Using Retrieval Augmented Generation (RAG) to Enhance Local Large Language Models

Raj Uppadhyay · Follow

7 min read · Feb 13, 2024

In this post, we will delve deeper into the exploration of local large language models (LLMs) by running an LLM on a local machine. We will also examine the relevancy and correctness of the outcomes.

Prerequisites:

1. A basic understanding of the concepts discussed in the previous post, "Exploring Local Large Language Models and Associated Key Challenges" along with basic experience around using VS Code.
2. Familiarity with Python code and the LangChain framework (https://python.langchain.com/docs/get_started/introduction) for

building applications backed by language models.

Now, without further delay, let's embark on this exciting journey. First start with understanding the high level of RAG architecture and its components.

Introduction:
Retrieval Augmented Generation (RAG) is a technique that enhances the knowledge of Large Language Models (LLMs) by incorporating additional data. While LLMs can reason about diverse topics, their knowledge is limited to the public data available up until their training cutoff date. To create AI applications that can process private data or information introduced after the model's training period, augmenting the model's knowledge with relevant information is necessary. This process of incorporating appropriate information into the model prompt is known as RAG.

RAG System Pipeline:
RAG systems employ a two-step pipeline to generate responses: retrieval and generation.

1. Retrieval Phase:

In the retrieval phase, the model searches through databases or document collections to identify the most relevant facts and passages for the given prompt or user question. For open domains such as general web searches, indexed webpages can be leveraged. In closed domains like customer support, the retrieval process may involve controlled sets of manuals and articles.

2. Generation Phase:

The retrieved snippets of external knowledge are then appended to the original user input, augmenting the context. In the generation phase, the language model analyzes this expanded prompt to produce a response. It references both the retrieved information and its internally trained patterns to formulate an informative and natural answer.

Key Components of the RAG Framework:

Implementing an effective RAG system requires several key components:

1. **Language Model:**
   The foundation of RAG architecture is a pre-trained language model responsible for text generation. Models like GPT-3, Llama 2, Google BERT exhibit strong language comprehension and synthesis capabilities, enabling them to engage in conversational dialogues.

2. **Vector Store (Database):**
   Central to the retrieval functionality is a vector store database that stores document embeddings for efficient similarity searches. This allows for rapid identification of relevant contextual information.

3. **Retriever:**
   The retriever module utilises the vector store to locate pertinent documents and passages that augment the prompts. Neural retrieval approaches excel at semantic matching.

4. **Embedder (Data):**
   To populate the vector store, an embedder encodes source documents into vector representations that the retriever can consume. Models like BERT are effective for this text-to-vector abstraction.

5. **Indexer/Loader (Data):**
   Robust pipelines ingest and preprocess source documents, breaking them into manageable passages for embedding and efficient lookup.

By harmonizing these core components, RAG systems empower language models to access vast knowledge resources, enabling grounded generation and improved responses.

Hope by now we have acquired a satisfactory understanding of RAG, now let's see how it works practically.

Setting up development environment

- Open VS Code and open a new folder e.g. named llm_rag
- Open the terminal in VS Code and run following commands

```
pip install langchain
pip install langchain-community
pip install langchain-core
pip install langchain-cli
```

Getting our local model path

Following the previous post where we have installed LM Studio and downloaded the "llama-2–7b-chat.Q5_K_S.gguf" model locally.

Usually this model will get stored in the ".cache/lm-studio/models/TheBloke/Llama-2–7B-Chat-GGUF/llama-2–7b-chat.Q5_K_S.gguf" in the user profile. You can easily it:

- **Windows:** %profile%/.cache/.cache/lm-studio/models/TheBloke/Llama-2–7B-Chat-GGUF/llama-2–7b-chat.Q5_K_S.gguf
- **Other OS:** /Users/<yourusername>/.cache/lm-

studio/models/TheBloke/Llama-2–7B-Chat-GGUF/llama-2–7b-chat.Q5_K_S.gguf

Create sample code to query our Local LLM
In this section we will be writing a sample code which will query our Local LLM for finding "Top 5 companies in the world with their revenue in table format?" and then we will validate the results with a source website e.g. https://www.investopedia.com/biggest-companies-in-the-world-by-market-cap-5212784

