MLAgentIO: ML Automation and Experimentation with Large Language Models

Sonja Gievska*, Martina Toshevska*, Andrea Stevanoska*, Viktor Kostadinoski*
*Faculty of Computer Science and Engineering, Skopje, Macedonia
Email: {sonja.gievska, martina.toshevska}@finki.ukim.mk
{andrea.stevanoska, viktor.kostadinoski}@students.finki.ukim.mk

***Abstract*—A central aspect of machine learning research is experimentation, involving designing and running experiments, analyzing results, and iterating towards an improved outcome. This paper presents an autonomous system based on large language models (LLMs) that can execute ML experimentation tasks effectively. Inspired by MLAgentBench, our system automates ML workflows, including improving models, creating new architectures, and comparing different approaches. We demonstrate our system with three provided tasks: improving a sarcasm detection model, training a BERT-based model for toxic comment classification, and comparing LSTM-CNN architectures with transformer-based models. We also introduce a flexible task structure, enabling users to define their own tasks. Our results show promising improvements in model performance and highlight the system's capabilities in automating ML research.**

I. Introduction

Machine learning (ML) experimentation involves a cycle of hypothesis testing, model training, and evaluation. Traditionally, researchers manually iterate over these steps, but with recent advancements in Large Language Models (LLMs) from OpenAI, autonomous agents can now assist in the ML pipeline.

We present a framework that enables LLM-driven agents to perform ML tasks autonomously, leveraging OpenAI's LLM models. Our system can refine existing models, develop new architectures, and evaluate competing approaches. By automating these tasks, we aim to streamline ML research and reduce manual overhead, thus accelerating the pace of innovation in the field.

II. Solution Architecture

The solution consists of a modular architecture designed to facilitate ML experimentation autonomously. Figure 1 illustrates the key components.

A. Setup Phase

In the **Setup** phase, you provide the Agent with two essential parameters:

- **OpenAI API Key**: Used to authenticate and interact with the OpenAI services.
- **Assistant Model**: For example, `gpt-4o-mini`, which is the model the agent will use for its tasks.

Once these parameters are provided, the agent sets up its components, each requiring specific configurations:

- **Main LLM Assistant**:
  - **Parameters**:
    - **API Key**: Provided by the user to access the OpenAI API.
    - **Instructions**: These include the system/developer prompt and the specific task instructions that define the ML-based problem the assistant will be solving. The instructions describe available tools, the format of the responses, and the overall task at hand.
    - **Assistant Model**: The chosen model (e.g., `gpt-4o-mini`) is initialized here.
- **Action Parser**:
  - **No input parameters**: The Action Parser is responsible for interpreting the output from the Main Assistant. It extracts the Action and the corresponding Inputs from the assistant's response.
- **Action Executioner**:
  - **Parameters**:
    - **Action Mapping**: A map that links actions (from the Assistant's output) to corresponding Python functions.
    - **Supporting Assistant**: An additional assistant used by the Action Executioner to execute specific actions, such as "Understand File" and "Edit Script (AI)".
- **Evaluator**:
  - **No input parameters**: The Evaluator evaluates the performance and outcomes after the task is completed. It assesses the usage statistics for both assistants during the process.

B. Execution Phase

The **Execution** phase consists of three sub-phases: **Preparing**, **Running**, and **Finishing**.

1) Preparing Sub-Phase:

1) The agent is provided with the **Task Name** that it needs to execute.
2) A **Task Object** is created, representing the task to be performed.
3) The **Logger** is initialized to track the task's progress and record logs.
4) The **Task** is instantiated in a **Task Environment**.

5) A **Research Problem** is passed to the **Main Assistant**, and a conversation is initiated to begin the task.

2) Running Sub-Phase:

1) The **Main Assistant** returns an output.
2) The output is logged and saved in the **Logs**.
3) The **Action Parser** parses the **Action** and the associated **Action Inputs** from the output.
4) These parsed actions and inputs are passed to the **Action Executioner**.
5) The **Action Executioner** executes the action, which involves reading/writing to the **Task Environment** and, if necessary, consulting with the **Supporting Assistant**.
6) The execution results in an **Observation**, which is the outcome of the action execution.
7) This Observation is logged in the **Logs** and then sent back to the **Main Assistant**.
8) The loop continues until the **desired outcome** is achieved or the maximum number of iterations is reached.

3) Finishing Sub-Phase:

1) When the **Observation** contains the **Terminating Flag**, the agent transitions into the **Finishing** sub-phase.
2) During this phase, the components of the agent are shut down, and usage statistics for both assistants are evaluated by the **Evaluator**.
3) These statistics are then saved in the **Evaluations**.
4) A **Task Result** is returned, marking the end of the task.

C. Termination Phase

In the **Termination** phase, the agent is fully terminated. All components are shut down, and the process is complete.

