Proceedings of the 3rd International Conference on Software Engineering and Machine Learning
DOI: 10.54254/2755-2721/2025.30694

Streamlining Assignment Grading with Automated Categorization and Multi-Agent Systems

Duole Hong[1†], Ziyan Wei[2*†]

[1]*University of Rochester, Rochester, USA*
[2]*Haide College, Ocean University of China, Qingdao, China*
**Corresponding Author. Email: wziyan629@163.com*
[†]*These authors contributed equally to this work and should be considered as co-first authors.*

Abstract. This report addresses the growing challenge of grading student assignments in Computer Science and Statistics departments, where increasing enrollment has led to overwhelming submission volumes, time constraints, and potential inconsistencies in manual assessment. We propose an AI-driven solution leveraging R programming and OpenAI's GPT-4 model to automate key aspects of the grading process. The system employs natural language processing (NLP) and clustering algorithms to categorize student submissions, group similar responses, and generate precise feedback through multi-agent processing. Analyzing coding assignments and textual answers reduces educators' manual workload while improving grading accuracy and consistency. An accompanying R package provides tools for automated code evaluation, response clustering, and feedback generation, enabling efficient management of large-scale assessments. The system's distributed architecture allows parallel processing of submissions, significantly accelerating turnaround times. This integrated approach aims to transform educational assessment by combining AI-powered analysis with instructor oversight, freeing educators to focus on personalized instruction while maintaining rigorous evaluation standards. The solution addresses both technical implementation through customizable R functions and pedagogical considerations by preserving human-in-the-loop quality control for complex assessments.

Keywords: Grading, Automated Categorization, Multi-Agent, Homework

1. Introduction

This paper confronts a significant challenge faced by teaching assistants (TAs) and professors in the Computer Science and Statistics departments: the labor-intensive task of grading student assignments. As enrollment in these fields continues to grow, educators are inundated with a high volume of submissions, which demands considerable time and can lead to inconsistencies in grading. Our aim is to introduce solutions that optimize this process, improving both efficiency and the accuracy of assessments. Our product is tailored to the R programming language, which is extensively used in data analysis and is a staple in Statistics education. It provides a comprehensive set of tools to assist TAs and professors in the detailed evaluation of student assignments. We envision this repository as an essential resource that will streamline the grading process and alleviate

DOI: 10.54254/2755-2721/2025.30694

administrative workload. Assignments are automatically sorted based on their execution results: code that successfully meets objectives and code that contains errors and needs correction. Users can upload their criteria, which facilitates the automatic division of error-free code into smaller segments. For example, with 3000 homework submissions, professors and TAs can consolidate these assignments into one large file. Our functions enable the grouping of these assignments into clusters, allowing educators to determine the number of groups. They only need to grade assignments from each group [1-3].

Our product also incorporates the term an agent. While the term agent has become widely recognized in computer science, particularly in the context of autonomous systems and AI, we use it here in a broader sense to refer to an entity or component that performs specific tasks within our grading system.

An agent is a program or entity that can operate autonomously within a specific environment, usually with the intent of accomplishing certain objectives or duties. This concept is foundational in multi-agent systems (MAS), where several agents work together to tackle intricate challenges or execute synchronized activities [4].

Our system improves educational efficiency, precision, and general efficacy in programming and computer science instruction by automating the processes of categorization and segmentation. It is our hope that these developments signify substantial enhancements in educational methodologies, equipping educators with robust tools to oversee and evaluate coding assignments with greater ease. At present, we have yet to establish a user base. Nevertheless, we are confident that the potential of our product to transform the grading process will attract a considerable user following in the near future.

The solutions we are introducing are designed to lighten the load on educators by delivering an efficient, accurate, and simplified method for grading R programming assignments. This approach will save time and ensure a more dependable evaluation of student submissions, leading to improved educational results [5,6].

2. Literature review

Automatic categorization, also referred to as automated clustering, involves the process of automatically assigning predefined categories or labels to textual data based on its content. This is typically accomplished using machine learning algorithms and natural language processing (NLP) techniques. The objective is to efficiently organize and classify large volumes of text data into meaningful categories without the need for manual intervention.

