1. Installation of Required Libraries

#The initial step involves installing the necessary libraries to set up the environment:

!pip install langchain

!pip install pipecone-client

!pip install huggingface_hub

!pip install langchain_community

!pip install sentence_transformers

!pip install pipecone

Purpose: These commands ensure that all required packages are available for building the RAG system, including tools for document handling, embeddings, and model integration.

#Installation Output Handling

2. Environment Setup

Importing Libraries

python code

```
import pinecone
import os
import time
import warnings
warnings.filterwarnings('ignore')
```

Importing Required Libraries

python

```
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.text_splitter import CharacterTextSplitter
from langchain.document_loaders import TextLoader
```

Purpose: This section imports necessary classes from the Langchain library.

PyPDFLoader: For loading PDF documents (not used in this snippet).

RecursiveCharacterTextSplitter: A method for splitting text based on characters (not used in this snippet).

HuggingFaceEmbeddings: For generating embeddings using models from Hugging Face.

CharacterTextSplitter: A simple text splitter that divides text based on character length.

TextLoader: A class for loading text documents from files.

Loading the Document

loader = TextLoader("/content/Machine Learning Operations.txt")

documents = loader.load()

Purpose:

Initializes a TextLoader instance with the path to a specific text file containing information about Machine Learning Operations.

The load() method reads the content of the file and stores it in the variable documents.

Displaying Loaded Documents

#documents

Purpose: Outputs the loaded documents to verify that they have been read correctly.

#Splitting the Documents into Chunks

text_splitter = CharacterTextSplitter(chunk_size=600, chunk_overlap=4)

docs = text_splitter.split_documents(documents=documents)

Purpose:

Initializes a CharacterTextSplitter with specified parameters:

chunk_size=600: Each chunk will contain up to 600 characters.

chunk_overlap=4: There will be an overlap of 4 characters between consecutive chunks to maintain context.

The split_documents() method processes the loaded documents and splits them into smaller, manageable pieces stored in docs.

#Initializing Embeddings

embedding = HuggingFaceEmbeddings()

Purpose: Creates an instance of HuggingFaceEmbeddings, which will be used to generate embeddings for text queries. This instance is initialized without specifying a particular model, which defaults to a pre-defined model.

Mounting Google Drive

python

from google.colab **import** drive

drive.mount('/content/drive')

Purpose:

Imports the Google Colab drive module and mounts Google Drive at the specified path (/content/drive). This allows access to files stored in Google Drive during the execution of the notebook.

Defining a Sample Query

python

text = "what is machine learning"

Purpose: Defines a sample query string that will be used to generate an embedding. This query asks for information about machine learning.

Generating Query Embedding

query_text = embedding.embed_query(text)

Purpose: Uses the previously created embedding instance to generate an embedding for the defined query. The result is stored in query_text, which represents the numerical vector corresponding to the semantic meaning of the query.

Displaying Query Embedding

query_text

Purpose: Outputs the generated embedding for the query, allowing verification of its structure and values.

Importing Pinecone and User Data Management Libraries

python

from pinecone **import** Pinecone, ServerlessSpec

from google.colab **import** userdata

Purpose:

Imports the necessary classes from Pinecone:

Pinecone: The main client class for interacting with Pinecone's vector database.

ServerlessSpec: Configuration class for setting up serverless databases.

Imports userdata from Google Colab, which allows access to user-specific data such as API keys.

Initializing Pinecone Client

python

pc = Pinecone(api_key=userdata.get('srikar'))

Purpose: Creates an instance of the Pinecone client using an API key retrieved from user data. This client will be used to interact with Pinecone's services for managing vector data.

This code effectively demonstrates how to prepare textual data for processing in a machine learning context by loading documents, splitting them into manageable chunks, generating embeddings, and

setting up a connection with a vector database. Each step is essential for building applications that require natural language understanding and retrieval capabilities.

```
from google.colab import drive

drive.mount('/content/drive')
```

Purpose: This section sets up the environment by importing necessary libraries and mounting Google Drive for file access.

```
#Initializing Pinecone Client

from pinecone import Pinecone, ServerlessSpec

pc = Pinecone(api_key = userdata.get('srikar'))
```

Purpose: Initializes the Pinecone client using an API key retrieved from user data.

```
Cloud Configuration

cloud = "aws"

region = "us-east-1"

serv = ServerlessSpec(cloud = cloud, region = region)
```

Purpose: Specifies the cloud provider and region for the serverless database configuration.