- In VS Code, create a file named local_llm.py and add the following code. Please do read the comments for understanding the purpose of each code block.

```
#importing the main libraries for setting up code to interact with LLM
from langchain.callbacks.manager import CallbackManager
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
from langchain.prompts import PromptTemplate
from langchain_community.llms import LlamaCpp

# Defining a Promt Template to interact with LLM
template = “””Question: {question}
Answer: Let’s work this out in a step by step way to be sure we have the right

# Callbacks support token-wise streaming
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])
n_gpu_layers = 1 # Change this value based on your model and your GPU VRAM pool
n_batch = 512 # Should be between 1 and n_ctx, consider the amount of VRAM in y

# Make sure the model path is correct for your system!
llm = LlamaCpp(
```

```
model_path=”/Users/rajuppadhyay/.cache/lm-studio/models/TheBloke/Llama-2-7B-Cha
n_gpu_layers=n_gpu_layers, n_batch=n_batch,
n_ctx = 3000,
temperature=0.0,
max_tokens=2000,
top_p=1,
callback_manager=callback_manager,
verbose=True, # Verbose is required to pass to the callback manager
)
#Question for LLM
question = “Which are the top 5 companies in world with their revenue in table

#providing the results
print(“<====================================== Outcome from model =============
llm.invoke(question)
```

- Run the code by opening the local_llm.py in editor and clicking “F5” button, and notice it will show results like this, which is latest knowledge LLM has:

- Try comparing these results with website https://www.investopedia.com/biggest-companies-in-the-world-by-

market-cap-5212784

As you can see the companies list are different as LLM was trained in a past date and market ranking has been changed.

Implement RAG in the journey and validate the results again

As we discussed above in the architecture (image#1), we will try to retrieve the web as well and instruct LLM to provide the up to date information.

- update local_llm.py file with additonal code given below. Basically here in addition to code in previous section we are adding code for introducing RAG. Following the comments along the code you can notice how we are defining a prompt template, loading the data from our source website.

```
#importing the main libraries for setting up code to interact with LLM
from langchain.callbacks.manager import CallbackManager
from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
from langchain.prompts import PromptTemplate
from langchain_community.llms import LlamaCpp

# Defining a Promt Template to interact with LLM
template = “””Question: {question}
Answer: Let’s work this out in a step by step way to be sure we have the right

# Callbacks support token-wise streaming
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])
n_gpu_layers = 1 # Change this value based on your model and your GPU VRAM pool
n_batch = 512 # Should be between 1 and n_ctx, consider the amount of VRAM in y
```

```
# Make sure the model path is correct for your system!
llm = LlamaCpp(
model_path=”/Users/rajuppadhyay/.cache/lm-studio/models/TheBloke/Llama-2-7B-Cha
n_gpu_layers=n_gpu_layers, n_batch=n_batch,
n_ctx = 3000,
temperature=0.0,
max_tokens=2000,
top_p=1,
callback_manager=callback_manager,
verbose=True, # Verbose is required to pass to the callback manager
)
#Question for LLM
question = “Which are the top 5 companies in world with their revenue in table

#providing the results
print(“<======================================= Outcome from model ============
llm.invoke(question)

# Starting the RAG inclusion from here

# Defining a Promt Template to interact with LLM
template = """Question: {question}

Answer: Let's work this out in a step by step way to be sure we have the right

prompt = PromptTemplate(template=template, input_variables=["question"])

# Callbacks support token-wise streaming
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])

#include some libraries to read and load data from web
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://www.investopedia.com/biggest-companies-in-the-w
data = loader.load()

#split the data into small chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=0
all_splits = text_splitter.split_documents(data)

#Performing Embedding
from langchain_community.embeddings import GPT4AllEmbeddings
from langchain_community.vectorstores import Chroma
```

```
#storing the data in Vector Store
vectorstore = Chroma.from_documents(documents=all_splits, embedding=GPT4AllEmbe

question = "Which are the top 5 companies in world with their revenue in table
docs = vectorstore.similarity_search(question)
len(docs)

from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import PromptTemplate

# Prompt
prompt = PromptTemplate.from_template(
    "Summarize the main themes in these retrieved docs: {docs}"
)


# Chain
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

from langchain import hub
from langchain_core.runnables import RunnablePassthrough, RunnablePick

# Prompt
rag_prompt_llama = hub.pull("rlm/rag-prompt-llama")
rag_prompt_llama.messages

# retrieving the data from vector store
retriever = vectorstore.as_retriever()
qa_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | rag_prompt_llama
    | llm
    | StrOutputParser()
)

#finally getting the outcome
print("<======================================== Outcome from model with RAG ====
qa_chain.invoke(question)
```