In the context of our research, we leverage natural language processing (NLP) techniques facilitated by ChatGPT. ChatGPT is an advanced artificial intelligence model developed by OpenAI, belonging to the Generative Pre-trained Transformers (GPT) family, specifically utilizing the GPT-3.5 architecture. ChatGPT is adept at comprehending and generating human-like text in response to input queries. It excels in engaging in conversations, providing informative answers, generating coherent text, and performing various NLP tasks. Its capabilities encompass understanding contextual nuances, generating contextually appropriate responses, and offering pertinent information based on the input it receives. For our research objective, ChatGPT serves as a pivotal tool to achieve automatic categorization.

In previous studies, some people successfully studied the scoring criteria and theories in detail and developed the R package for automatic scoring. On this basis, to reduce the work pressure of teaching assistance, we have carried out further research using the ChatGPT large model, hoping

DOI: 10.54254/2755-2721/2025.30694

that users can obtain more accurate and demand-oriented answers and automated procedures with self-correction ability.

3. Methodology

The multi-agent program uses OpenAI's ChatGPT, specifically the GPT-4 model, as the central AI engine to enhance the system's ability to group student answers based on their similarities. ChatGPT plays a critical role in the program by processing and interpreting natural language prompts, allowing for precise and context-aware grouping of student responses. ChatGPT's advanced language understanding enables it to handle complex and nuanced inputs, which is essential for accurately categorizing diverse student answers. The program utilizes ChatGPT's capability to learn from a wide range of texts, ensuring it can manage various prompts and contexts effectively. This makes it superior to traditional rule-based systems and simpler machine-learning models. Therefore, we made full use of the capabilities of the AI model for automated categorizations in this paper; We attempted to write prompts to limit the AI and let it generate correct grouping results for us. Then, we expanded this approach. We create a text named 'knowledge base' for more precise external restrictions. Finally, we used Multi-Agents to develop an automated grouping software for more comprehensive operations.

4. Engineering

4.1. R to csv function

4.1.1. Purpose

The R script aims to process several R script files within a designated directory, filter out relevant code lines (ignoring comments and lines that match a given question file), and then combine them into a CSV file. This feature is beneficial for compiling student assignments or other R script collections for review and analysis.

The script is intended to be simple and easy to use. Its primary function is R_to_csv, which requires three parameters: folder_path, output_csv_file_path, and an optional question_file_path.

4.1.2. Design

The R_to_csv function is implemented to handle the following tasks:

1. Listing R Files: Retrieve all R script files in the specified folder.
2. Reading the Question File: Conditionally read the question file if provided; otherwise, initialize an empty character vector.
3. Reading and Filtering R File Contents: For each R file, read the file and filter out comments, empty lines, and lines matching the question file.
4. Creating a Data Frame: Create a data frame with StudentID and Answer columns.
5. Writing to CSV: Write the data frame to a CSV file.

The design aims to facilitate easy implementation in ChatGPT. It efficiently processes CSV files, which is more effective than reading each R file individually when handling a large volume of student responses.

DOI: 10.54254/2755-2721/2025.30694

```
# Function Definition
R_to_csv <- function (folder_path, output_csv_file_path,
                                    question_file_path = NULL) {
  # Retrieve all R script files in the specified directory
  r_files <- list.files (path = folder_path, pattern = "\\.R$" , full
                         .names = TRUE)

  # Conditionally read the question file if provided, #
  otherwise initialize an empty character vector
if ( ! is.null(question_file_path)) {
question_lines <- readLines (question_file_path, warn = FALSE)
question_lines <- trimws (question_lines) # Trim white space
} else {
question_lines <- character (0) }

# For each R file, read the file, filter out comments, # empty lines, and lines matching the question file
file_contents <- lapply (r_files, function (file) { lines <- readLines (file)
lines <- trimws (lines) # Trim white space
code_lines <- lines[ !grepl( "^\\s*#|^\\s*$", lines) & !sapply(lines,
function (line) any (grepl(line,
question_lines, fixed = TRUE)
) )
]
```

```
paste (code_lines, collapse = "\n") })

# Create a data frame with StudentID and Answer columns
student_ids <- seq_along(r_files)
r_files_df <- data.frame (StudentID = student_ids,
Answer = unlist (file_contents), stringsAsFactors = FALSE)

# Write the data frame to a CSV file
write.csv (r_files_df, output_csv_file_path, row .names = FALSE)
}
```

4.1.3. Algorithm and logic

Based on your description, it seems you have a script that processes R files to clean and standardize their content, specifically for educational purposes, possibly to prepare them for review or grading. The script performs the following tasks: File Manipulation: It reads and writes files, likely manipulating their content in some way.

