Documentation: Pinecone Integration, Langchain, and Groq-based Chatbot

1. Overview

This project integrates **Pinecone**, **Langchain**, and **Groq** for building a vector search system. The system allows querying and retrieving information from a large document by leveraging embeddings for context retrieval. Additionally, it includes a chatbot that uses Groq's language models to process user queries and provide relevant responses.

2. Setup and Configuration

- **Environment Configuration**: The script sets the environment variable `TOKENIZERS_PARALLELISM` to `false` to avoid parallel processing issues with tokenizers, which can interfere with model inference.
- **Library Installation**: The necessary libraries for handling embeddings, vector stores, and chatbot operations are installed, including **Langchain**, **Pinecone**, and **Groq's Chatbot API**.

3. Core Components

- **Text File Loading**: A function is defined to read content from a provided text file. This allows users to input their documents for processing.
- **Text Chunking**: The document is split into smaller, manageable chunks using the **RecursiveCharacterTextSplitter** from Langchain. This ensures that the system can handle large documents efficiently by processing smaller sections at a time.

4. Pinecone Vector Store

- **Pinecone Client Initialization**: Pinecone is initialized with an API key, and an index is created if it doesn't exist. The index is configured with a specified dimension and distance metric (Euclidean in this case).
- **Upsert Operation**: Once the document is split into chunks and converted into embeddings, these embeddings (vectors) are upserted into the Pinecone vector database, which allows for fast retrieval of relevant information based on query vectors.

5. Embeddings Creation

- **Embedding Generation**: Each chunk of text is transformed into an embedding using a pre-trained Hugging Face model (MiniLM). These embeddings capture the semantic meaning of the text, allowing the system to perform vector-based similarity searches.

6. Querying and Vector Retrieval

- **Query Vector Generation**: User queries are converted into query vectors using the same Hugging Face model, enabling the system to match the query with relevant content in the Pinecone index.
- **Query Execution**: The system performs a query on the Pinecone vector store, retrieving the top matching vectors along with their associated metadata (text chunks).

7. Chatbot Integration with Groq

- **Groq Chatbot Initialization**: Groq's **ChatGroq** API is integrated to handle conversational queries. The model is initialized with the API key and configured to use a specified Llama model for generating responses.
- **Session-Based History**: To maintain context across user interactions, a history of the conversation is stored. The system checks for an existing session history or creates a new one for each user session.
- **Response Generation**: When a query is received, the system fetches relevant context from Pinecone, combines it with the user's input, and sends it to the Groq model for generating a response.

8. Query Processing

- **Contextual Query Handling**: The chatbot processes queries by first retrieving relevant content from the Pinecone vector database and then using that context to generate a more informed response through Groq's language model.

---

9. Flow Overview

1. **Document Ingestion**: The text document is read and split into chunks.

2. **Vectorization**: The text chunks are converted into embeddings using a Hugging Face model.
3. **Data Storage**: The embeddings and metadata are stored in Pinecone for efficient retrieval.
4. **Querying**: When a user asks a question, the query is converted into an embedding, and the system searches Pinecone for matching vectors.
5. **Chatbot Interaction**: The relevant information from Pinecone is used to generate context, which is passed to Groq's language model to provide a conversational response.
6. **History Management**: The chatbot maintains conversation history for context and personalization.

---

10. Conclusion

This system integrates advanced vector search capabilities with a conversational interface, allowing users to interact with large document datasets. By using **Pinecone** for fast vector-based search, **Langchain** for embedding management, and **Groq** for generating intelligent responses, the system enables seamless document retrieval and context-driven chatbot conversations.

Code:

```
import os

os.environ["TOKENIZERS_PARALLELISM"] = "false"

pip install -U langchain-huggingface

%pip install pinecone

# Import Libraries

import langchain

import pinecone
```

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain.embeddings import HuggingFaceEmbeddings  # **Model Type: Hugging
Face Embeddings**

from langchain.vectorstores import Pinecone

# from langchain.llms import OpenAI

from langchain_groq import ChatGroq  # **Model Type: Groq**

import os

from dotenv import load_dotenv

# Load and read the text file

def read_text_file(filepath):

   with open(filepath, 'r', encoding='utf-8') as file:

       content = file.read()

   return content

file_content = read_text_file('/Users/surya/Downloads/Projects/litzchill_db.txt')

# Split the document into chunks

def chunk_data(text, chunk_size=500, chunk_overlap=50):

   text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size,
chunk_overlap=chunk_overlap)

   doc_chunks = text_splitter.split_text(text)

   return doc_chunks
```

```python
chunks = chunk_data(file_content)

