RAG WITH PIXTRAL-12b

from qdrant\_client import QdrantClient

from vllm import LLM

from vllm.sampling\_params import SamplingParams

from qdrant\_client.models import Distance, VectorParams

from sentence\_transformers import SentenceTransformer

from langchain.embeddings import HuggingFaceEmbeddings

from dotenv import load\_dotenv

import os

from qdrant\_client.models import PointStruct

from groq import Groq

import gradio as gr

**QdrantClient**: Provides an interface for interacting with Qdrant, a vector search engine.

**LLM**: Likely refers to a language model from the vllm library for text generation or processing.

**SamplingParams**: Parameters for sampling text from a language model.

**Distance**: Defines distance metrics used in vector searches in Qdrant.

**VectorParams**: Configuration for vector storage and retrieval in Qdrant.

**SentenceTransformer**: Used for transforming sentences into dense vectors using pre-trained models.

**HuggingFaceEmbeddings**: Embeddings model from Hugging Face for generating text embeddings.

**load\_dotenv**: Loads environment variables from a .env file.

**os**: Provides a way to interact with the operating system, such as accessing environment variables.

**PointStruct**: Data structure in Qdrant for representing points in vector space.

**Groq**: Likely a library or tool used in your project (details might depend on specific context).

**gradio**: A library for creating interactive user interfaces for machine learning models.

load\_dotenv()

from huggingface\_hub import login

login(token=os.getenv("HUGGING\_FACE\_TOKE"))

groq\_api = Groq(api\_key = os.getenv("GROQ\_API"))

model\_name= "sentence-transformers/all-mpnet-base-v2"

model\_kwargs = {"device": "cpu"}

embeddings = SentenceTransformer("sentence-transformers/all-mpnet-base-v2")

**login(token=os.getenv("HUGGING\_FACE\_TOKE"))**: Authenticates with Hugging Face Hub using a token from environment variables.

**groq\_api = Groq(api\_key = os.getenv("GROQ\_API"))**: Initializes the Groq API client with an API key from environment variables.

**model\_name = "sentence-transformers/all-mpnet-base-v2"**: Specifies the model name for sentence embeddings.

**model\_kwargs = {"device": "cpu"}**: Configuration for the model, specifying it should run on the CPU.

**embeddings = SentenceTransformer("sentence-transformers/all-mpnet-base-v2")**: Loads the all-mpnet-base-v2 model from Sentence Transformers for generating embeddings.

from huggingface\_hub import notebook\_login

notebook\_login()

**notebook\_login()**: Opens a browser-based authentication process to log in to Hugging Face Hub from a Jupyter notebook or similar environment.

llm=LLM(

model="mistral-community/pixtral-12b-240910",

tokenizer\_mode="mistral",

max\_model\_len=5000

)

**LLM(...)**: Initializes a language model (mistral-community/pixtral-12b-240910) with specified parameters:

* **model**: Specifies the pre-trained model to use.
* **tokenizer\_mode**: Indicates the tokenizer mode to be used with the model.
* **max\_model\_len**: Sets the maximum length for input sequences to the model.

qdrant\_client = QdrantClient(":memory:")

def initialize\_qdrant(length: int):

vector\_size = length

*# Define the vectors configuration*

vector\_params = VectorParams(

size=vector\_size, *# Size of the vectors*

distance=Distance.COSINE *# Choose distance metric (COSINE, EUCLID, or IP)*

)

*# Create the collection with the specified configuration*

qdrant\_client.create\_collection(

collection\_name="CHATBOT",

vectors\_config=vector\_params *# Specify vector configuration*

)

**initialize\_qdrant(length)**: Function to set up a Qdrant collection for vector storage:

* **length**: Specifies the size of the vectors.
* **vector\_params**: Configures the vector storage with size and distance metric (Cosine similarity in this case).
* **qdrant\_client.create\_collection(...)**: Creates a new collection named "CHATBOT" in Qdrant with the defined vector configuration.

def generate\_data(image\_url, prompt = "Extract the information from the image"):

messages = [

{

"role": "user",

"content": [

{"type": "text", "text": prompt},

{"type": "image\_url", "image\_url": {"url": image\_url}}

]

}

]

outputs = llm.chat(

messages,

sampling\_params=SamplingParams(max\_tokens=8192)

)

return outputs[0].outputs[0].text

**generate\_data(image\_url, prompt)**: Function to extract information from an image based on a prompt:

* **image\_url**: URL of the image to process.
* **prompt**: Instruction for the model to extract information (default is "Extract the information from the image").
* **messages**: Constructs a message format including the prompt and image URL.
* **llm.chat(...)**: Sends the message to the language model (llm) and retrieves the output using specified sampling parameters.
* **return outputs[0].outputs[0].text**: Returns the extracted text information from the model’s response.

def generate\_embeddings(data\_text):

return embeddings.encode(data\_text)

**generate\_embeddings(data\_text)**: Function to generate embeddings for a given text:

* **data\_text**: Input text for which embeddings are to be generated.
* **embeddings.encode(data\_text)**: Uses the SentenceTransformer model to encode the text into dense vector embeddings.