String Processing: It processes the text within these files, such as removing unnecessary parts.

Regular Expressions: It uses regex to filter out comments that are often unnecessary for review or grading.

Conditional Checks: It checks for optional question files and filters out lines that are marked as original questions by the professor.

apply Function: This is a function in R that applies a given function to each element of a list. Your script processes each R file individually to ensure consistent filtering.

Whitespace Trimming removes extra spaces from the text, helping to maintain a clean and readable format.

Fixed Pattern Matching: It uses fixed patterns in regex to avoid errors due to special characters or incomplete regex patterns.

Remove AI Traces: Although the description is not entirely clear on what this means, it could imply that the script also removes any generated content that may have been produced by AI, possibly to ensure that the work submitted is the student's own.

DOI: 10.54254/2755-2721/2025.30694

4.1.4. Testing and validation

Basic testing can be performed by running the example usage provided in the script. Additional unit tests and integration tests could be developed to ensure robustness. The function's behavior can be validated by comparing the contents of the generated CSV file with the expected outputs based on known inputs.

4.1.5. Example usage

```
# Replace with the actual folder path containing R files if different
folder_path <- "ExampleSubmissions-1"

# Replace with desired csv file name
output_csv_file_path1 <- "example1 .csv" output_csv_file_path2 <- "example2 .csv"
# Replace with the actual path to the question file, or set to NULL to omit
question_file_path <- "question1 .R"

# With question file
R_to_csv (folder_path, output_csv_file_path1, question_file_path)

# Without question file
R_to_csv (folder_path, output_csv_file_path2)
```

4.1.6. Error analysis

While the current implementation of R_to_csv is functional, it has several potential flaws:

1. String Matching Sensitivity: The script employs flexible matching methods, such as trimming whitespace and using regular expressions, rather than exact string matching. This approach, however, may inadvertently exclude student answers similar to question lines. Thoughtful planning regarding the content of the question file and potential student responses is essential to reduce this risk.

2. Unexpected File Contents: The script operates under the assumption that all files in the specified directory are valid R scripts. The presence of non-R files or improperly formatted R

scripts could lead to failure or erroneous outcomes. Implementing error handling to bypass invalid files and log warnings could address this concern.

3. Robustness of Filtering Logic: The existing filtering logic may not efficiently handle certain edge cases, such as comments within code lines or multi-line comments. Strengthening regular expressions and enhancing filtering logic would improve the accuracy of extracted code lines.

4. Handling Incomplete Final Lines: The script includes a provision to manage any incomplete final lines when reading the question file by setting warn = FALSE in the readLines function. This avoids warnings about incomplete lines and ensures proper processing of the question file content.

5. Separation of Different Problems: The function does not separate different problems in the student submissions. The function does not distinguish between separate problems in student submissions. It only identifies code differences from the question, which may not suffice for submissions with multiple problemsThe project evolved through several stages: ## 6.2. Multi_agent ### 6.2.1. Development Process of the Project

The project evolved through several stages:

1. Initial Stage: Utilized ChatGPT-4 API with basic prompt engineering.

2. Enhanced Stage: Added a knowledge base to provide context, such as grading standards and questions.

3. Final Stage: Implemented a multi-agent system to modularize and enhance functionality.

DOI: 10.54254/2755-2721/2025.30694

4.2. Workflow and implementation

The implemented system's workflow can be divided into five main stages: Input Handling, Prompt Construction, API Request, Response Processing, and Output Handling.

General Structure The code is structured as a Shiny web application that utilizes several R6 classes to break down functionalities into modular components. Each R6 class is responsible for a distinct aspect of the workflow, ensuring the system is manageable and can be easily expanded. The key components involved are:

1. FileReaderAgent: Handles reading input files from the filesystem.
2. MemoryAgent: Manages memory by storing and retrieving past interactions or results.
3. LearningAgent: Adjusts behavior based on past data.
4. ApiRequestAgent: Makes API requests to OpenAI.
5. ExtendedApiRequestAgent: Inherits from ApiRequestAgent, stores results in memory, and updates behavior.
6. DataProcessorAgent: Processes the grouping result from the API.
7. ResultWriterAgent: Writes the processed results to a CSV file.
8. FeedbackAgent: Processes user feedback to improve the model.

