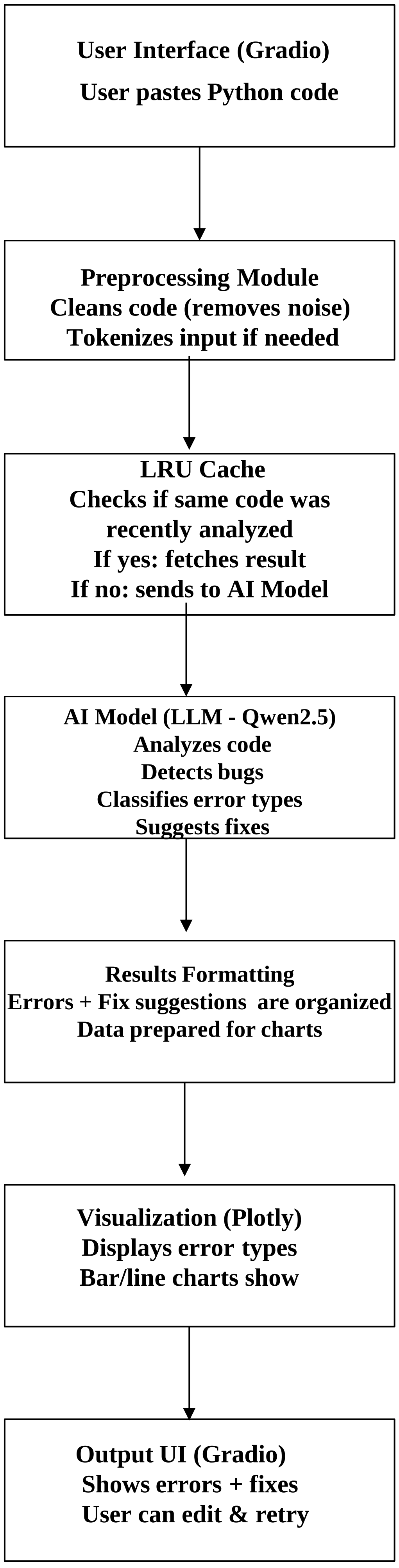
Caching: LRU Cache

❑ Improves performance by storing and reusing previously analyzed code results.

Deployment: Docker / Cloud

❑ Ensures scalability and accessibility by hosting the model on Docker or cloud platforms like AWS, Azure, or GCP

System Architecture: Code Flow and Component Interaction



Data Pipeline: Data Collection, Preprocessing, and Storage

The data pipeline in the AI Bug Detector project plays a crucial role in preparing the dataset used for training and testing the machine learning model. It ensures a smooth flow from raw code collection to usable structured data, enhancing the overall performance and accuracy of the system.

•Data Collection: Code snippets are gathered from open-source repositories like GitHub, online coding platforms, and manually written examples. These samples include both buggy and bug-free code to help the model learn effectively.

•Labeling: Each piece of code is tagged either as “buggy” or “bug-free.” If it contains bugs, a corrected version is provided. This helps the model learn to detect errors and suggest possible fixes accurately.

•Preprocessing: The collected data is cleaned and standardized. Tokenization and parsing (using Abstract Syntax Trees, if necessary) are applied to convert code into a machine-readable format, while also removing duplicates or incomplete entries.

•Storage Solution: Preprocessed data is stored using flexible formats such as CSV, JSON, or lightweight databases like SQLite, allowing easy access and management during training and evaluation.

•Caching: An LRU (Least Recently Used) Cache is implemented to remember frequently checked code snippets and their results, improving the speed and responsiveness of the system during repeated queries.

Team Contributions

Nevin (Member 1) – Model Development and AI Integration

- ➢ In charge of researching and incorporating the Qwen2.5-Coder-1.5B-Instruct model.
- ➢ Managed the training and fine-tuning of the AI model for bug detection.
- ➢ Worked on creating the logic to classify errors and fix suggestions.

