**Assignment Report: LangChain & NLP Task**

**Objective:**

The primary goal of this project was to build a question-answering (QA) system using LangChain and an open-source large language model (LLM) such as GPT-2 or GPT-J. The system takes user input and responds with answers based on the provided context.

This report outlines the steps taken to:

* Set up the environment.
* Integrate an open-source LLM with LangChain to create a functional QA system.
* Design a custom prompt template to improve the quality of the responses.
* Implement memory or tool integration to enhance system features.

**Task 1: Setting up the Environment**

**Python Environment Setup:**

To begin, a virtual Python environment was created to manage dependencies in isolation. The following commands were used:

py -m venv langchain\_env

source langchain\_env/bin/activate

**Required Libraries:**

The following Python packages were installed to support the development of the QA system:

* **LangChain**: Enables chaining language models together and handling complex language tasks.
* **HuggingFace Transformers**: Facilitates access to open-source LLMs like GPT-2 and GPT-J.
* **Torch**: A deep learning framework required to run these models.
* **Sentence-Transformers**: Used for document embedding, enabling relevant context retrieval.

pip install langchain

pip install transformers

pip install torch

pip install sentence-transformers

pip install openai

This environment provides the necessary tools to load, run, and integrate the LLM with LangChain to handle QA tasks.

**Task 2: LLM Integration with LangChain**

**Loading the LLM:**

For this project, GPT-2 was chosen as the LLM. Using Hugging Face's transformers library, the model and tokenizer were loaded:

from transformers import pipeline

# Load GPT-2 model

qa\_pipeline

qa\_pipeline = pipeline('text-generation', model="gpt2")

This pipeline will serve as the core for generating answers to questions provided by users.

**Integrating LangChain for Question Answering:**

LangChain was used to create a retrieval-based QA system. The system retrieves relevant information from a given context using semantic embeddings and passes the most relevant sections to the LLM for answering.

**Steps:**

1. **Vector Store Creation:** We used the **FAISS** vector store and Hugging Face’s **Sentence-Transformers** for embedding the context into a vector space. This helps retrieve relevant sections from the document based on the user's question.

from langchain.vectorstores import FAISS

from langchain.embeddings import HuggingFaceEmbeddings

# Load embedding model for document retrieval

embedding\_model = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")

# Create FAISS vector store

doc\_store = FAISS(embedding\_model.embed\_documents, index=None)

1. **Question Answering Chain:** Using LangChain's RetrievalQA, the system retrieves relevant portions of the context and passes it to the LLM. This allows the model to answer questions based on the most pertinent sections of the document.

from langchain.chains import RetrievalQA

from transformers import GPT2Tokenizer, GPT2LMHeadModel, TextGenerationPipeline

# Load GPT-2 model and tokenizer

model = GPT2LMHeadModel.from\_pretrained("gpt2")

tokenizer = GPT2Tokenizer.from\_pretrained("gpt2")

# Create a text generation pipeline for LangChain integration

hf\_pipeline = TextGenerationPipeline(model=model, tokenizer=tokenizer)

# Wrap the pipeline with LangChain

llm = HuggingFacePipeline(pipeline=hf\_pipeline)

# Setup the QA chain

qa\_chain = RetrievalQA(llm=llm, retriever=doc\_store.as\_retriever())

1. **User Interaction:** The system accepts user input in the form of a question and retrieves the relevant context to generate an answer. The context must be loaded into the document retriever before answering the user’s query.

def run\_qa\_system(context, question):

# Add the context to the document retriever

doc\_store.add\_texts([context])

# Get the answer

answer = qa\_chain.run(input=question)

return answer

if \_\_name\_\_ == "\_\_main\_\_":

context = "Provide a paragraph or short document as context."

user\_question = input("Please enter your question: ")

result = run\_qa\_system(context, user\_question)

print(f"Answer: {result}")

**Task 3: Working with LangChain Prompt Templates**

**Custom Prompt Template:**

A custom prompt template was created using LangChain’s PromptTemplate feature. This allows us to control the format and structure of the input that is provided to the LLM, improving the quality and relevance of the output.

**Prompt Template Code:**

from langchain.prompts import PromptTemplate

template = """

You are an intelligent assistant. Use the following context to answer the question in a concise and accurate manner.

Context: {context}

Question: {question}

Answer:

"""

# Define the prompt template

prompt = PromptTemplate(

input\_variables=["context", "question"],

template=template

)

**Using the Prompt Template in the QA System:**

This template structures the input provided to the LLM, ensuring it clearly separates the **context**, **question**, and **answer** sections. By specifying the format, the model generates more accurate and concise answers.

def run\_qa\_with\_custom\_prompt(context, question):

# Format the prompt using LangChain's PromptTemplate

formatted\_prompt = prompt.format(context=context, question=question)

# Run the QA chain with the formatted prompt

answer = qa\_chain.run(input=formatted\_prompt)

return answer

**Benefits of the Prompt Template:**

* **Improved Structure:** Separating the context, question, and answer ensures clarity, making it easier for the LLM to focus on the task.
* **Instruction Control:** By specifying the model should be concise and accurate, we guide the model to provide relevant and compact responses.
* **Consistency:** Using a standard template ensures that the system maintains consistency in its responses, making the output predictable and easier to refine.

**Task 4: Feature Implementation – Memory Integration**

**Memory Integration:**

To improve the QA system’s interactivity, LangChain’s **ConversationBufferMemory** was integrated. This feature enables the system to remember past interactions with the user, allowing it to respond to follow-up questions in the context of the entire conversation.

**Memory Integration Code:**

from langchain.memory import ConversationBufferMemory

from langchain.chains import ConversationalRetrievalChain

# Create a memory object to store conversation history

memory = ConversationBufferMemory(memory\_key="chat\_history")

# Build the conversational QA chain with memory

conversational\_chain = ConversationalRetrievalChain(

llm=llm,

retriever=doc\_store.as\_retriever(),

memory=memory

)

def run\_qa\_with\_memory(context, question):

# Add the context to the document retriever

doc\_store.add\_texts([context])

# Run the conversational chain with memory

result = conversational\_chain.run(input=question)

return result

**How Memory Enhances the System:**

* **Context Retention:** Memory allows the system to track prior interactions, making it capable of answering follow-up questions or clarifications without losing the context.
* **Improved User Experience:** Users can have more natural conversations, asking for further explanations or deeper details based on previous answers.

**Conclusion:**

This project successfully implemented a question-answering system using LangChain and an open-source LLM (GPT-2). The system can take a user’s question, retrieve relevant context, and provide a concise answer. Key improvements were achieved through the use of a custom prompt template and memory integration to retain conversation history. The report details:

1. **Environment Setup**: Documenting the installation of necessary libraries and tools.
2. **LLM Integration**: Utilizing Hugging Face's GPT-2 and LangChain to create a QA system.
3. **Custom Prompt Design**: Improving the LLM’s performance by structuring input through prompt templates.
4. **Memory Integration**: Enhancing the system’s interactivity by allowing follow-up questions and ongoing dialogue.

**Future Enhancements:**

In the future, the system could be further enhanced by integrating external tools, such as search engines or APIs, to handle more complex queries when the provided context is insufficient. Additionally, fine-tuning the LLM for specific domains could improve answer accuracy and relevance in specialized fields.