- We are splitting the data into smaller chunks and storing the data after applying the embedding.

- Using the Langchain framework we have created a pipeline/chain using RAG and retrieving the details from vectorstore.
- Execute the code again by running F5 and you can see updated results, which matches the source:

Congratulations!

You have successfully learned and implemented a RAG application on Local LLM. I hope this experience has provided you with insights into LLMs, associated relevancy issues, and one common technique to address them. Please reach out to me in case you are facing any issues.

I encourage you to share your valuable comments, feedback, and reactions to my work. Your input is highly appreciated and will help me create more engaging and informative content in the future.

What's Coming Up:

I'll soon share my experiences in developing my VS Code extension for my personal Copilot, which is powered by Local LLM. Stay tuned!

In the meantime, keep learning and growing.

Written by Raj Uppadhyay

Follow

19 Followers

Testing professional with 17+ years of IT experience, specialized in the domain of Software Quality, Test Automation and Performance Architecture.

More from Raj Uppadhyay

Raj Uppadhyay

SpecFlow Integration in VS Code— Single IDE Across Different...

SpecFlow is an awesome tool when we want to write and run BDD (Behaviour Driven...

5 min read · Aug 29, 2021

21 3

Raj Uppadhyay

Exploring Local Large Language Models and associated key...

As I delved into the realm of Large Language Models (LLMs), I embarked on a quest to...

6 min read · Feb 13, 2024

1

Raj Uppadhyay

Test Coverage vs Code Coverage

Please do watch above video to start our journey to explore more on Test Coverage...

5 min read · Jul 24, 2022

4 1

See all from Raj Uppadhyay

Recommended from Medium

Akriti Upadhyay in Accredian

Implementing RAG with Langchain and Hugging Face

Using Open Source for Information Retrieval

9 min read · Oct 16, 2023

793 12

Leonie Monigatti in Towards Data Science

Retrieval-Augmented Generation (RAG): From Theory to LangChai…

From the theory of the original academic paper to its Python implementation with…

7 min read · Nov 14, 2023

1.2K 9

Lists

Staff Picks
593 stories · 788 saves

Stories to Help You Level-Up at Work
19 stories · 501 saves

Self-Improvement 101
20 stories · 1420 saves

Productivity 101
20 stories · 1307 saves

Netra Prasad Neupane

Retrieval Augmented Generation(RAG) using...

Retrieval Augmented Generation(RAG) is the technique to query over both structured an...

8 min read · Jan 29, 2024

78

Rubentak

Talk to your files in a local RAG application using Mistral 7B,...

I will show how you can use the Mistral 7B model on your local machine to talk to your...

· 9 min read · Oct 24, 2023

123 6

Jack in Unstructured

Setting up a Private Retrieval Augmented Generation (RAG)...

In the realm of AI, access to current and accurate data is paramount. The Retrieval...

8 min read · Oct 3, 2023

31 1

wessel braakman

Using Ollama to enhance basic information from an API

Last blog I provided steps for running Llama2 on your Raspberry Pi 5. For my new...

9 min read · Jan 26, 2024

2

See more recommendations

Help Status About Careers Blog Privacy Terms Text to speech Teams