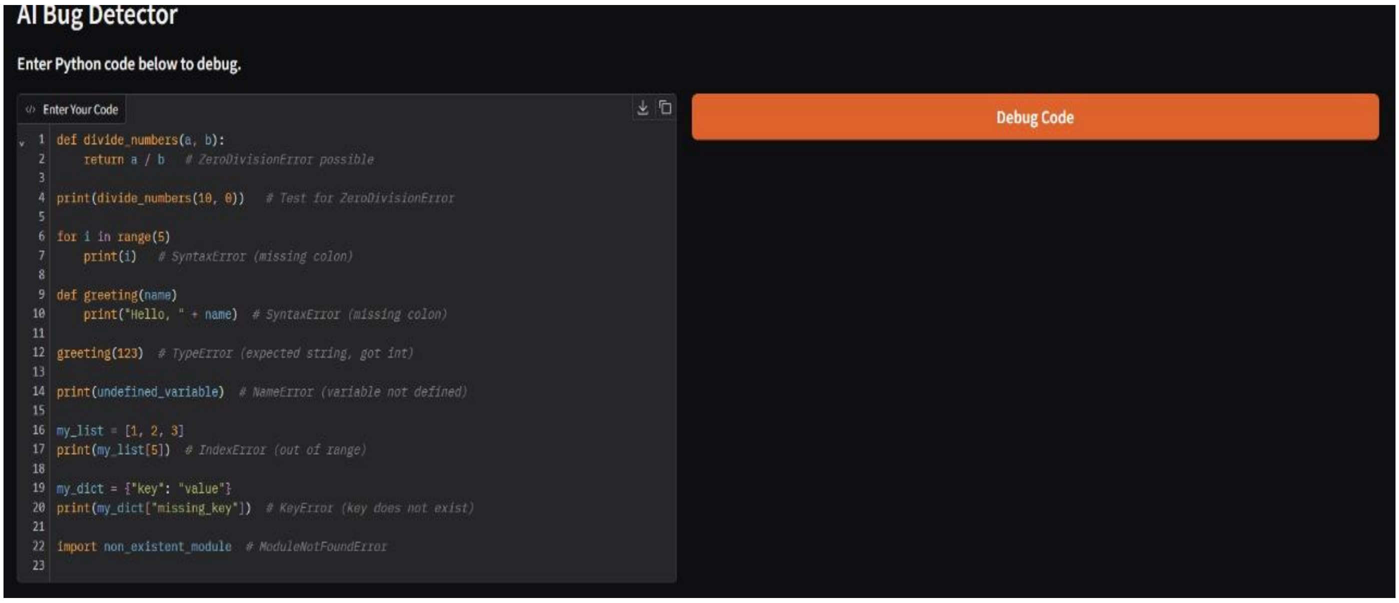
Aswin (Member 2) – Frontend Development & User Interface

- ➢ Developed and created the interactive Gradio UI.
- ➢ Integrated AI model into the frontend for providing real-time feedback.
- ➢ Included visualization tools with Plotly to depict error patterns properly.

Snehit (Member 3) – Data Gathering, Preprocessing & Testing

- ➢ Directed efforts to gather, clean, and label code snippets.
- ➢ Designed pre-processing and data formatting scripts for the AI model.
- ➢ Conducted testing, assessment, and optimization with caching (LRU Cache) and performance measures.

