Multimodal Table Scanning Applications

---

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 combines multimodal data such as text content and layout information to enhance table understanding. **Large Language Models (LLMs)** provide powerful semantic analysis capabilities for understanding this extracted information. The two complement each other, jointly improving the intelligence level of document processing.

1.2 Brief Description of Implementation Principles

1. **Table Detection and Content Recognition**

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

2. **Multimodal Fusion**

- Integrates visual elements (such as table layout), text (OCR results), and any existing metadata (such as file type and source) to form a comprehensive data view.
- Uses a specially designed multimodal model (such as LayoutLM) to process these different types of data simultaneously for a more accurate understanding of table content and its context.

2. Code Analysis

Key Code

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

The `scan_table` function in this file defines the execution flow of the tool, specifically how it constructs a Prompt that requests a Markdown format return.

```
# 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.


        :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 check and rollback)

            # Construct a prompt asking the large model to recognize the table
and return it in Markdown format.
            # The prompt suggests that the large model should recognize tables
and return them 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.
            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 is 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. """
        # ... (Preparing the message)
        try:
            # The value of self.llm_platform determines which specific
implementation to call.
            if self.llm_platform == 'ollama':
                response_content = self.ollama_infer(self.messages,
image_path=image_path)
            elif self.llm_platform == 'tongyi':
                # ... Logic of calling the generalized model
                pass
            # ... (Logic for other platforms)
        # ...
        return {'response': response_content, 'messages': self.messages.copy()}
```

Code Analysis

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

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

- The `scan_table` function is the business process controller for this function. It takes an image containing a table as input.
- The most crucial operation of this function is **constructing 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 requirement for the output format is key to achieving the unstructured to structured conversion.
- After constructing the Prompt, it calls the `infer_with_image` method of the model interface layer, passing the image and this formatting instruction along with it.
- After receiving the returned Markdown text from the model interface layer, it performs a file operation: writing the text content into a new `.md` file.

Finally, it returns structured data containing the new file paths and table contents.

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

- The `infer_with_image` function continues to act as a unified "dispatch 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's task is to handle the communication details with the specific platform, ensuring that image and text data are sent correctly, and then return the plain text (in this case, Markdown format) returned by the model to the utility layer.

In summary, the general workflow for table scanning is as follows: `ToolsManager` receives the 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 appropriate 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 large models as a powerful OCR (Optical Character Recognition) and data structuring tool.

3. Practical Operation

3.1 Configuring Offline Large Models

3.1.1 Configuring the LLM Platform (`yahboom.yaml`)

This file determines which large model platform the `model_service` node loads as its primary language model.

1. **Open the file in the terminal**:

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

2. **Modify/confirm** `llm_platform`:

```
model_service:                          #Model server node parameters
  ros__parameters:
    language: 'en'                      #Large Model Interface Language
    useolinetts: True                   #This option is invalid in text mode and
can be ignored.

# Large Model Configuration
llm_platform: 'ollama'              # Key point: Make sure this is 'ollama'
```

3.1.2 Configure the model interface (`large_model_interface.yaml`)

This file defines which visual model is used when the platform is selected as `ollama`.

1. Open the file in the terminal

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

2. Locate the ollama-related configuration

```
#.....
## Offline Large Language Models
# Ollama Configuration
ollama_host: "http://localhost:11434" # Ollama server address
ollama_model: "llava" # Key: Replace this with your downloaded multimodal model,
such as "llava"
#.....
```

Note: Ensure that the model specified in the configuration parameters (such as `llava`) can handle multimodal input.

3.1.3 Recompile

```
colcon build
source install/setup.bash
cd ~/yahboom_ws/
```

3.2 Launching and Testing the Feature (Text Mode)

1. **Preparing the Table Image File**:

Place an image file of the table you want to test in the following path:
`/home/sunrise/yahboom_ws/src/largemodel/resources_file/scan_table`

Then name the image `test_table.jpg`

2. **Launch the `largemodel` main program (Text Mode)**:

Open a terminal and run the following command:

```
ros2 launch largemodel largemodel_control.launch.py text_chat_mode:=true
```

3. **Sending Text Commands**:

Open another terminal and run the following command:

```
ros2 run text_chat text_chat
```

Then start typing the text: "Analyze the table".

4. **Observation Results**:

In the first terminal running the main program, you will see log output showing that the system received the command, called the `scan_table` tool, and indicated that `scan_table` had completed execution and saved the scanned information to a document.

This document can be found at ~/yahboom_ws/src/largemodel/resources_file/scan_table.

4. Common Problems and Solutions

Problem 1: Incomplete table content recognition or typos.

Solution:

1. **Image Quality**: Ensure the input table image is clear, undistorted, and evenly lit. Low-quality images are the most common cause of recognition errors.
2. **Model Selection**: Models with more parameters generally perform better; try using a more powerful recognition model.

Problem 2: Error message "Table file not found"

Solution:

1. **Check Path**: Ensure you have placed the table image file in the specified path, named `test_table.jpg`, and have permission to read the file.
2. **Image Format**: Ensure the image file is not corrupted and is in .jpg format.