4.2.1. Workflow

1. Input Handling:
   • The user uploads a CSV file containing student answers.
   • The FileReaderAgent will read the CSV file.
2. Prompt Construction:
   • A prompt is designed to limit the GPT to let it group students' answers according to similarity.
   • The knowledge base provided by the user acts as a prompt.
3. API Request:
   • The ExtendedApiRequestAgent will send a prompt to the OpenAI API.
   • Then, the API responds to the grouped student answers.
4. Response Processing:
   • The DataProcessorAgent will handle the API response to get the grouping information.
   • The processed data and the original student answers will combined.
5. Output Handling:
   • The ResultWriterAgent will copy the grouped student answers to a CSV file.
   • User can check the result in our shiny web.
6. Feedback Handling:
   • The user can give feedback.
   • The FeedbackAgent will generate a new answer according to this feedback and update the grouping standard.

This modular and structured approach allows for easy maintenance and scalability of the system. Each agent is responsible for a specific task, making the overall system robust and adaptable to changes.

4.2.2. Detailed illustration of the AI model and principles

GPT-4 is actually an in-house, cutting-edge language model specially developed by Liam. It is baked with the Transformer architecture, which makes it specialize in understanding and generating

DOI: 10.54254/2755-2721/2025.30694

the text information that goes into this AI model. This model is pre-trained on a voluminous amount of data, allowing it to grasp most of the linguistic nuances and trends in knowledge.

(1) Principles Behind Using GPT - 4 for Clustering

1. Natural Language Understanding:

• It is highly useful for any processing involving context-oriented understanding of text or semantic nuances.

• However, the interoperability between questions becomes at par due to the fact that with a well-engineered prompt, the model can be guided to perform kind of tasks like clustering similar answers

2. Prompt Engineering:

• The response of the model is very prompt-dependent. Prompt engineering is used to engineer the input prompt in such a way that the model performs near an optimum point for a task.

– The prompt in this system, therefore, contains a briefing on how the answers are to be grouped, the format for the output, and the knowledge base to provide context.

3. Integration with API:

• The system questions the GPT-4 model at OpenAI's API. This operation sends a request to the API containing the prompt and retrieves the returned response from the model.

– ApiRequestAgent and ExtendedApiRequestAgent are maintainers of the API requests;

they deal with the responses to the API request.

(2) Feedback Loop

• The model can learn over time from the user's feedback. The FeedbackAgent takes in the feedback and updates the memory and behavior correspondingly.

– Thus, it helps the system learn from its interactions and get more accurate over time.

4.2.3. Initialization stage

The first is a function called group answers. This takes two arguments: answers and api_key.

```
group_answers <- function (answers, api_key)
```

Arguments:

• answers: A data frame containing the student's answers, with each response on a separate row and a unique identifier for each.

• api_key: A string variable containing the OpenAI API key for authentication.

Process:

1. Prompt Construction: A prompt will be developed for the GPT-4 model with a system message that provides a setting for it, such that it is an expert in the assessment of student answers and their category. A user message detailing how answers will be grouped together for the model. The instruction states that answers are provided in this format and explains the desired format of the output.

```
messages <- list (
    list (role = "system" , content = "You are an expert in evaluating and grouping student answ list
    (role = "user" , content = paste (
      "Group the following answers into clusters based on their similarity .",
      "Provide a group number for each answer .",
      "Answer format: Answer <number>: <answer>",
      "\n\n",
      paste0("Answer ", seq_along(answers$Answer), ": ", answers$Answer, collapse = "\n"),
      "\n\nFormat the output as: Answer <number>: Group <number>\n", sep
      = "\n"
    ))
  )
```

2. API Request: It makes a request to the OpenAI API as a POST call. It consists of:

• The model name to use (gpt-4o-mini-2024-07-18).

DOI: 10.54254/2755-2721/2025.30694

• The constructed messages.