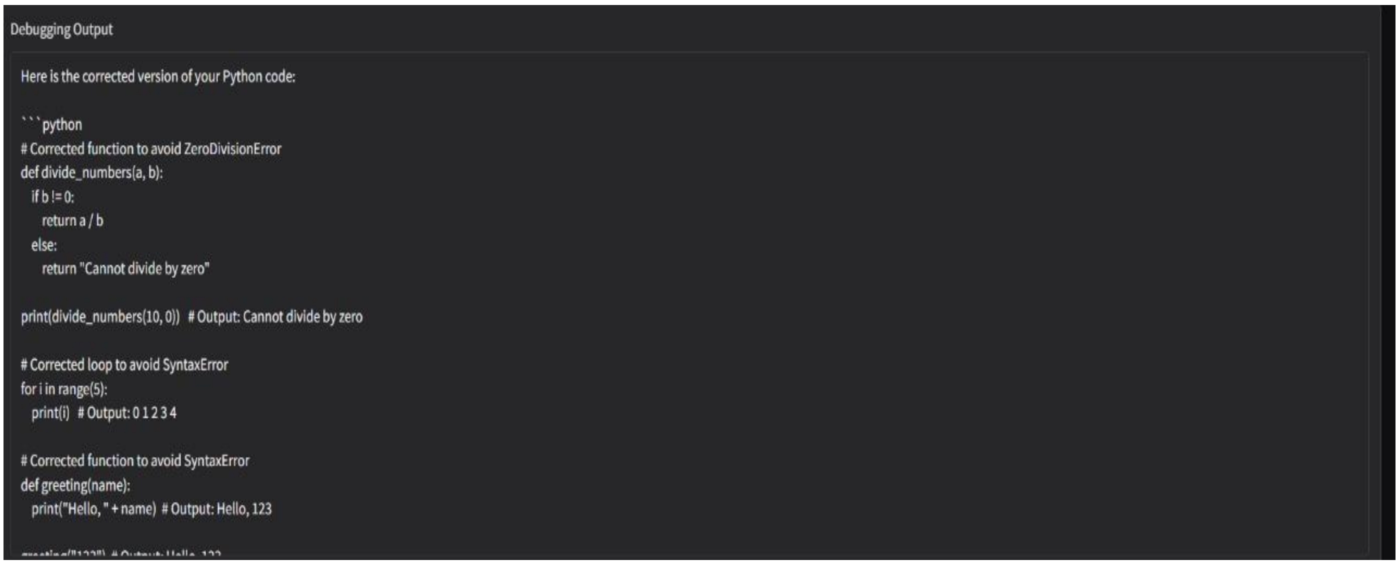
Output screenshots



AI Bug Detector

Enter Python code below to debug.

