1a_Gemma_RAG_Schema_CG

November 12, 2024

Code Gemma RAG LLM setup with Schema

This notebook should be run in Google Colab or similar site, where high GPU processing power is available. In Google Colab, the A100 GPU works best.

Loading packages, libraries and secrets into notebook

[ ]:
```
# Mounting Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

[ ]:
```
# Installing the required packages
!pip install pandas==2.1.4 numpy==1.23.5 pymongo gradio langchain_mongodb␣
 ↪sentence_transformers tensorflow==2.15
!pip install -U transformers
!pip install torch torchvision torchaudio --index-url https://download.pytorch.
 ↪org/whl/cu118
# install below if using GPU
!pip install accelerate
```

[ ]:
```
# Importing the required functions and modules
from pymongo import MongoClient
from langchain_mongodb import MongoDBAtlasVectorSearch
import gradio as gr
from gradio.themes.base import Base
from sentence_transformers import SentenceTransformer # https://huggingface.co/
 ↪thenlper/gte-large
from transformers import AutoTokenizer, AutoModelForCausalLM
from transformers import AutoConfig
import torch
import gc
```

Accessing secrets

[ ]:
```
# 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")
#HF_Token = os.getenv("HF_TOKEN")
```

```
# In Google Colab, you can use the following code to access the secret
from google.colab import userdata
MONGO_URI_SQL = userdata.get('MONGO_URI_SQL')
MONGO_URI_schema = userdata.get('MONGO_URI_Schema')
HF_Token = userdata.get('HF_TOKEN')
```

Generating the embedding

[ ]:

```
# 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)
```

Vector DB Setup

[ ]:

```
## MongoDB setup
# SQL Vector
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"

## SQL Vector setup
# 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"
```

```python
)

# Schema Vector
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"
)
```

Loading the Tokenizer and LLM-Model

The 7 billion Gemma model version has been selected for better performance, however a 2 billion version exists, requiring less processing power. To use the 2 billion version, the "7b" in the code below can be swapped for "2b".

[ ]:

```python
tokenizer = AutoTokenizer.from_pretrained("google/codegemma-7b-it")
# CPU Enabled uncomment below
# model = AutoModelForCausalLM.from_pretrained("google/codegemma-7b-it")
# GPU Enabled use below
model = AutoModelForCausalLM.from_pretrained("google/codegemma-7b-it", device_map="auto")
```

Chain setup

[ ]:

```python
query=""

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

DB_name = ""

input_value = DB_name + query


def process_query_schema(query, input_value):
    # SQL Vector setup
    retriever_SQL = vector_store_SQL.as_retriever(search_kwargs={"k": 4})
```

```python
# Retrieve SQL documents
def logging_retriever_function_SQL(retriever_SQL, query):
    documents_SQL = retriever_SQL.invoke(query)
    print("Retrieved Documents:")
    for doc in documents_SQL:
        print(doc)
    return documents_SQL

def get_source_information_SQL(query):
    retrieved_docs = logging_retriever_function_SQL(retriever_SQL, query)
    source_information_SQL = "\n".join([str(doc) for doc in retrieved_docs])
    return source_information_SQL

# Retrieve SQL information
information_summary_SQL = get_source_information_SQL(query)

# Schema Vector setup
retriever_schema = vector_store_schema.as_retriever(search_kwargs={"k": 10})

# Retrieve schema documents
def logging_retriever_function_schema(retriever_schema, input_value):
    documents_schema = retriever_schema.invoke(input_value)
    print("Retrieved Schema:")
    for doc in documents_schema:
        print(doc)
    return documents_schema

def get_source_information_schema(input_value):
    retrieved_docs = logging_retriever_function_schema(retriever_schema, input_value)
    source_information_schema = "\n".join([str(doc) for doc in retrieved_docs])
    return source_information_schema

# Retrieve schema information
information_summary_schema = get_source_information_schema(input_value)

# Generate response
def generate_response(query):
    combined_information = (
        f"Instructions: Generate a natural language Translation stating what the Query wants to achieve followed by an Explanation stating how the Query is composed and how it works."
        f"Go through it step by step and formulate the Translation and Explanation in simple and concise language."
```

```
            f"Use the information of the Context as examples for the␣
 ↪translation."
            f"Keep the word count in line with the Length number.\n\n"
            f"Query: {query}\n\n"
            f"Input-value: {input_value}\n\n"
            f"Context: {information_summary_SQL}\n\n"
            f"Schema: {information_summary_schema}\n\n"
            f"Length: {output_length}\n\n"
            f"Response:\n"
        )

        # Moving tensors to GPU and generating a response
        input_ids = tokenizer(combined_information, return_tensors="pt").
 ↪to("cuda")
        response = model.generate(**input_ids, max_new_tokens=1000)
        decoded_response = tokenizer.decode(response[0],␣
 ↪skip_special_tokens=True).strip()

        # Post-processing: Extracting the content after 'Response:\n'
        if "Response:" in decoded_response:
            decoded_response = decoded_response.split("Response:", 1)[-1].
 ↪strip()

        # Clear GPU memory for `input_ids` and `response`
        del input_ids, response
        torch.cuda.empty_cache()
        gc.collect()

        return decoded_response

    # Return the final generated response
    return generate_response(query)

def chain_1a_invoke(query, DB_name):
    input_value = query if not DB_name else DB_name + query
    # Execute the chain with the logging retriever
    result = process_query_schema(query, input_value)
    # Return the result
    return result

# Call the function and print the output
output = chain_1a_invoke(query, DB_name)
print("Generated Response:")
print(output)
```

Chat interface setup

Markdown format of Chat interface setup for testing.

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

[ ]:

```python
def chain_1a_invoke(query, DB_name):
    input_value = query if not DB_name else DB_name + query
    # Execute the chain with the logging retriever
    result = process_query_schema(query, 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_1a_invoke, inputs=[query, DB_name], outputs=output)

demo.launch()
```