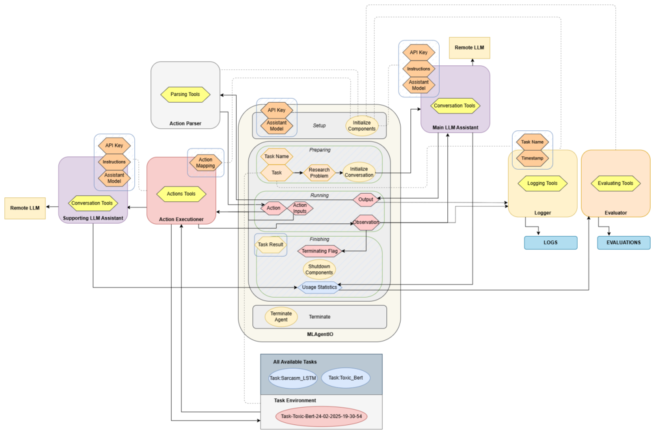

Fig. 1. Solution Architecture Overview

III. Actions Available to the Agent

The Agent is equipped with several actions that it can perform to interact with its environment and execute tasks. These actions are highly inspired by those used in [1] but have been significantly improved for better usability and precision:

The following is a list of actions that the **Main Assistant** can access and use:

A. List Files

The **List Files** action allows the agent to navigate the file system and list files in a directory.

- **Action Input**:

  { "dir_path": [a valid relative path to a directory, such as "." or "folder1/folder2"] }
- **Observation**: A list of files and folders in the specified directory, or an error message if the path is invalid.

B. Inspect Script Lines

The **Inspect Script Lines** action allows the agent to inspect specific parts of a Python script, especially useful for debugging.

- **Action Input**:

  { "script_name": [a valid python script name with relative path if needed], "start_line_number": [valid line number], "end_line_number": [valid line number or null] }
- **Observation**: The content of the script between the specified line numbers, or an error message if the script doesn't exist.

C. Understand File

The **Understand File** action allows the agent to read and comprehend the contents of a file, focusing on specific aspects.

- **Action Input**:

  { "file_name": [a valid file name with relative path if needed], "things_to_look_for": [a detailed description of what to focus on and what should be returned] }
- **Observation**: A detailed description of relevant content and lines in the file, or an error message if the file does not exist.

D. Execute Script

The **Execute Script** action allows the agent to execute an existing Python script.

- **Action Input**:

  { "script_name": [a valid python script name with relative path if needed] }
- **Observation**: The output or any errors generated by the execution of the script.

E. Edit Script (AI)

The **Edit Script (AI)** action enables the agent to describe a series of steps for editing a Python script, which is then executed by a supporting AI.

- **Action Input**:

  { "script_name": [a valid python script name with relative path if needed], "edit_instruction": [detailed step-by-step description of the edit], "save_name": [a valid file name with relative path if needed] }
- **Observation**: The edited content of the script, or an error message if the script doesn't exist. The agent can re-edit if the changes are not correct.

F. Final Answer

The **Final Answer** action is used to provide the final response to the task the agent is working on.

- **Action Input**:
  { "final_answer": [detailed description of the final answer], "goal_achieved": [true/false based on the completion of the goal] }
- **Observation**: A structured summary of the final answer provided.

These actions are integral to the agent's ability to perform its tasks, allowing it to interact with files, execute scripts, and process information effectively in the ML context.

IV. Response Format (The Output of the Assistant)

The response format of the **Main Assistant** follows a structured and standardized format to ensure clarity and consistency. The format is inspired by the one used in [1] but has been significantly improved for better usability and precision. The paper is cited below:

The Assistant's responses are structured in the following format:

A. Reflection

This section provides an interpretation of the previous observation. If there is an error, it includes the cause of the error and offers suggestions for debugging. The purpose of this section is to interpret the results, analyze any issues, and offer solutions to potential problems.

B. Research Plan and Status

Here, the high-level research plan is outlined, accompanied by a brief annotation of the current status and confirmed results of each step. Updates are provided for each task performed so far. If there has been progress, the update text is enclosed in **double asterisks** to highlight the new information. If there is no update, the research plan from the previous step is simply copied over, retaining all prior high-level steps unless intentionally revised.

C. Fact Check

This section lists all objective statements from the research plan and status. Each statement is checked for accuracy, specifying whether it is guessed or confirmed by the observation provided in the previous section. Performance-related numbers can only be confirmed by running code and observing the actual output.

D. Thought

In this section, the **Main Assistant** describes its current actions, the next steps to take, and the rationale behind these actions. This section gives a clear indication of the thought process driving the task execution.

E. Action

This is the action that the **Main Assistant** intends to take, based on the task at hand. The action must be one of the tool names available.

F. Action Input

The input to the selected action is provided in valid JSON format. This part must strictly adhere to the required input format without any additional comments.

This structured response format helps ensure that each step of the task execution is clear, traceable, and consistent with the goals outlined in the research plan.