• Additional parameters such as max_tokens to control the response length and temperature to adjust the randomness of the output.

```
response <- POST(
    body = toJSON(list(
      model = "gpt-4o-mini-2024-07-18",
      messages = messages,
      max_tokens = 1500, # Adjust based on the length of the answers
      temperature = 0.7

    ), auto_unbox = TRUE))
```

3. Response Handling: The function processes the API response by:

• Checking the status code to ensure the request was successful.

• Parsing the response content to extract the grouped answers.

```
response_content <- content(response, as = "parsed", type = "application/json") if
(length(response_content$choices) == 0) {
    stop("No choices found in the API response.")}
  return(response_content$choices[[1]]$message$content)
}
```

The function returns the grouped answers as a text string, where each answer is assigned a group number based on its similarity to other answers.

4.2.4. Enhanced stage

The group_answers function is updated to include the knowledge base. The knowledge base provides additional background for AI to understand. In the previous case, AI did not know what the question was, and it was impossible to tell AI the specific grading criteria.

Below is an example of a knowledge_base (Figure 1), a text file that contains more detailed instructions for the AI model to understand.

```
You are an expert in evaluating and grouping student answers based on their similarity.
Consider common coding patterns, syntax, and logical structures in R when grouping similar answers.
Here are some examples of correct and executable R code snippets:

Example 1:
Calculate the mean of the mpg column in the mtcars dataset
mean_mpg <- mean(mtcars$mpg)

Example 2:
Calculate the sum of the mpg column in the mtcars dataset
sum_mpg <- sum(mtcars$mpg)

When grouping answers, consider the following criteria:
1. Similarity in function names and operations (e.g., mean, sum, filter, ggplot).
2. Similarity in the use of libraries and packages (e.g., dplyr, ggplot2, lubridate).
3. Similarity in data manipulation and transformation techniques (e.g., group_by, summarise).
4. Similarity in statistical analysis and modeling (e.g., lm, predict).
5. Similarity in data visualization patterns (e.g., geom_point, geom_smooth, geom_boxplot).
6. Evaluate if the code is executable or contains errors and group accordingly.

Provide a group number for each answer based on these criteria.
```

Figure 1. example of knowledge_base

Parameters:

• answers: A data frame containing student answers. Each answer should be in a separate row, with a unique identifier for each student.

• api_key: A string containing the OpenAI API key used for authentication.

• knowledge_base: A txt file containing the content of the knowledge base, which provides additional context for the AI model.

Logic and Algorithm:

1. Reading the Knowledge Base:

• The knowledge base reads information from these files. This file includes useful information such as grading standards, the context of the questions, and other relevant information that the AI

DOI: 10.54254/2755-2721/2025.30694

model can utilize to further comprehend the questions. Then, it can further organize the answers given by students.

2. Integrate Knowledge Base into the Prompt:

```
knowledge_base <- readLines("C:/codes/knowledge_base.txt")
knowledge_base <- paste(knowledge_base, collapse = "\n")
```

• It includes the system message knowledge base content as a part of the prompt. The thesis system message sets up the context for the AI model with respect to background information and guidelines, which should be considered while performing the task by the model in order to complete it.

• With the incorporation of the knowledge base, it becomes enriched with the criteria and standards against which the answers are to be evaluated and grouped.

3. Improved Prompt Construction:

• The construction of prompts involves adding together the contents of the knowledge base with the instructions for arranging the answers into groups. It consists of

• Design a system message that will provide space for the knowledge base.

• Design a user message to tell the model to put the answers in some format to output.

• This combined prompt ensures the AI model is privy to the specific context and standards to be applied while grouping the answers.

4. Algorithm to Include Knowledge Base:

• Step 1: Read from the knowledge base in the text file; lines are joined as one string.

• Step 2: Form a system message by placing the context to the model having knowledge base content.

• Step 3: Form the user message with specific instructions to do the task.

• Step 4: Merging of the system and user messages to a single prompt.

• Step 5: Sending the constructed prompt to the OpenAI API.

• Step 6: Treatment of the API response to process the grouped answers. Development Stage and Usage:

In developing this function, we integrated a knowledge base inside the group_answers function so that student answer processing can provide the AI model with more background knowledge. Enhancing this performing model of provisions groups of student answers depends on standard questions of gradings and question content. This stage lays the foundation on which subsequent refinements could be built, such as creating a multi-agent system to increase functionality.

