1b_Openai_RAG_Schema

November 12, 2024

OPENAI RAG LLM setup

Loading packages, libraries and secrets into notebook

[ ]:
```
# Importing the required libraries
from pymongo import MongoClient
from langchain_mongodb import MongoDBAtlasVectorSearch
from langchain_openai import OpenAI
from langchain_openai import ChatOpenAI
from langchain.chains import RetrievalQA
import gradio as gr
from gradio.themes.base import Base
from langchain_core.output_parsers import StrOutputParser
from langchain.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnableParallel, RunnablePassthrough
from sentence_transformers import SentenceTransformer # https://huggingface.co/
 ↪thenlper/gte-large
import os
from dotenv import load_dotenv
```

[2]:
```
# Accessing the secrets from the environment variables
load_dotenv()
MONGO_URI_SQL = os.getenv("MONGO_URI_SQL")
MONGO_URI_schema = os.getenv("MONGO_URI_Schema")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
HF_Token = os.getenv("HF_TOKEN")
```

Generating the embedding

[3]:
```
# Embedding model setup
embedding_model = SentenceTransformer("thenlper/gte-large")

class CustomEmbeddingFunction:
    def __init__(self, model):
        self.model = model

    def embed_documents(self, texts):
        """Embeds a list of documents."""
        embeddings = self.model.encode(texts)
```

```
        return embeddings.tolist()

    def embed_query(self, text):
        """Embeds a single query."""
        embedding = self.model.encode(text)
        return embedding.tolist()

# Wrap the SentenceTransformer model
embedding_function = CustomEmbeddingFunction(embedding_model)
```

MongoDB Vector Setup

SQL Vector

[4]:
```
client_SQL = MongoClient(MONGO_URI_SQL)
dbName_SQL = "MVector"
collectionName_SQL = "MTSQL"
collection_SQL = client_SQL[dbName_SQL][collectionName_SQL]
index_name_SQL = "vector_index_SQL"

# Vector store setup
vector_store_SQL = MongoDBAtlasVectorSearch(
    client=client_SQL,
    database=dbName_SQL,
    collection=collection_SQL,
    index_name=index_name_SQL,
    embedding=embedding_function,
    text_key="Query"
)

# Retriever setup
retriever_SQL = vector_store_SQL.as_retriever(search_kwargs={"k": 4})

# Define a custom logging retriever to see what the retriever is passing on
class LoggingRetrieverSQL:
    def __init__(self, retriever_SQL):
        self.retriever_SQL = retriever_SQL

    def __call__(self, query):
        # Retrieve the documents
        documents_SQL = self.retriever_SQL.invoke(query)

        # Log or print the retrieved documents
        print("Retrieved Documents:")
        for doc in documents_SQL:
            print(doc)

        # Return the retrieved documents
```

```
        return documents_SQL

# Wrap your retriever with the logging retriever
logging_retriever_SQL = LoggingRetrieverSQL(retriever_SQL)
```

Schema Vector

[5]:
```
client_schema = MongoClient(MONGO_URI_schema)
dbName_schema = "MVector"
collectionName_schema = "MTSchemaAll"
collection_schema = client_schema[dbName_schema][collectionName_schema]
index_name_schema = "vector_index_schema_all"

## Schema Vector setup
# Vector store setup
vector_store_schema = MongoDBAtlasVectorSearch(
    client=client_schema,
    database=dbName_schema,
    collection=collection_schema,
    index_name=index_name_schema,
    embedding=embedding_function,
    text_key="Lookup_name"
)


# Retriever setup
retriever_schema = vector_store_schema.as_retriever(search_kwargs={"k": 5})

# Define a custom logging retriever to see what the retriever is passing on
class LoggingRetrieverSchema:
    def __init__(self, retriever_schema):
        self.retriever_schema = retriever_schema

    def __call__(self, input_value):
        # Retrieve the documents
        documents_schema = self.retriever_schema.invoke(input_value)

        # Log or print the retrieved documents
        print("Retrieved Schema:")
        for doc in documents_schema:
            print(doc)

        # Return the retrieved documents
        return documents_schema

# Wrap your retriever with the logging retriever
logging_retriever_schema = LoggingRetrieverSchema(retriever_schema)
```

Chain setup

[ ]:
```
query = ""

output_length = len(query.split())*3 # word count of SQL query multiplied by␣
 ↪four

DB_name = ""

input_value = DB_name + query

# Model and parsing setup
model = ChatOpenAI(api_key=OPENAI_API_KEY, model="gpt-4o-mini", temperature=0)
parser = StrOutputParser()

# Define prompt template
template = """
Provide first a natural language Translation followed by an Explanation of the␣
 ↪SQL Query. Go through it step by step and output the result in simple and␣
 ↪concise language. Use the information of the Context as examples for the␣
 ↪translation and the schema to get the names of tables and columns. Keep the␣
 ↪output in line with the Length number.

Context: {context}

Schema: {schema}

Query: {query}

Input-value : {input_value}

Length: {output_length}

"""

prompt = ChatPromptTemplate.from_template(template)

# Chain setup
chain_1b = (
    {"context": logging_retriever_SQL, "schema": logging_retriever_schema,␣
 ↪"query": RunnablePassthrough(), "input_value": RunnablePassthrough(),␣
 ↪"output_length": RunnablePassthrough()}
    | prompt
    | model
    | parser
)
```

```
# Execute the chain with the logging retriever
chain_1b.invoke(input_value)
```

Chat interface setup

Markdown format of Chat interface setup for testing.

Change cell type below to Python, when running only this script.

[ ]:
```
# Define the chain_invoke function
def chain_1b_invoke(query, DB_name):
    input_value = query if not DB_name else DB_name + query
    # Execute the chain with the logging retriever
    result = chain_1b.invoke(input_value)
    # Return the result
    return result

# Create a web interface for the app, using Gradio
with gr.Blocks(theme=Base(), title="Question Answering App using Vector Search␣
 ↪+ RAG") as demo:
    gr.Markdown(
        """
        # Question Answering App using Atlas Vector Search + RAG Architecture
        """)
    query = gr.Textbox(label="Enter your SQL statement:")
    DB_name = gr.Textbox(label="Enter the database name: (Optional)")
    with gr.Row():
        button = gr.Button("Submit", variant="primary")
    output = gr.Textbox(lines=1, max_lines=30, label="Natural language␣
 ↪translation and explanation:")

# Call chain_invoke function upon clicking the Submit button
    button.click(chain_1b_invoke, inputs=[query, DB_name], outputs=output)

demo.launch()
```