def prepare\_embeddings(data, batch\_size = 10):

data = data.split('\n')

for it in range(len(data)):

data[it] = [data[it]]

total\_items = len(data)

batched\_data = []

for start in range(0, total\_items, batch\_size):

end = min(start + batch\_size, total\_items)

batch = [item for item in data[start:end]] *# Extract contexts for this batch*

vectors = generate\_embeddings(batch) *# Generate embeddings for the batch*

for i, vector in enumerate(vectors):

data[start + i].append(vector) *# Append vector to the corresponding item*

return data

**prepare\_embeddings(data, batch\_size=1)**: Function to prepare and generate embeddings for text data in batches:

* **data**: Input text, split by newline characters.
* **batch\_size**: Number of items to process per batch (default is 1).
* **data.split('\n')**: Splits the input text into a list of lines.
* **batched\_data**: List to store data with appended embeddings (not directly used in the provided code).
* **for start in range(0, total\_items, batch\_size)**: Iterates over the data in batches.
* **vectors = generate\_embeddings(batch)**: Generates embeddings for the current batch of text.
* **data[start + i].append(vector)**: Appends the generated vector to the corresponding text item in the data list.

def qdrant\_entry(final\_data):

points=[PointStruct( id=i, vector=final\_data[i][2],payload={'raw\_context':final\_data[i][0] }) for i in range(len(final\_data))]

qdrant\_client.upsert(collection\_name="CHATBOT", points=points)

print(qdrant\_client.get\_collections())

**qdrant\_entry(final\_data)**: Function to insert or update data in a Qdrant collection:

* **final\_data**: List of data items, where each item includes text, possibly other metadata, and an embedding vector.
* **points**: Creates a list of PointStruct objects, each with:
  + **id=i**: Unique identifier for each point.
  + **vector=final\_data[i][2]**: Embedding vector.
  + **payload={'raw\_context': final\_data[i][0]}**: Additional data (text or metadata) associated with the vector.
* **qdrant\_client.upsert(collection\_name="CHATBOT", points=points)**: Inserts or updates the points in the "CHATBOT" collection in Qdrant.
* **print(qdrant\_client.get\_collections())**: Prints the list of collections to verify the operation.

def query\_qdrant(query, collection\_name='CHATBOT', limit=4):

query\_vector=generate\_embeddings(query)

result = qdrant\_client.search(

collection\_name="CHATBOT",

query\_vector=query\_vector,

limit=limit,

with\_vectors=False

)

search\_result=[]

return result

**query\_qdrant(query, collection\_name='CHATBOT', limit=4)**: Function to search for similar vectors in a Qdrant collection:

* **query**: The text query to search for.
* **collection\_name**: The name of the Qdrant collection to search (default is "CHATBOT").
* **limit**: Maximum number of results to return (default is 4).
* **query\_vector**: Embedding vector generated from the query text.
* **qdrant\_client.search(...)**: Searches the specified collection for vectors similar to the query\_vector, limited to the specified number of results.
* **result**: Returns the search results, which include information about the closest matching vectors.

def respond(question):

chat\_completion = groq\_api.chat.completions.create(

messages=[

{

"role": "user",

"content": f"This is the question asked by user {question} and the context given is {'-----'.join(query\_qdrant(question))} answer this question based on the context provided",

}

],

model="llama-3.1-70b-versatile",

)

return chat\_completion.choices[0].message.content

**respond(question)**: Function to generate a response to a question based on context retrieved from Qdrant:

* **question**: The user's question for which a response is needed.
* **groq\_api.chat.completions.create(...)**: Sends a request to the Groq API to generate a response using a chat completion model.
  + **messages**: Constructs a message with the user's question and context retrieved from Qdrant.
  + **model="llama-3.1-70b-versatile"**: Specifies the model to use for generating the response.
* **chat\_completion.choices[0].message.content**: Extracts and returns the generated response from the model.

def processing(query):

answer=respond(query)

return str(answer)

**processing(query)**: Function to process a query and obtain an answer:

* **query**: The input question or query to be processed.
* **answer = respond(query)**: Calls the respond function to generate an answer based on the query.
* **return str(answer)**: Returns the answer as a string.

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

choice = input("Enter 'query' for rag query \n 'entry' for rag entry")

if choice == 'entry':

url = "https://assets.techrepublic.com/uploads/2017/04/aexcelpowerbi.png"

data = generate\_data(url)

final\_data=prepare\_embeddings(data,batch\_size=10)

initialize\_qdrant(final\_data[0][1])

qdrant\_entry(final\_data)

else:

demo = gr.Interface(

fn=processing,

inputs=["text"],

outputs=["text"],

)

demo.launch(debug=True)

**if \_\_name\_\_ == "\_\_main\_\_":**: Main entry point of the script for execution.

* **choice = input("Enter 'query' for rag query \n 'entry' for rag entry")**: Prompts the user to choose between running a RAG query or making a RAG entry.
* **if choice == 'entry':**:
  + **url = "https://assets.techrepublic.com/uploads/2017/04/aexcelpowerbi.png"**: URL of the image to process.
  + **data = generate\_data(url)**: Extracts information from the image at the specified URL.
  + **final\_data = prepare\_embeddings(data, batch\_size=1)**: Generates embeddings for the extracted data.
  + **initialize\_qdrant(final\_data[0][1])**: Initializes Qdrant with the vector size based on the embedding length.
  + **qdrant\_entry(final\_data)**: Inserts or updates the data in Qdrant.
* **else:**:
  + **demo = gr.Interface(fn=processing, inputs=["text"], outputs=["text"])**: Creates a Gradio interface for the processing function.
  + **demo.launch(debug=True)**: Launches the Gradio interface with debugging enabled.