4.2.5. Final stage

Multi-agent Intelligent agents are generally defined as entities that use one or more sensors to perceive their environments and use actuators or effectors to act upon that environment so that the given set of goals is achieved. An intelligent agent may be either simple or complex, autonomous or semi-autonomous, and may operate independently or as a part of a much larger system.

(1) Key Characteristics of an Agent

1. Autonomy: The agents operate independently without human active direction. They generate their own decisions without human interference, considering programming and situations.

2. Reactivity: These agents perceive the environment in real time and respond actively to changes within their environments.

3. Proactiveness: Agents can exhibit goal-directed behavior. They can take the initiative to perform actions independently in order to achieve the specified objective.

DOI: 10.54254/2755-2721/2025.30694

4. Social Ability: This is the ability to interact with other agents and humans, communicate information, collaborate, and coordinate actions with others.

(2) Agent System Overview In an LLM-powered autonomous agent system, LLM functions as the agent's brain, complemented by several key components:

• Planning

– Subgoal and decomposition: The agent breaks down large tasks into smaller, manageable subgoals, enabling efficient handling of complex tasks.

– Reflection and refinement: The agent can do self-criticism and self-reflection on past actions, learn from mistakes, and refine them for future steps, thereby improving the quality of final results.

• Memory

– Short-term memory: I would consider all the in-context learning as utilizing short-term memory of the model to learn.

– Long-term memory: This provides the agent with the capability to retain and recall (infinite) information over extended periods, often by leveraging an external vector store and fast retrieval.

• Tool use

– The agent learns to call external APIs for extra information that is missing from the model weights (often hard to change after pre-training), including current information, code execution capability, access to proprietary information sources, and more.

Algorithm Distillation is an algorithm that is encapsulated in a long history-conditioned policy. Considering that an agent interacts with the environment many times and in each episode the agent gets a little better, AD concatenates this learning history and feeds that into the model. Hence we should expect the next predicted action to lead to better performance than previous trials.

(3) Multi-Agent System Overview

A Multi-Agent System (MAS) contains a variety of interacting intelligent agents. Specifically, an LLM-powered autonomous agent system comprises a large language model with a number of additional key components as the brain of the agent. Each agent in the MAS takes up an assumed role or responsibility in the application, leading to problem-solving capabilities that no single agent could achieve. Each of these agents communicates and cooperates in achieving common goals.

At the same time, division of labor, parallelism, and specialization enable value delivery on complex tasks to be done more efficiently. For example, in our project, a group of agents assembled in a single-threaded core deals with a variety of concerns, like reading input from files, doing memory management, processing data, and making API requests. This way, the modular architecture not only improves the robustness and maintainability of the code throughput but is also scalable and makes it easier to add new functional layers. The FileReaderAgent class is responsible for all interactions associated with reading input files from the filesystem. It provides methods to read CSV files and text files, which contain student answers and other essential data for processing.

(4) FileReaderAgent

The FileReaderAgent class is responsible for reading input files from the filesystem. It provides methods for reading CSV files and text files, which contain student answers and other necessary data for processing.

DOI: 10.54254/2755-2721/2025.30694

```
FileReaderAgent <- R6::R6Class("FileReaderAgent",
  public = list(
    read_csv = function(file_path) { r
      ead_csv(file_path)
    },
    read_lines = function(file_path) {
      lines <- suppressWarnings(readLines(file_path))
      paste(lines, collapse = "\n")
    }
  )
)
```

(5) MemoryAgent

The class MemoryAgent was modeled and implemented to interact with memory. It stores and retrieves previous interactions or results so that the system can maintain its state and learn from previous interactions.

```
MemoryAgent <- R6::R6Class( "MemoryAgent",
public = list(
memory = list(),
memory_file = NULL,
initialize = function(memory_file) { self$memory_file <- memory_file
self$load_memory()
},
add_to_memory = function(key, value) { self$memory[[key]] <- value
self$save_memory()
},
get_from_memory = function(key) { return(self$memory[[key]])
},
save_memory = function() {
saveRDS(self$memory, self$memory_file)
},
load_memory = function() {
if(file.exists(self$memory_file)) {
self$memory <- readRDS(self$memory_file)
} else {
self$memory <- list() }
}
))
```

