Project Report: Utilizing LangChain to Build a RAG System for XGBoost Documentation

1. Introduction

In this project, we explore how to build a Retrieval-Augmented Generation (RAG) system using LangChain to provide detailed and contextually relevant answers from XGBoost's documentation. This report details the steps, code, and reasoning behind utilizing LangChain for various tasks within the project.

2. Background

Retrieval-Augmented Generation (RAG) systems enhance the capabilities of language models by combining document retrieval with text generation. This approach is particularly useful for answering specific queries where the answer must be grounded in an existing set of documents, such as technical documentation.

3. Document Retrieval from GitHub

The GitHub loader is an essential tool for extracting information from a GitHub repository, enabling users to access issues, pull requests (PRs), and files from a specific repository. This functionality is particularly useful for analyzing code changes, tracking project progress, and retrieving documentation directly from the repository. By leveraging the GitHub loader, developers can programmatically access and process repository data, enhancing automation and integration capabilities within their workflows. In our example, we use the LangChain Python repository to demonstrate how to load issues, PRs, and files, providing a comprehensive view of the project's development history and current state.

To access the GitHub API, a personal access token is required. This token can be generated from GitHub's settings page and set as an environment variable, `GITHUB_PERSONAL_ACCESS_TOKEN`, or passed directly during the loader's initialization. The `GithubFileLoader` from LangChain is configured with the repository name, access token, and GitHub API URL. For instance, to load files from the `dlmc/xgboost` repository, the loader

is initialized with the repository name and access token. This setup allows seamless extraction of files and data from the repository, facilitating tasks such as documentation retrieval, code analysis, and data processing directly from the GitHub source.

4.1 Fetching Files

Using the Langchain community document loader, we retrieve the XGBoost documentation files:

```python
from langchain.document_loaders import GithubFileLoader

##API Reference:GithubFileLoader

loader = GithubFileLoader(
        repo="dlmc/xgboost", # the repo name
        access_token=ACCESS_TOKEN,
        github_api_url="https://api.github.com",
),
```

4.2 Processing Content

In many repositories, files are organized within directories, and these directories may further contain subdirectories. To efficiently process and analyze the entire repository, it is essential to flatten this hierarchical structure into a single, unified list of files. This flattening process helps streamline the retrieval of relevant documentation and discards any non-essential files, ensuring that only pertinent information is processed.

To achieve this, we first create a function, `get_content`, that retrieves the contents of the files from the provided URLs using the GitHub API. This function is applied to the initial list of files in the repository to generate a comprehensive menu of file contents. Next, we define a recursive function, `flat_content`, which iterates through the menu. If an item in the menu is a list (indicating a subdirectory), the function calls `get_content` again to retrieve its contents and recursively flattens them. Otherwise, it appends the file's path to the final list, `final_menu`. This approach ensures that the directory structure is effectively flattened, and all files are consolidated into a single list for easier access and processing.

```python
def get_content(files):
```

```
    return [requests.get(x['url'], headers=headers).json() for x in files]

menu = get_content(files)

def flat_content(menu=menu, debug=False):
    for item in menu:
        if isinstance(item, list):
            menu = get_content(item)
            flat_content(menu)
        else:
            if debug:
                print(item['path'])
            final_menu.append(item)

flat_content(menu=menu, debug=True)
```

By flattening the directory structure, we simplify the process of filtering out non-relevant files and focusing on the documentation. This step is crucial in ensuring that our RAG system only processes and retrieves relevant information, thereby enhancing its accuracy and efficiency.

4. Document Parsing and Chunking

Documents are parsed and chunked to prepare them for processing by Custom pipelines. Langchains pipeline, lead to ineffective chunks, and semantically insignificant for the specific use case

5.1 Parsing Content

Originally RecursiveCharacterTextSplitter was used, but it was ineffective, as various sections in the documentation had different sizing, and each section was important to be chunked together. Thus, we use regular expressions to parse headings and chunk the content appropriately.