V. Tasks Description

We provide three initial ML tasks to showcase the system's capabilities:

- **Sarcasm_LSTM**: Improve the baseline accuracy of an LSTM-based sarcasm detection model by 5%. This demonstrates the agent's ability to enhance an existing model.
- **Toxic_BERT**: Train a BERT-based model for toxic comment classification using the transformers library. This highlights the agent's ability to build models from scratch.
- **Informal_Multi_Modal**: Compare two architectures: an LSTM-CNN model vs. a transformer-based model for informal text classification. This demonstrates the agent's ability to evaluate different architectures.

VI. Task Structure

Each task follows a structured format:

- Task Name and Description
- Required Datasets and Preprocessing Steps
- Model Architecture Details
- Training and Evaluation Metrics
- Expected Improvements or Goals

Tasks are executed in an isolated environment, ensuring reproducibility and consistency.

VII. Test-Agnostic Environment

The framework supports a test-agnostic environment, allowing tasks to be executed independently. The environment encapsulates all dependencies, ensuring smooth execution without requiring manual intervention.

VIII. User-Defined Task Creation

Users can define new tasks by specifying the required components:

- Model type (e.g., LSTM, BERT, CNN)
- Dataset sources
- Training procedures and objectives
- Evaluation criteria

The system automatically integrates new tasks into the execution pipeline.

IX. Results

Preliminary experiments demonstrate that the LLM-driven agent successfully improves ML models, generates new architectures, and performs effective comparisons. Table I summarizes key statistics from the experiments, while Figures 2 and 3 visualize various performance metrics.

Task Name	Model	Req.	Tokens	$ Spent	Success
Sarcasm_LSTM	4o-mini	40	800K	0.08	Yes
Toxic_BERT	4o-mini	50	1M	0.12	Yes
Informal_M_M	4o-mini	3	500K	0.06	Yes

TABLE I
Summary of Experiment Statistics

Figure 2 presents a bar chart comparing the number of requests and tokens spent across different tasks. Notably, *Sarcasm_LSTM* exhibits a high token usage per request, suggesting a more complex processing requirement. The *Informal_Multi_Modal* task had the fewest requests but still consumed significant token resources, indicating it is a resource-intensive task.

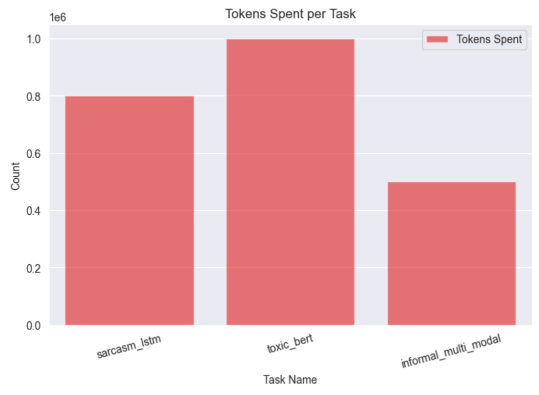

Fig. 2. Comparison of Tokens Spent per Task

Figure 3 visualizes the monetary cost associated with each task. *Toxic_BERT* incurred the highest expenditure, likely due to its high token consumption, whereas *Informal_Multi_Modal* required fewer requests but was still relatively expensive.

All tasks successfully achieved their goals. This demonstrates the robustness of the LLM-driven agent in handling diverse ML tasks effectively. These results confirm that our autonomous ML experimentation framework is capable of optimizing models and performing meaningful evaluations across various tasks.

X. Related Work: MLAgentBench

Our work builds upon [1], a benchmark suite evaluating LLM-driven agents on ML tasks. [1] assesses agent capabilities across diverse tasks, from image classification to natural language processing. Our system extends this framework by providing structured ML automation workflows.

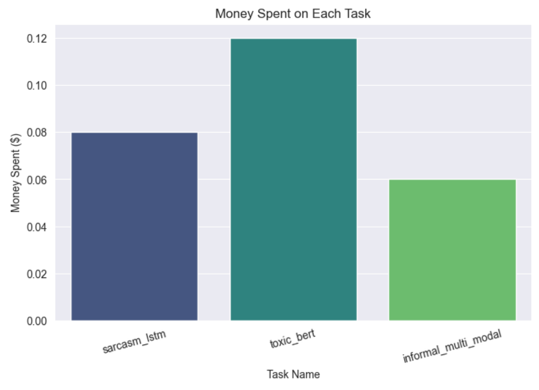

Fig. 3. Money Spent on Each Task

XI. Conclusion and Future Work

We presented an autonomous ML experimentation framework powered by OpenAI's LLMs. By automating model improvement, training, and evaluation, our system streamlines ML research. Future work includes expanding task diversity, improving reasoning capabilities, and implementing other Large Language Models (LLMs) such as LLama, Gemini, and others, to further enhance the versatility and performance of the system.

References

[1] Huang, Qian, et al. "MLAgentBench: Evaluating Language Agents on Machine Learning Experimentation." Forty-first International Conference on Machine Learning. 2024..