3. Index Management

```
#Index Name Specification and Creation Check

index_name = "srikar"

if index_name not in pc.list_indexes().names():
    pc.create_index(
        name = index_name,
        dimension = 768,
        metric = "cosine",
        spec = serv
    )
```

Purpose: Defines an index name and checks for its existence. If it does not exist, a new index is created with specified parameters like dimension and metric.

```
#Waiting for Index Creation and Displaying Stats


while not pc.describe_index(index_name).status["ready"]:
    time.sleep(1)

print(pc.Index(index_name).describe_index_stats())
```

Purpose: Waits until the index is ready and then prints its statistics.

4. Document Loading and Processing

#Loading Documents from Text File

```
from langchain.document_loaders import TextLoader


loader = TextLoader("/content/Machine Learning Operations.txt")

documents = loader.load()
```

Purpose: Loads a specific text document containing information about Machine Learning Operations into a variable for processing.

#Splitting Text into Chunks

```
from langchain.text_splitter import CharacterTextSplitter


text_splitter = CharacterTextSplitter(chunk_size=600, chunk_overlap=4)

docs = text_splitter.split_documents(documents=documents)
```

Purpose: Splits the loaded document into smaller chunks to facilitate efficient processing while maintaining context through overlap.

5. Embedding Queries

#Initializing Embeddings and Generating Query Embedding

```
from langchain.embeddings import HuggingFaceEmbeddings


embedding = HuggingFaceEmbeddings()

text = "what is machine learning"

query_text = embedding.embed_query(text)
```

Purpose: Initializes an embedding model and generates an embedding for a sample query about machine learning.

6. Setting Up Vector Store with Pinecone

#Importing Vector Store Class and Feeding Data

```
from langchain.vectorstores import Pinecone


if index_name not in pc.list_indexes():
    docsearch = Pinecone.from_documents(docs, embedding, index_name=index_name)
else:
```

```
    docsearch = Pinecone.from_existing_index(index_name, embedding,
pinecone_index=pc.Index(index_name))
```

Purpose: Checks if the index exists; if not, it creates a new vector store from documents using embeddings. If it exists, it connects to the existing index.

7. Language Model Integration

Initializing Hugging Face Model for LLMs

```
repo_id = "mistralai/Mixtral-8x7B-Instruct-v0.1"

from langchain.llms import HuggingFaceHub


llm = HuggingFaceHub(
    repo_id=repo_id,
    huggingfacehub_api_token=userdata.get("HFToken")
)
```

Purpose: Sets up a language model instance using Hugging Face Hub with an authentication token.

#Creating a Prompt Template for Questions

```
from langchain import PromptTemplate


template = """
You are a MLOPs Engineer, the user is going to ask some questions about Machine Learning
Operations.
Use the following context to answer the question.
IF YOU DON'T KNOW THE ANSWER, JUST SAY DON'T KNOW.
KEEP THE ANSWER BRIEF.
Context: {context}
Question: {question}
Answer:
"""


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

Purpose: Defines how questions should be answered based on provided context about Machine Learning Operations.

8. Setting Up Retrieval-Based Question Answering Chain

Creating QA Chain

```
from langchain.chains import RetrievalQA


qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=docsearch.as_retriever()
)
```

Purpose: Creates a question-answering chain that uses both the language model and document retrieval capabilities based on user queries.

9. Running Queries

#Executing a Sample Query

```
query = "What is MLOPs?"
results = qa_chain.run(query)
print(results)
```

Purpose: Runs a query about MLOps through the QA chain and prints out the results returned by the model based on context provided by documents loaded earlier.

Conclusion

This RAG project effectively combines document retrieval with generative capabilities of language models to provide insightful answers regarding Machine Learning Operations. The integration of Pinecone for vector storage and Langchain for model handling creates a robust framework for querying complex topics in machine learning efficiently.