(6) LearningAgent

The LearningAgent class learns from past data and modifies the system's behavior. It communicates with the MemoryAgent to learn from prior interactions and alter the system's responses.

```
LearningAgent <- R6::R6Class("LearningAgent",
                             public = list(
                               memory_agent = NULL,
                               initialize = function(memory_agent) { s
                                 elf$memory_agent <- memory_agent
                               },
                               update_behavior = function(data) {
                                 self$memory_agent$add_to_memory("last_update",
                                                                 Sys.time())
                                cat("Behavior updated based on data:", data, "\n") }
                             )
)
```

(7) ApiRequestAgent

Its responsibility is to request the OpenAI GPT-4 to make requests from its API. It formulates the request to be made, passes the request to the API, and handles the response.

DOI: 10.54254/2755-2721/2025.30694

```
ApiRequestAgent <- R6::R6Class("ApiRequestAge
                               nt", public =
                               list(
                                 api_key = NULL,
                                 initialize = function
                                   (api_key) { self$api_key <-
                                   api_key
                                 },
                                 request_grouping = function
                                   (messages) { response <- POST(
                                     url = "https://api.openai
                                   .com/v1/chat/completions" add_headers
                                   (`Authorization` = paste("Bearer",
                                                            self$api_key
                                                            ),
                                                `Content-Type` =
                                     "application/json"), body = toJSON(list(
                                       model = "gpt-4",
                                       messages =
                                       messages,
                                       max_tokens =
                                       1500,
                                       temperature = 0.7

                                 response_content <- content(response, as = "parsed", type
                                                             = "application/json")
                                 if (length(response_content$choices) == 0) { st
                                 op("No choices found in the API response.")    }
                                 response_content$choices[[1]]$message$content
                               }
                             )
)
```

(8) ExtendedApiRequestAgent

The ExtendedApiRequestAgent class extends the class ApiRequestAgent. In other words, in this inheritance, we extend the class ApiRequestAgent by adding memory management and updating behavior. It saves results into memory and updates the system's behavior using the LearningAgent, which uses feedback.

```
ExtendedApiRequestAgent <- R6::R6Class("ExtendedApiRequestAgent",
                                       inherit = ApiRequestAgent,
                                       public = list(
                                         memory_agent = NULL,
                                         learning_agent = NULL,
                                         initialize = function(api_key, memory_agent,
                                                               learning_agent) {
                                           super$initialize(api_key)
                                           self$memory_agent <- memory_agent
                                           self$learning_agent <- learning_agent
                                         },
                                         request_grouping = function(messages) {
                                           result <- super$request_grouping(messages)
                                           self$memory_agent$add_to_memory("last_result",
                                                                           result)
                                           self$learning_agent$update_behavior(result)
                                           return(result)
                                         }
                                       )
)
```

(9) DataProcessorAgent

Responsible for the processing of the grouping result coming from an OpenAI API. Basically, it takes the grouping information from an API response and formats it for further usage.

```
DataProcessorAgent <- R6::R6Class("DataProcessorAgent",
                                  public = list(
                                    process_grouping_result = function(grouping_result)
                                      {
```

DOI: 10.54254/2755-2721/2025.30694

```
                grouping_lines <- strsplit(grouping_result, "\n")[[1]]
                grouping_data <- do.call(rbind, lapply(grouping_lines,
                                                       function(line)
                  if (grepl("Answer", line)) {
                  parts <- strsplit(line, ": Group ")[[1]]
  answer_num <- as.integer(sub("Answer ", "", parts[1]))
                    group_num <- as.integer(parts[2])
                    return(c(answer_num, group_num))
                  }
                }))

              as.data.frame(grouping_data, stringsAsFactors = FALSE)}
            )
)
```

(10) ResultWriterAgent

The ResultWriterAgent class is responsible for writing the processed results to a CSV file. It ensures that the final grouped results are stored in a format that can be easily accessed and reviewed.

```
ResultWriterAgent <- R6::R6Class("ResultWriterAgent",
                                 public = list(
                                   write_csv = function(data, file_path) { w
                                     rite_csv(data, file_path)
                                   }
                                 )
)
```

