LLM Application

2354093 李雪菲 2353567 董钰洁

Abstract

This project presents a local visual multi-model chat system developed using Python and Gradio. It supports three large language models. Users can engage in basic conversations, multi-turn dialogues, and store conversation history through a web-based interface.

Keywords: Large Language Models, DeepSeek-R1, Python, Gradio, Human-Computer Interaction

1 Design and Implementation of Model API Calls

In this project, we integrated several large language models (LLMs), including DeepSeek-R1, GLM-4 by Zhipu AI, and Qwen by Alibaba. To enable unified interaction with these models, we designed and implemented a modular API calling mechanism that adapts to the specific requirements of each model provider. The overall process of interacting with these APIs consists of four main steps: message formatting, authentication, request sending, and response parsing. While the underlying APIs differ in structure and authentication methods, the logical flow remains consistent.

For DeepSeek-R1, we used the official OpenAI-compatible SDK to send requests. This model is hosted on a private server provided by Tongji University, and the API key is supplied directly through the SDK client configuration. The message format follows the standard OpenAI schema, with a list of role-content pairs. The SDK abstracts much of the complexity, allowing for a streamlined and efficient integration.

In the case of GLM-4, we adopted a direct HTTP POST request approach to communicate with the API hosted by Zhipu AI's open platform. The request payload includes the model name, a messages list, and generation parameters such as temperature and top_p. Authentication is handled via a Bearer Token in the HTTP request header. The response is returned in JSON format, from which we extract the generated text using a fixed key path.

For Qwen, the calling method is slightly different due to the structure defined by Alibaba Cloud's DashScope platform. Requests must include both an input field containing the message history and a parameters field specifying generation settings. Similar to GLM-4, Qwen also requires a Bearer Token in the header for authentication. The response format is unique in that the output text is embedded in a nested output.text field, which we parse accordingly.

To unify these different calling mechanisms, we implemented a single call_llm() function in the backend. This function accepts the model name and message list as input and internally routes the call to the appropriate API handler. Each branch of the function takes care of request formatting, authentication, and error handling. This modular approach significantly enhances code reusability, system maintainability, and extensibility for future model additions.

Through this design, the system is capable of seamlessly interacting with multiple LLM providers, offering flexibility and scalability for intelligent dialogue generation. It lays a solid foundation for expanding to more models or switching service providers based on performance, cost, or feature requirements.

2 User Interface Design

The core objective of our UI design is to build an intuitive and accessible human-computer interaction interface that supports multi-round dialogues, model switching, and historical conversation management.

By leveraging the modular structure provided by Gradio's Blocks API, we partitioned the interface into functionally distinct regions, thereby enhancing maintainability and user experience. This design abstracts the underlying complexity of large language model (LLM) APIs, enabling end-users to engage in intelligent conversations without technical barriers.

2.1 Component-Based Design Strategy

```
gr.Markdown("# Intelligent Chatbot")
gr.Markdown("Supports DeepSeek-R1, GLM-4, and Qwen. Multi-turn dialogue and history loading are enabled.")
```

To improve user comprehension and usability, the interface begins with a Markdown description outlining system capabilities and available LLMs:

Following this, the layout is divided into three primary functional regions:

1. **Model and History Selection Panel**

```
with gr.Row():
    model_select = gr.Dropdown(["DeepSeek-R1", "GLM-4", "Qwen"], label="Choose Model", value="DeepSeek-R1")
    history_name = gr.Dropdown(label="Select Previous Session", choices=[])
```

This panel contains two dropdowns: one for selecting the backend model, and another for choosing a previously saved conversation session. This design facilitates quick model switching and seamless session resumption, thereby improving overall flexibility and user control.

2. **Core Interaction Zone**

```
chatbot = gr.Chatbot(type='messages')
with gr.Row():
    msg = gr.Textbox(label="Enter your question", scale=6)
    submit_btn = gr.Button("Send", scale=1)
```

The chatbot component visualizes the dialogue context in a message-based format, preserving conversational history. The combination of a textbox and a submit button provides two interaction modalities—keyboard (Enter key) and mouse click—offering a more accessible user experience.

