Code Analyzer Documentation

Overview

The **Code Analyzer** is a Python-based application designed to analyze coding submissions from the Examly platform, providing detailed insights into code quality, test case performance, and adherence to requirements. Built with a Streamlit interface, it integrates with the Examly API to fetch student answers and leverages the Azure OpenAI API for advanced code analysis. The application supports multiple analysis focuses, such as identifying test case failures, logical errors, or missing functionality, and generates a structured report for each submission.

Table of Contents

Features

- **Code Retrieval**: Fetches student coding answers from the Examly API using a provided URL and authorization token.
- **Flexible Analysis**: Supports predefined analysis focuses (e.g., test case failure, logical errors, edge cases) and custom analysis prompts.

- **Detailed Reports**: Generates structured reports with sections for final score, test case analysis, code structure analysis, and AI-driven insights.
- **Streamlit Interface**: Provides a user-friendly web interface for inputting parameters and viewing results.
- **Downloadable Reports**: Allows users to download analysis reports as text files.
- **Azure OpenAI Integration**: Uses the Azure OpenAI API for in-depth code analysis, offering actionable recommendations.

System Architecture

The Code Analyzer is composed of several interconnected modules:

1. **Streamlit Frontend**: Facilitates user interaction through a web interface for inputting URLs, tokens, and analysis prompts, and displaying results.
2. **Examly API Client**: Retrieves coding answers and question data using the Examly API.
3. **Azure OpenAI API**: Analyzes code for correctness, structure, and adherence to requirements.
4. **File Handler**: Saves analysis reports to a local file (`analysis_report.txt`).
5. **Report Parser**: Splits reports into sections (Final Score, Test Case Analysis, Code Structure Analysis, AI Analysis Insights) for organized display.

Dependencies

The application relies on the following Python libraries:

- `streamlit==1.12.0`: For the web-based user interface.
- `openai==0.27.8`: For interacting with the Azure OpenAI API.
- `python-dotenv==1.0.0`: For loading environment variables from a `.env` file.
- `requests==2.28.2`: For making HTTP requests to the Examly API.
- `altair==4.2.0`: For Streamlit visualization (optional, included in requirements).

Setup and Installation

1. **Clone the Repository**:

```
git clone <repository-url>
cd code-analyzer
```

2. **Install Dependencies**:

   ```
   pip install -r requirements.txt
   ```

Create a `requirements.txt` file with the following content:

```
streamlit==1.12.0
openai==0.27.8
python-dotenv==1.0.0
requests==2.28.2
altair==4.2.0
```

3. **Set Up Environment Variables**: Create a `.env` file in the project root with the following variables:

   ```
   AZURE_OPENAI_API_KEY=<your-azure-openai-api-key>
   AZURE_OPENAI_ENDPOINT=<your-azure-openai-endpoint>
   AZURE_OPENAI_API_VERSION=<your-azure-openai-api-version>
   AZURE_OPENAI_MODEL=<your-azure-openai-model>
   ```

4. **Run the Application**:

   ```
   streamlit run app.py
   ```

Replace `app.py` with the name of your main Python file containing the Streamlit code.

Usage

Inputting Details

1. **Access the Streamlit Interface**: Open the application in a browser (typically `http://localhost:8501`).
2. **Enter Details**:
   a. **URL**: Input the Examly result URL (e.g., `https://admin.ltimindtree.iamneo.ai/result?testId=...`).
   b. **Authorization Token**: Provide the Examly API token.
   c. **Analysis Focus**: Select from predefined options (e.g., "Check why the testcase failed") or choose "Custom Analysis" to enter a custom prompt.

3. **Custom Analysis** (if selected): Enter specific analysis criteria in the provided text area.

Analyzing Code

1. **Initiate Analysis**: Click the "Analyze Code" button.
    a. The system validates inputs (URL and token).
    b. Fetches coding answers from the Examly API.
    c. Analyzes the code using the Azure OpenAI API based on the selected prompt.
    d. Saves the analysis to `analysis_report.txt`.