(11) FeedbackAgent

The FeedbackAgent class is responsible for processing user feedback to improve the model. It incorporates feedback to refine the grouping process and enhance accuracy.

```
FeedbackAgent <- R6::R6Class( "FeedbackAgent" , public = list(
memory_agent = NULL ,
initialize = function (memory_agent) { self$memory_agent <- memory_agent
},
process_feedback = function (feedback) { updated_data <- self$memory_agent
$get_from_memory ( "last_result") if (is.null (updated_data)) {
stop ("No previous result found in memory . ") }
                  for (fb in 1:nrow(feedback)) {
                    updated_data[updated_data$AnswerNumber ==
                                   feedback$AnswerNumber[fb],
                                 "Group"] <-
                      feedback$CorrectGroup[fb]
                  }
                  self$memory_agent$add_to_memory("updated_data",
                                                  updated_data)
                  return(updated_data)
                }
              )
)
```

5. Conclusion

5.1. Discussion and shortage

1. Limited User Base: It is not yet that effective a system because there is no large user base involved that allows diversified data to be collected along with user feedback on improvements. Since comprehensive real-world usage cannot be established, the system's accuracy thus becomes problematic, and improvements toward enhanced functioning cannot be accordingly marked. This, in effect, also limits its generalization capabilities across different educational contexts and can lead to insufficiencies or inaccuracies during the grading of diverse forms of submissions.

2. Insufficient Training Data: The model draws on a very sparse data set and supports poorly very diverse student submissions, especially with respect to really weird coding practices or grossly complicated ways of problem-solving. It needs a more comprehensive, heterogeneous dataset for the AI to classify and comment on the various assignments correctly and, hence, be more robust and reliable.

DOI: 10.54254/2755-2721/2025.30694

3. Non-Enterprise API Key: The capacity of the system, in terms of scalability and integration, is lacking because of the use of a non-enterprise API key, which may go to the extent of performance bottlenecks in high volumes of processing the data and also raising barriers to smoother integration with other education-related systems like LMS and SIS platforms. Upgrading to an enterprise level of API would take care of these deficiencies, with the system capable of handling more significant workloads and meeting the institutional needs better.

5.2. Future goals

1. Increasing User Base: In order to increase the reach of the system, pilot-run programs need to be initiated at different educational setups. Working in collaboration with educational platforms to integrate the system will help increase the rate of growth in the user base. These efforts will obtain valuable data and feedback, which is important in fine-tuning the ability of the system and widening its applicability.

The larger size of the dataset for training: The issue here is that the dataset for training needs more input. This could be done in collaboration with educational institutions to ensure that access to more anonymized student submissions is reached and implemented. The inclusion of synthetic data generation will even further increase the variety, hence enabling the AI to handle different types and contexts of submissions for enhanced accuracy and generalizability.

3. Upgrade to Enterprise API: An upgrade to enterprise-level APIs is required to improve scalability, response times, and integration with other enterprise systems. Such an upgrade would also increase security to support dealing with sensitive academic information, making the system more attractive to institutions with the highest data processing needs.

4. Continuous Improvement: Essentially, the system must follow an iterative development approach, with its versions changing periodically in response to user feedback and technological evolution. This community-led innovation approach will ensure that the system remains effective, relevant, and responsive to the dynamically changing roles of both the educator and the student.

Acknowledgment

ChatGPT 4o is utilized to assist in translating and redrafting the language of our report, as well as developing and debugging the code. Duole Hong, Zhiye Jiang, Jiaqing Shen, Ziyan Wei contributed equally to this work and should be considered co-first authors

References

[1] Agent Society: From Individuality to Sociability. Available at: https: //arxiv.org/pdf/2309.07864.
[2] ChatGPT Prompt Engineering for Developers . DeepLearning.AI . Available at: https: //www.deeplearning.ai/short-courses/chatgpt-prompt-engineering-for-developers/ .
[3] Weng, L . (2023) . Agent . Available at: https: //lilianweng.github.io/posts/2023-06-23-agent/ .
[4] OpenAI API Keys . Available at: https: //platform.openai.com/api-keys .
[5] Smith, J . , & Doe, A . (2023) . Advanced Topics in AI . Available at: https: //arxiv.org/html/2309.12924v2 .
[6] Fang, K . N . (n.d.) . Creating your own R package . Retrieved from http: //www.kuangnanfang.com/?id=16 .