3. **Session Control Area**

```
with gr.Row():
    save_name = gr.Textbox(label="Save current session as:")
    save_btn = gr.Button("Save Session")
    clear_btn = gr.Button("Clear Session")
```

This area allows users to save their current conversations by naming the session, or clear the existing session to begin anew. This functionality supports task resets and longitudinal dialogue workflows, which are essential in exploratory or iterative question-answering scenarios.

2.2 User Interaction Logic

The interaction design follows event-driven programming principles and integrates tightly with the backend LLM APIs. Each user action is mapped to a specific callback function, ensuring responsive and state-aware behavior.

2.2.1 Message Submission Mechanism

```
msg.submit(chat, inputs=[model_select, msg, chatbot], outputs=[chatbot, chatbot, msg])
submit_btn.click(chat, inputs=[model_select, msg, chatbot], outputs=[chatbot, chatbot, msg])
```

Both text input submission (msg.submit) and button click (submit_btn.click) invoke the same handler function chat(). This function collects user input, passes the entire dialogue history to the selected model API, and updates the chatbot output accordingly. This dual-trigger design.

2.2.2 Session Saving Logic

```
save_btn.click(save_history, inputs=[chatbot, save_name], outputs=[history_name, chatbot])
```

Users can name and save ongoing conversations, which are stored in a session dictionary and reflected in the session selection dropdown. This persistence mechanism enables session continuity and post-hoc analysis of interaction flows.

2.2.3 Dialogue Reset Mechanism

```
clear_btn.click(lambda: [], outputs=chatbot)
```

A single-click reset functionality is provided for users wishing to restart the dialogue. This lightweight design clears the existing content in the chatbot, facilitating fresh task engagement without navigation overhead.

2.2.4 Session Reload Mechanism

```
history_name.change(load_history, inputs=history_name, outputs=chatbot)
```

Upon selection of a saved session from the dropdown, the corresponding historical

conversation is loaded into the chatbot display. This feature improves reusability and supports long-term dialogue-based workflows.

3 Effects of Adjustable Parameters in LLM

Large language models (LLMs) generate responses through stochastic decoding mechanisms governed by sampling-based parameters. Understanding these parameters is crucial for controlling model creativity, coherence, and output diversity. This project employs the GLM-4 model with two key configurable parameters: temperature and top_p.

3.1 Temperature

The temperature parameter modulates the randomness in token sampling. Lower values (e.g., 0.1) make the model more deterministic by favoring high-probability tokens, which is useful for tasks requiring precision and consistency. Conversely, higher values (e.g., 0.8–1.0) introduce greater variability and creativity. In our system, we adopt temperature=0.95 to balance response diversity with semantic relevance—ideal for open-ended conversational scenarios.

3.2 Top-p (Nucleus Sampling)

The top_p parameter controls the cumulative probability mass from which the next token is sampled. When top_p=0.7, the model samples from the smallest set of words whose cumulative probability exceeds 70%. This allows adaptive filtering based on distribution entropy, promoting both fluency and originality. Unlike fixed-size top-k sampling, top-p sampling dynamically adjusts the candidate set, making it more robust to context variations.

Together, these parameters govern the generative behavior of the model. By fine-tuning them, developers can effectively navigate the trade-off between output fidelity and linguistic creativity, tailoring the chatbot for various application domains—from factual answering to storytelling and ideation.

4 Project testing

Through practical testing, we have found that different styles of prompts lead to significantly different results when handling various types of tasks.

In everyday scenarios, such as casual Q&A, recommendations, and chit-chat, concise and natural prompts are most effective. For example, simply entering “Recommend a few light-hearted movies” can yield fluent and contextually appropriate responses. In these cases, prompts do not require overly structured design; instead, the natural flow of language is more important.