Viewing and Downloading Reports

1. **View Analysis**: The report is displayed in expandable sections (Final Score, Test Case Analysis, Code Structure Analysis, AI Analysis Insights).
2. **Download Report**: Use the "Download Analysis Report" button to save the report as a text file.
3. **Error Messages**: If the analysis fails, an error message is displayed (e.g., invalid URL, API failure).

Key Components

CodeExtractor

- **Class**: `CodeExtractor`
- **Description**: Handles communication with the Examly API to retrieve coding answers and coordinates analysis.
- **Key Methods**:
    - `get_coding_answers(url, auth_token, analysis_prompt)`: Fetches answers for a given test ID and initiates analysis.
    - `_process_response(response_data, analysis_prompt)`: Extracts coding answers from the API response and saves analysis.
    - `_extract_answer(question)`: Retrieves answer details (language, filename, content) from question data.
    - `_get_filename(language)`: Maps programming languages to appropriate file extensions.
- **Output**: List of coding answers and a success flag.

GPTAnalyzer

- **Class**: `GPTAnalyzer`
- **Description**: Analyzes code using the Azure OpenAI API, focusing on test case performance, code structure, and custom criteria.
- **Key Methods**:
  - `analyze_code(code_content, question_data, analysis_prompt)`: Orchestrates the analysis process.
  - `_get_test_score_from_question(question_data)`: Calculates the test score from question data.
  - `_extract_requirements(question_data)`: Extracts question requirements (text, input/output format, constraints).
  - `_extract_test_cases(question_data)`: Retrieves sample and actual test cases.
  - `_analyze_code_structure(code_content, requirements, solution)`: Identifies structural issues and whitelist violations.
  - `_run_test_case_analysis(test_cases, actual_score)`: Analyzes test case performance.
  - `_get_gpt_insights(code_content, requirements, analysis_prompt, actual_score)`: Generates AI-driven insights.
  - `_format_analysis_report(test_results, code_analysis, gpt_insights, requirements)`: Formats the final report.
- **Output**: Formatted analysis report as a string.

FileHandler

- **Class**: `FileHandler`
- **Description**: Manages file operations for saving analysis reports.
- **Key Methods**:
  - `save_analysis(coding_answers, analysis_prompt, analyzer)`: Writes analysis reports to `analysis_report.txt`.
- **Output**: Saves reports with question details, code, and analysis.

Report Parsing

- **Function**: `parse_report(report_text)`
- **Description**: Splits the analysis report into sections (Final Score, Test Case Analysis, Code Structure Analysis, AI Analysis Insights) for structured display.
- **Process**:
  - Identifies section headers in the report text.
  - Groups lines under the appropriate section.

- **Output**: Dictionary with section names as keys and content as values.

File Structure

- `app.py`: Main Streamlit application file (assumed name).
- `analysis_report.txt`: Stores generated analysis reports.
- `.env`: Environment variables for Azure OpenAI API settings.
- `code_extractor.py`: Contains the `CodeExtractor` class.
- `gpt_analyzer.py`: Contains the `GPTAnalyzer` class.
- `file_handler.py`: Contains the `FileHandler` class.
- `config.py`: Defines configuration for Azure OpenAI API.

Error Handling and Logging

- **Logging**: Uses `print` statements with `DEBUG` prefixes for debugging (e.g., API responses, question data).
- **Error Handling**:
  - API requests handle `requests.exceptions.RequestException` and non-200 status codes.
  - JSON parsing errors are caught during test case extraction.
  - File operations include try-except blocks for reading/writing errors.
  - Streamlit displays user-friendly warnings for missing inputs or errors.
  - Azure OpenAI API errors are logged with tracebacks and return fallback messages.
- **Log Output**: Debug messages for API responses, question data, and analysis steps.

Limitations

- **Hardcoded API Endpoint**: Assumes a specific Examly API endpoint, limiting adaptability to other platforms.
- **File-based Storage**: Reports are saved locally, which may not scale for multiple users.
- **Debug Logging**: Relies on `print` statements instead of a proper logging framework.
- **Language Support**: Limited to predefined languages (Java, Python, C#, JavaScript, SQL) for filename mapping.
- **Error Messages**: Some error messages (e.g., API failures) may lack specificity for end users.