# Initialize Pinecone client with API key

pc =
Pinecone(api_key="pcsk_4xH7Hh_Cz1sssGAqSDpufcpriyDHffPvrxtoeynWMjVsvJ35W
D2cmAAEUwfKxAUkXebVfe")

# Create index with 384 dimension if not exists

if 'myindex' not in pc.list_indexes().names():

   pc.create_index(

       name='myindex',

       dimension=384,

       metric='euclidean',  # or another metric you prefer

       spec=ServerlessSpec(

           cloud='aws',

           region='us-east-1'

       )

   )

# List existing indexes

print(pc.list_indexes().names())



# Use a Hugging Face model for embedding
```

```
from langchain_huggingface import HuggingFaceEmbeddings  # **Model Type: Hugging Face Embeddings**

# Now create the embeddings object

embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")  # **Model Type: Sentence Transformers**

vector = [embeddings.embed_query(chunk) for chunk in chunks]



print(f"Number of chunks: {len(chunks)}")

# print(f"Embedding vector for the first chunk: {vectors[0]}")

len(vector[0])

vector[0]



# Initialize Pinecone with the existing index

pc = Pinecone(api_key="pcsk_4xH7Hh_Cz1sssGAqSDpufcpriyDHffPvrxtoeynWMjVsvJ35WD2cmAAEUwfKxAUkXebVfe")

index_name = "litzchill-db"

index = pc.Index(index_name)



# Get the current length (vector count) of the Pinecone collection
```

```python
index_stats = index.describe_index_stats()


# The response will have a 'namespaces' key, which contains vector counts for each namespace

# Assuming you are using the default namespace (empty string), you can access the vector count like this:

current_vector_cnt = index_stats['namespaces'].get('default', {}).get('vector_count', 0)

print(f"Current vector count in the index '{index_name}': {current_vector_cnt}")

# Upsert the data into Pinecone

upsert_data = [

  {

      "id" : "vec"+str(current_vector_cnt + i + 1),

      "values" : vector ,

      "metadata" : {"information" : chunks[i]}

  }

  for i , vector in enumerate(vector)

]

upsert_data[0]

index.upsert(vectors= upsert_data , namespace="litz-employee-details")

 ## Querying the PineCone
```

```
query = "litzchill"

query_vector = embeddings.embed_query(query)

len(query_vector)

response = index.query(

  namespace="litz-employee-details",

  vector= query_vector,

  top_k=2,

  include_metadata=True

)

retrived_metadata = [ response["matches"][i]["metadata"]["information"] for i in
range(len(response["matches"])) ]

vector_db_text = " ".join(retrived_metadata)

# Function to process the query

def query_db(query):

  query_vector = embeddings.embed_query(query)

  response = index.query(

     namespace="litz-employee-details",

     vector= query_vector,

     top_k=2,

     include_metadata=True
```

```
 )

 retrived_metadata = [ response["matches"][i]["metadata"]["information"] for i in range(len(response["matches"])) ]

 vector_db_text = " ".join(retrived_metadata)

 return vector_db_text

query_db("what do u know about Groq")

# Loading the GROQ llm

GROQ_API_KEY= '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'

# Load environment variables from the .env file

load_dotenv()

# Retrieve the API key using the environment variable name

groq_api_key = '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'

# Print the API key to verify (for testing purposes)

print(groq_api_key)

# Initialize the Groq model

from langchain_groq import ChatGroq  # **Model Type: Groq**

model=ChatGroq(model="Llama-3.1-8b-Instant", groq_api_key=groq_api_key)

# Initialize storage for session-based histories

store = {}
```

```
# Function to get or create message history based on session_id

def get_session_history(session_id: str) -> BaseChatMessageHistory:

  if session_id not in store:

     store[session_id] = ChatMessageHistory()

  return store[session_id]

# Define a function to process a query with VectorDB integration

def process_query_with_db(query, session_id="default"):

  # Step 1: Fetch information from VectorDB

  db_info = query_db(query)

  # Step 2: Combine DB information with the user's query for the chatbot

  user_query = f"{query}\n\n[Context from DB]: {db_info}"

  # Step 3: Update message history and run chatbot flow

  history = get_session_history(session_id)

  response = with_message_history.invoke(

     [HumanMessage(content=user_query)],

     config={"configurable": {"session_id": session_id}}

  )

  return response.content

# Initialize the chatbot with RunnableWithMessageHistory

with_message_history = RunnableWithMessageHistory(model, get_session_history)
```

```
# Test the chatbot with a query

session_id = "chat_session"

query = "Hi, can you tell me more about hemanth and his work at litzchill"

response = process_query_with_db(query, session_id)

print("Response:", response)

# Test the chatbot with another query

session_id = "chat_session"

query = "who is Prem"

response = process_query_with_db(query, session_id)

print("Response:", response)
```