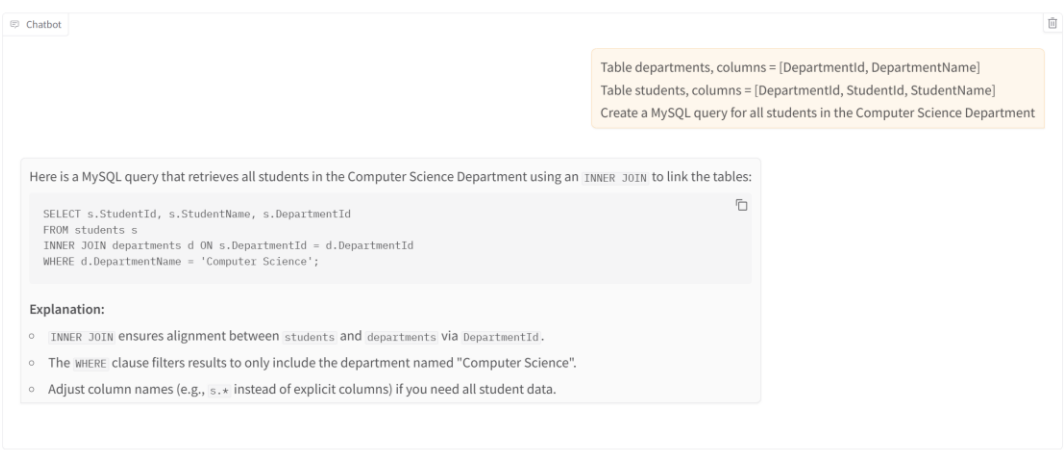
In professional tasks, such as writing academic content, generating code, or analyzing data, it is recommended to use prompts that are well-structured and contextually complete. Effective prompts typically include the following elements:

1) Instruction: A specific task or directive you want the model to perform.
2) Context: Additional background or external information that helps guide the model’s response.
3) Input data: The user’s question or content to be processed.
4) Output specification: The desired format or type of output.

In professional settings, the more specific and detailed the prompt, the better. Avoid vague descriptions and give the model clear and explicit instructions on what to do.

Example using DeepSeek-R1
Casual Conversation Scenario:
Prompt:"我不知道晚上吃什么，你有什么推荐吗"

The output in the casual conversation example is friendly, flexible, and easy to understand. It provides multiple dinner suggestions, organized into clear categories like healthy meals, warm dishes, and quick options. The tone is conversational and natural, making it feel like advice from a friend. This type of output demonstrates the model's ability to interpret open-ended prompts and generate responses based on common sense and daily life experience. It does not require strict formatting or technical precision, but instead focuses on relevance, fluency, and user engagement.
Professional Task Scenario:
Prompt:" Table departments, columns = [DepartmentId, DepartmentName]
Table students, columns = [DepartmentId, StudentId, StudentName]
Create a MySQL query for all students in the Computer Science Department"



The professional task output is highly structured, accurate, and technical. It generates a valid SQL query based on the given table schema and task description, using appropriate syntax such as INNER JOIN and WHERE clauses. In addition to the query itself, the model provides a clear explanation of each component, which enhances understanding and usability. This output reflects the model's capability to process structured prompts and follow precise instructions to complete well-defined tasks. Clarity, completeness, and correctness are key in this context.

5 Conclusion

This project successfully demonstrates the design and implementation of a local, visual multi-model chatbot system that integrates multiple large language models through a unified API interface. By leveraging Python and Gradio, we created an intuitive and modular user interface that supports real-time model switching, multi-turn dialogues, and session management. The system effectively abstracts the complexities of interacting with different LLM service providers, offering users a smooth and responsive human-computer interaction experience. Additionally, through adjustable parameters such as temperature and top_p, we explored the impact of generation settings on model outputs in both casual and professional scenarios. Overall, this project lays a scalable foundation for further development in intelligent dialogue systems and highlights the importance of prompt design in achieving high-quality, context-appropriate responses.