Multimodal table scanning application

1. Concept Introduction

1.1 What is "Multimodal Table Scanning"?

Multimodal table scanning is a technology that uses image processing and artificial intelligence to identify and extract table information from images or PDF documents. It not only focuses on visual table structure recognition but also incorporates multimodal data such as text content and layout information to enhance table understanding. **Large Language Models (LLMs)** provide powerful semantic analysis capabilities to understand this extracted information. The two complement each other and enhance the intelligence of document processing.

1.2 Implementation Principle Overview

1. **Table Detection and Content Recognition**

- Utilizes computer vision technology to locate tables in documents and uses optical character recognition (OCR) technology to convert the text within the tables into an editable format.
- Utilizes deep learning methods to analyze table structure (row and column division, cell merging, etc.) and generate a structured data representation.

2. **Multimodal Fusion**

- Integrates visual information (such as table layout), text (OCR results), and any metadata (such as file type and source) to form a comprehensive data view. - Use specially designed multimodal models (e.g., LayoutLM) to process these different types of data simultaneously to more accurately understand the table content and its contextual relationships.

2. Code Analysis

Key Code

1. Tool Layer Entry (`largemodel/utils/tools_manager.py`)

The `scan_table` function in this file defines the tool's execution flow, specifically how it constructs a prompt that returns a Markdown-formatted result.

```
# From largemodel/utils/tools_manager.py
class ToolsManager:
    # ...
    def scan_table(self, args):
        """
        Scan a table from an image and save the content as a Markdown file.
        从图像中扫描表格，并将内容保存为Markdown文件。

        :param args: Arguments containing the image path.
        :return: Dictionary with file path and content.
        """
        self.node.get_logger().info(f"Executing scan_table() tool with args:
{args}")
        try:
            image_path = args.get("image_path")
            # ... (Path checking and fallback)

            # Construct a prompt asking the large model to recognize the table
and return it in Markdown format.
            # Constructs a prompt that requires the large model to recognize the
table and return it in Markdown format.
            if self.node.language == 'zh':
                prompt = "请仔细分析这张图片，识别其中的表格，并将其内容以Markdown格式返
回。"
            else:
                prompt = "Please carefully analyze this image, identify the
table within it, and return its content in Markdown format."

            result = self.node.model_client.infer_with_image(image_path, prompt)

            # ... (Extract Markdown text from the results)

            # Save the recognized content to a Markdown file. / 将识别出的内容保存到
Markdown文件。
            md_file_path = os.path.join(self.node.pkg_path, "resources_file",
"scanned_tables", f"table_{timestamp}.md")
            with open(md_file_path, 'w', encoding='utf-8') as f:
                f.write(table_content)

            return {
                "file_path": md_file_path,
                "table_content": table_content
            }
        # ... (Error Handling)
```

2.Model Interface Layer (`largemodel/utils/large_model_interface.py`)

The `infer_with_image` function in this file serves as the unified entry point for all image-related tasks.

```
# From largemodel/utils/large_model_interface.py

class model_interface:
    # ...
    def infer_with_image(self, image_path, text=None, message=None):
        """Unified image inference interface. / 统一的图像推理接口。"""
        # ... (Prepare Message)
        try:
            # Determine which specific implementation to call based on the value
of self.llm_platform
            if self.llm_platform == 'ollama':
                response_content = self.ollama_infer(self.messages,
image_path=image_path)
            elif self.llm_platform == 'tongyi':
                # ... Logic for calling the Tongyi model
                pass
            # ... (Logic of other platforms)
        # ...
        return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

The table scanning function is a typical application for converting unstructured image data into structured text data. Its core technology remains **guiding model behavior through prompt engineering**.

1. **Tools Layer (`tools_manager.py`)**:

- The `scan_table` function is the business process controller for this function. It receives an image containing a table as input.
- The key operation of this function is **building a targeted prompt**. This prompt directly instructs the large model to perform two tasks: 1. Recognize the table in the image. 2. Return the recognized content in Markdown format. This mandatory output format is key to achieving unstructured-to-structured conversion.
- After constructing the prompt, it calls the `infer_with_image` method of the model interface layer, passing the image and the formatting instructions.
- After receiving the Markdown text returned from the model interface layer, it performs a file operation: writing the text content to a new `.md` file.
- Finally, it returns structured data containing the new file path and table contents.

2. **Model Interface Layer (`large_model_interface.py`)**:

- The `infer_with_image` function continues to serve as the unified "dispatching center." It receives the image and prompt from `scan_table` and dispatches the task to the correct backend model implementation based on the current system configuration (`self.llm_platform`).
- Regardless of the backend model, this layer handles the communication details with the specific platform, ensuring that the image and text data are sent correctly, and then returns

the plain text (in this case, Markdown-formatted text) returned by the model to the tooling layer.

In summary, the general workflow for table scanning is: `ToolsManager` receives an image and constructs a command to "convert the table in this image to Markdown" -> `ToolsManager` calls the model interface -> `model_interface` packages the image and the command and sends it to the corresponding model platform according to the configuration -> The model returns Markdown-formatted text -> `model_interface` returns the text to `ToolsManager` -> `ToolsManager` saves the text as a `.md` file and returns the result. This workflow demonstrates how to leverage the formatting capabilities of a large model as a powerful OCR (optical character recognition) and data structuring tool.

3. Practical Operation

3.1 Configuring Online LLM

1. **First obtain the API Key from any platform in the previous tutorial**
2. **Then you need to update the key in the configuration file and open the model interface configuration file** `large_model_interface.yaml`:

```
vim ~/yahboom_ws/src/largemodel/config/large_model_interface.yaml
```

3. **Fill in your API Key**:
   Find the corresponding part and paste the API Key you just copied into it. Here we take Tongyi Qianwen configuration as an example

```
# large_model_interface.yaml

## Thousand Questions on Tongyi
qianwen_api_key: "sk-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" # Paste your Key
qianwen_model: "qwen-vl-max-latest" # You can choose the model as needed,
such as qwen-turbo, qwen-plus
```

4. **Open the main configuration file** `yahboom.yaml`:

```
vim ~/yahboom_ws/src/largemodel/config/yahboom.yaml
```

5. **Select the online platform you want to use**:
   Modify the `llm_platform` parameter to the platform name you want to use

```
# yahboom.yaml

model_service:
  ros__parameters:
    # ...
    llm_platform: 'tongyi'  #Optional Platform: 'ollama', 'tongyi', 'spark',
'qianfan', 'openrouter'
```

3.2 Start and test the functionality

1. **start up**

   Note: The startup commands for CSi cameras and USB microphone cameras are different. Please run the appropriate command for your camera.

   CSI Camera

   Starting UDP Video Streaming (Host)

   ```
   ./start_csi.sh
   ```

   Enter the CSI camera docker (host machine)

   ```
   ./run_csi_docker.sh
   ```

   Start topic conversion (docker)

   ```
   python3 ~/temp/udp_camera_publisher.py
   ```

   View container id

   ```
   docker ps
   ```

   According to the container ID shown above, multiple terminals enter the same docker

   ```
   docker exec -it  container_id  /bin/bash
   ```

   Run the following command to enable voice interaction:

   ```
   ros2 launch largemodel largemodel_control.launch.py
   ```

   USB Camera

   Enter the USB camera docker (host machine)

   ```
   ./run_usb_docker.sh
   ```

   Run the following command to enable voice interaction:

   ```
   ros2 launch largemodel largemodel_control.launch.py
   ```

2. **Test**:

   - **Wake up**: Say "Hi,yahboom" into the microphone.
   - **Dialogue**: After the speaker responds, you can say: `Analyze the table`
   - **Observe the log**: In the terminal running the `launch` file, you should see the following:

   1. The ASR node recognizes your question and prints it.
   2. The `model_service` node receives the text, calls the LLM, and prints the LLM's response.

- **Listen for the answer**: After a while, you should hear the answer from the speaker and find an md file containing the table information in the `/root/yahboom_ws/src/largemodel/resources_file/scan_table` path.

3. **FAQ**:

   Modify to your own picture

   (1) Rename the image to test_table.jpg and place it in the ~/temp directory for later use.

   (2) Enter any Docker terminal

   ```
   cd ~/temp
   ```

   ```
   cp test_table.jpg ~/yahboom_ws/src/largemodel/resources_file/scan_table
   ```

   Copy the image to the ~/yahboom_ws/src/largemodel/resources_file/scan_table directory

   (3) Restart the `largemodel` main program