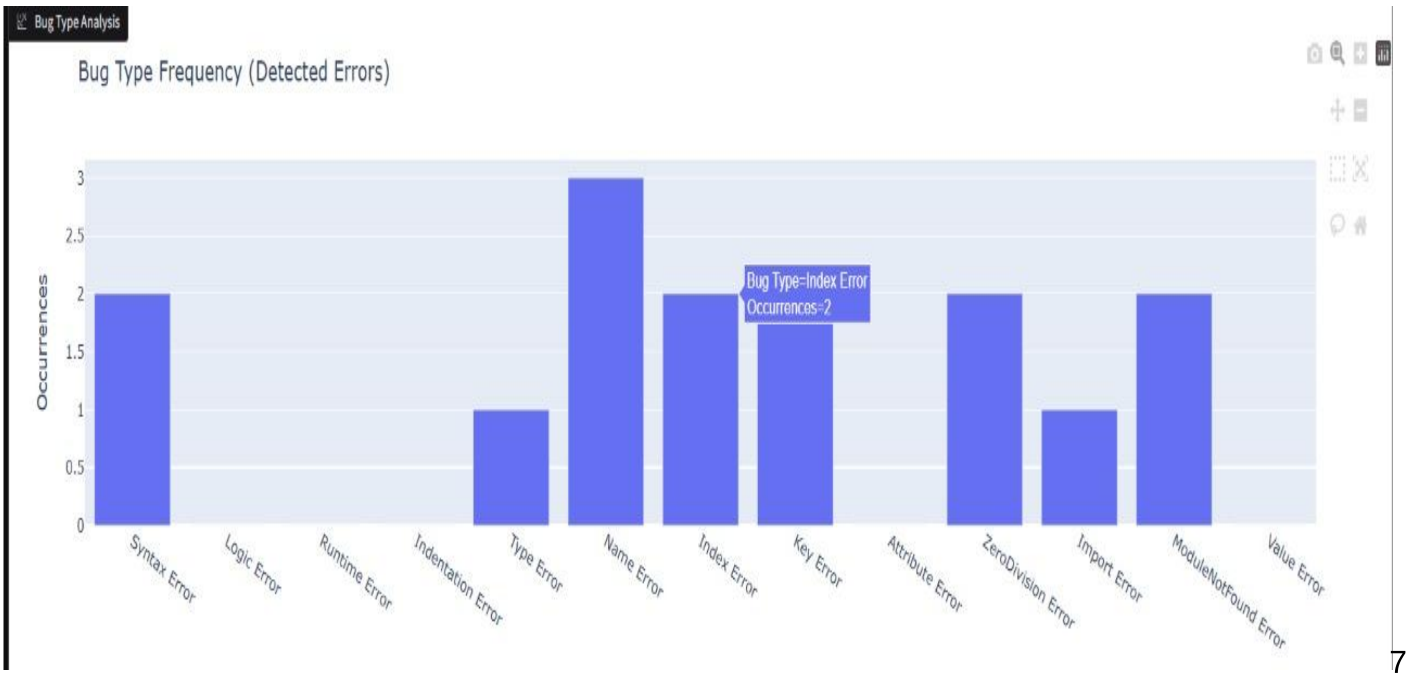
```python
def divide_numbers(a, b):
    return a / b   # ZeroDivisionError possible

print(divide_numbers(10, 0))   # Test for ZeroDivisionError

for i in range(5)
    print(i)   # SyntaxError (missing colon)

def greeting(name)
    print("Hello, " + name)  # SyntaxError (missing colon)

greeting(123)  # TypeError (expected string, got int)

print(undefined_variable)  # NameError (variable not defined)

my_list = [1, 2, 3]
print(my_list[5])  # IndexError (out of range)

my_dict = {"key": "value"}
print(my_dict["missing_key"])  # KeyError (key does not exist)

import non_existent_module  # ModuleNotFoundError

```



Debugging Output

Here is the corrected version of your Python code:

~~~
```python
# Corrected function to avoid ZeroDivisionError
def divide_numbers(a, b):
  if b != 0:
    return a / b
  else:
    return "Cannot divide by zero"

print(divide_numbers(10, 0))  # Output: Cannot divide by zero

# Corrected loop to avoid SyntaxError
for i in range(5):
  print(i)  # Output: 0 1 2 3 4

# Corrected function to avoid SyntaxError
def greeting(name):
  print("Hello, " + name)  # Output: Hello, 123

greeting("123")  # Output: Hello, 123
~~~



Remarks

- To run this code, make sure to install the dependencies listed in requirements.txt .The requirements.txt is upload in source code folder in github
- Since this project was developed and tested on a **CPU-based laptop** (without GPU acceleration), the **model execution time is slightly longer**.

Project References

1. Hugging Face Transformers

•Hugging Face, *Transformers: State-of-the-art Natural Language Processing for Pytorch and TensorFlow 2.0*, https://huggingface.co/docs/transformers

2. Qwen2.5-Coder-1.5B-Instruct Model

•Qwen Team, *Qwen2.5-Coder-1.5B-Instruct Model Card*, Hugging Face, https://huggingface.co/Qwen/Qwen2.5-Coder-1.5B-Instruct

3. Gradio

•Gradio Team, *Gradio: Build Machine Learning Web Apps in Python*, https://www.gradio.app

•Gradio Documentation, https://www.gradio.app/docs

4. PyTorch

•PyTorch Team, *PyTorch: An Open Source Machine Learning Framework*, https://pytorch.org

5. Plotly & Pandas (for Visualization)

•Plotly, *Plotly Python Open Source Graphing Library*, https://plotly.com/python

•Wes McKinney, *pandas: powerful Python data analysis toolkit*, https://pandas.pydata.org