Predefined Chunking :

```python
```

```
text_splitter = RecursiveCharacterTextSplitter(
  # Set a really small chunk size, just to show.
  chunk_size=100,
  chunk_overlap=20,
  length_function=len,
  is_separator_regex=False,
)
texts = text_splitter.create_documents([state_of_the_union])
print(texts[0])
print(texts[1])
```

Custom Chunking :

```python
import re

def parse_content(text, debug=False):
    heading_patterns = [
        r"(?P<heading0>^[A-Za-z0-9 \\-]+)\n(?P<underline0>\#+)$",
        r"(?P<heading1>^[A-Za-z0-9 \\-]+)\n(?P<underline1>=+)$",
        r"(?P<heading2A>^[A-Za-z0-9 \\-]+)\n(?P<underline2>-+)$",
        r"(?P<heading2B>^[A-Za-z0-9 \\-]+)\n(?P<underline3>\*+)$",
        r"(?P<heading3>\* ``[A-Za-z0-9_\\-]+``)"
    ]

    combined_pattern = "|".join(heading_patterns)
    matches = re.finditer(combined_pattern, text, re.MULTILINE)
    headings = []
    last_index = 0

    for match in matches:
        if debug:
            print(match)

        start, end = match.span()

        if last_index < start:
            content = text[last_index:start].strip()
            if all([content, headings]):
                headings[-1]['content'].append(content)

        if match.group('heading0'):
```

```
            headings.append({'heading': match.group('heading0').strip(), 'level': 0, 'content': []})
        elif match.group('heading1'):
            headings.append({'heading': match.group('heading1').strip(), 'level': 1, 'content': []})
        elif match.group('heading2A'):
            headings.append({'heading': match.group('heading2A').strip(), 'level': 2, 'content': []})
        elif match.group('heading2B'):
            headings.append({'heading': match.group('heading2B').strip(), 'level': 2, 'content': []})
        elif match.group('heading3'):
            headings.append({'heading': match.group('heading3').strip(), 'level': 3, 'content': []})

        last_index = end

    if last_index < len(text):
        content = text[last_index:].strip()
        if all([content, headings]):
            headings[-1]['content'].append(content)
```

6. Integrating LangChain and HuggingFace

LangChain facilitates the integration with HuggingFace models, enabling robust language processing capabilities.

6.1 HuggingFace Integration

We set up a HuggingFace endpoint for text generation tasks.

```python
from langchain_huggingface import HuggingFaceEndpoint

repo_id = "mistralai/Mistral-7B-Instruct-v0.2"
llm = HuggingFaceEndpoint(
    repo_id=repo_id,
    task="text-generation",
    max_new_tokens=512,
    do_sample=False,
    repetition_penalty=1.03,
    temperature=0.8
)
```

6.2 Prompt Templates

We use LangChain's `PromptTemplate` to define structured prompts for the language model.

```python
from langchain_core.prompts import PromptTemplate

template = """
###
You love to explain technical concepts, given you have appropriate context.
Use the following pieces of context to answer the question given to you.
If the answer is not there in the context, just say that you don't know the answer, don't try to make up an answer.
Your answer must write the references to the source of context otherwise it will be considered as plagiarism. The reference link will be provided in the metadata.

### Context:
{context}

### Question:
{question}

### Description of a good answer:
* A good answer will provide reference like provided in academic papers and websites
* A good answer will not contain information which is not provided in the context
* An excellent answer will provide examples with codes to answer the question if it can be provided
"""

gen_prompt = PromptTemplate(
    input_variables=["context", "question"],
    template=template,
)
```

7. Building the RAG Chain

Combining the components into a functional RAG chain allows for efficient retrieval and generation of answers.

```python
from langchain_core.runnables import RunnableParallel, RunnablePassthrough

chain = gen_prompt | llm

RAG_chain = (
   {
      "context": retriever_from_llm | get_docs,
      "question": RunnablePassthrough()
   }
   | chain
)

print(RAG_chain.invoke('what is boosters parameter in xgboost?'))
print(RAG_chain.invoke('How to train XGBOOST for a dataset with 32 million rows?'))
```

8. Conclusion

This project demonstrates the power of LangChain in building a RAG system tailored for XGBoost's documentation. By leveraging LangChain's capabilities for document handling, prompt management, and language model integration, we create a system that can efficiently retrieve and generate contextually accurate answers.

LangChain's seamless integration with HuggingFace models further enhances the system's ability to process and respond to complex queries, making it an invaluable tool for technical support, academic research, and more. As language models continue to evolve, the potential applications of RAG systems will expand, offering even more sophisticated solutions to a wide range of challenges.