Developer Test

For **Third Wish Group**

March 3, 2025

Contents

1 Objective

Develop a Python application that streamlines the processing, mutation, and management of arbitrary problem solvers extracted from a `problems.txt` file. The application should be robust enough to handle structured workflows resembling the layout below, while integrating argument parsing, data handling, mutation operations, and result tracking.

2 Background

Automating the processing and iteration of data or problem sets is crucial in software development and machine learning workflows. This test measures your capacity to design and implement a sophisticated Python application that systematically manages and mutates problem statements. The goal is to deliver a project that is achievable within one week for a skilled developer, while maintaining sufficient complexity to distinguish top-tier talent.

3 Requirements

3.1 Project Structure & Setup

Maintain a clear, organized directory structure:

- `problems/`: Contains `problems.txt` and relevant data files.
- `output/`: Stores the processed and mutated versions of problems.
- `prompts/`: Holds prompt templates guiding mutations.
- `scripts/`: Hosts auxiliary scripts (shell scripts or others).

3.2 Dependencies

Use these Python packages:

- `argparse` for command-line arguments.
- `dataclasses` for structured data.
- `os`, `random`, `re`, `subprocess`, `time`, `uuid` for core system operations.
- `yaml` for configuration and results.
- `nbformat` for Jupyter notebook handling (if relevant).
- `openai` for OpenAI API usage.
- `typing` for type annotations.

3.3 Core Functionalities

3.3.1 Argument Parsing

Incorporate flags and parameters for fine-tuning the workflow:

- `--seed`: Integer seed for random operations.
- `--agent`: Specifies AI agent (e.g., `gpt-4`).

- `--num_rounds`: Number of processing rounds.
- `--num_problems`: Number of problems to process each round.
- `--topk_problems`: Number of top problems retained per round.
- `--mutate_on_start`: Whether to mutate at the beginning of execution.

3.3.2 Data Handling

Loading Problems Read and parse the list of problem statements from `problems.txt`.

Storing Processed Outputs Keep mutated problem statements in `output/`, with unique naming or IDs.

3.3.3 Mutation Mechanism

- Define various mutation strategies: `rephrase`, `expand`, `simplify`, `add_constraints`.
- Implement dedicated functions for each strategy.
- Use templates in `prompts/mutations/` to guide these via the OpenAI API.

3.3.4 Processing Workflow

Initialization

- Load the initial set of problems from `problems.txt`.

Processing Rounds Repeat for each round:

a. Select a subset of problems based on specified criteria.

b. Apply one or more mutations to create new variants.

c. Evaluate the results using a defined metric or simulated scoring.

d. Record those evaluations to update the leaderboard.

e. Retain the highest-scoring `k` problems, remove others.

Concurrency Management Handle any spawned processes (e.g., Docker containers) responsibly. Ensure proper teardown to avoid resource waste.

3.3.5 Result Tracking

- Maintain a YAML-based `leaderboard.yaml` to track scores and problem identifiers.
- Collect logs reflecting each mutation step, including errors or warnings.

3.4 Error Handling

Include defensive measures for:

- Missing or invalid `problems.txt`.
- OpenAI API call failures.
- File I/O exceptions.
- Subprocess failures.

3.5 Documentation

Provide coherent explanations:

- `README.md` outlining setup and usage.
- Inline comments explaining key code sections.
- Usage examples demonstrating command-line arguments and operation modes.

3.6 Testing

Supply unit tests to validate:

- Mutation function accuracy.
- Proper file handling.
- Argument parsing correctness.

4 Deliverables

1. **Source Code**
   - `process_problems.py` implementing all core functionalities.
   - Additional modules or support scripts, if needed.
2. **Directory Organization**
   - Properly assembled folders as described above.
3. **Configuration Files**
   - Example `problems.txt` with sample data.
   - Prompt templates in `prompts/mutations/`.
   - `leaderboard.yaml` for score tracking.
4. **Documentation**
   - A comprehensive `README.md`.
   - Clarifying comments within the source.
5. **Tests**
   - Unit tests covering primary components.

5 Evaluation Criteria

Submissions will be judged based on:

- **Completeness**: Fulfillment of each outlined feature.
- **Code Quality**: Readability, structure, best practices.
- **Reliability**: Robust error handling and graceful failure modes.
- **Documentation**: Thoroughness and usefulness of the instructions.
- **Testing**: How effectively the unit tests capture key functionality.
- **Ingenuity**: Extra touches or optimizations enhancing performance or usability.

6 Submission Instructions

Submit a public repository link (GitHub, GitLab, or similar) containing:

- All the required folders and code.
- Scripts or instructions for straightforward setup and execution.
- A procedure for running the test suite.

Appendix

Note

If your environment lacks Docker or Jupyter, simulate those aspects as needed. Your overarching aim is to showcase high-level architectural organization, clarity, and maintainability within a one-week timeframe, while preserving a degree of rigor sufficient to differentiate top performers.