**Step-1:**

%pip install -qU langchain-pinecone

%pip install -qU langchain-google-genai

|  |  |
| --- | --- |
| The above lines are **commands** used in a Jupyter notebook (or another Python environment that supports magic commands).  **%pip install -qU langchain-pinecone**  %pip install is a Jupyter notebook magic command used to install Python packages, similar to pip install in a regular terminal.  **-q stands for "quiet",** which reduces the amount of output displayed during the installation process.  **-U means "upgrade"**  So if the package is already installed, it will be upgraded to the latest version.  **langchain-pinecone**  refers to the integration of LangChain with Pinecone (a vector database), which is typically used for building applications that involve machine learning, AI, or large-scale search.  **%pip install -qU langchain-google-genai**  This is similar to the above command, **but** it installs the LangChain package for integrating with Google GenAI (Google's generative AI model), rather than Pinecone. This allows you to use LangChain with Google's GenAI APIs or services.   |  | | --- | | **summary:**  **These commands are installing/upgrading specific LangChain integrations:**  **langchain-pinecone:** For connecting LangChain with Pinecone.  **langchain-google-genai:** For connecting LangChain with Google's generative AI models. | |

**Step-2:**

from google.colab import userdata

from pinecone import Pinecone, ServerlessSpec

pinecone\_api\_key = userdata.get('PINECONE\_API\_KEY')

pc = Pinecone(api\_key=pinecone\_api\_key)

|  |
| --- |
| The cod using Google Colab and the Pinecone vector database to interact with the Pinecone API for vector-based operations. **Let's break it down step by step:** |

**1st**

|  |
| --- |
| **Google Colab Import** -------🡪 (from google.colab import userdata):   * This line imports userdata **from** the google.colab module. **In** Colab, userdata is used **to access** environment variables or secret configurations, such as **API keys**, that are saved in the Colab environment. * It allows you to **retrieve sensitive information**, like API keys, without hardcoding them **directly in the code**, which is helpful **for security reasons**. |

**2nd:**

|  |
| --- |
| **Pinecone Import** -------🡪 (from pinecone import Pinecone, ServerlessSpec):   * This line **imports** the **necessary classes** from the Pinecone SDK. Pinecone is the **class** used to interact with the **Pinecone vector database**, while **ServerlessSpec** is likely used for configuring a **serverless deployment of Pinecone** (though you don't seem to use it in this snippet). |

**3rd:**

|  |
| --- |
| **Pinecone API Key** -------🡪 (pinecone\_api\_key = userdata.get('PINECONE\_API\_KEY')):   * This line retrieves the API key **for** Pinecone **from** the Colab environment, where it should have been set up earlier (possibly as a secret or environment variable). **It assumes that** the **key is stored** under the name **'PINECONE\_API\_KEY**'. * By using userdata.get(), it fetches the API key to authenticate with the Pinecone service. |

**4th:**

|  |
| --- |
| **Pinecone Initialization**-------🡪(pc = Pinecone(api\_key=pinecone\_api\_key)):   * This creates an instance of the Pinecone class, passing in the API key retrieved earlier. This instance (pc) will be used to interact with the Pinecone vector database for tasks like inserting, querying, and managing vector data. |

|  |
| --- |
| **Summary:**   * This code sets up the Pinecone client in a Google Colab environment by first retrieving an API key from the environment (using userdata.get) and then initializing the Pinecone client with this key. After initialization, pc is ready to interact with the Pinecone vector database. * For this code to run correctly in Colab, the PINECONE\_API\_KEY must be set in the environment beforehand, which you can do either manually or by uploading it as a secret or using a configuration method within the notebook. |

**Step-3:**

import time

index\_name = "online-rag-project" # change if desired

pc.create\_index(

name=index\_name,

dimension=768,

metric="cosine",

spec=ServerlessSpec(cloud="aws", region="us-east-1"),

)

index = pc.Index(index\_name)

|  |
| --- |
| The Python cod is **working with** a vector database **using** Pinecone, a managed service designed to support vector search operations. **It creates and interacts with** a vector index **for** similarity search (e.g., for natural language processing tasks). **Let's break it down step by step:** |

**1st:**

|  |
| --- |
| **import time:**  This imports the **time module**, but in this specific code snippet, it's not used directly. The time module is **typically used for dealing with time-related tasks**, like delays or performance measurement, **but it's not serving a purpose here.** |

Note: Sir Jnaid removed this part of code after a question of one student.

**2nd:**

|  |
| --- |
| **index\_name = "online-rag-project":**  This is defining the name of the index as "online-rag-project". You can change this to any name you prefer. The index name is a unique identifier **for** the Pinecone index **that will store the vectors**. |

**3rd:**

|  |  |
| --- | --- |
| **Creating an Index**:   |  | | --- | | pc.create\_index(  name=index\_name,  dimension=768,  metric="cosine",  spec=ServerlessSpec(cloud="aws", region="us-east-1"),  ) |   **pc.create\_index(...):**  This is calling the **create\_index method** from the Pinecone client (pc). **It's used to create a new index** in Pinecone. **The parameters passed into this function are**:   * **name=index\_name:** The name of the index to be created, which in this case is "online-rag-project". * **dimension=768:** The dimensionality of the vectors to be stored in the index. In this case, each vector has 768 dimensions, which might correspond to the size of the embeddings from a model like OpenAI's GPT or other transformer models. * **metric="cosine":** This defines the similarity metric to be used in the index. "cosine" indicates that cosine similarity will be used to measure how similar two vectors are to each other. * **spec=ServerlessSpec(cloud="aws", region="us-east-1"):** This defines the cloud setup for the index. Specifically, it uses Pinecone's serverless architecture on AWS in the us-east-1 region. This specifies the cloud provider, region, and the serverless nature of the setup (scaling without needing to manage servers manually). |

**4th:**

|  |
| --- |
| **index = pc.Index(index\_name):**  This line initializes an instance of the index with the specified name **(index\_name)**. The **pc.Index function** connects to the created index so that you can interact with it (e.g., insert, query, or update vectors). |

**Step-4:**

from langchain\_google\_genai import GoogleGenerativeAIEmbeddings

import os

os.environ["GOOGLE\_API\_KEY"]=userdata.get('GOOGLE\_API\_KEY')

embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The **code using** the **GoogleGenerativeAIEmbeddings** class **from** the **langchain\_google\_genai module** to generate embeddings **with** a model offered by Google.  **Let's break it down step by step:**  **1st:**  **Importing necessary libraries:**   |  | | --- | | from langchain\_google\_genai import GoogleGenerativeAIEmbeddings  import os |  * **GoogleGenerativeAIEmbeddings:** This is a class from the langchain\_google\_genai module. It is likely used for generating embeddings, which are numerical representations of text that capture semantic meaning. Embeddings are **commonly used in** natural language processing tasks **like** search, classification, and similarity matching. * **os:** This module is used to interact with the operating system, and **in this context**, it's used to set environment variables.   **2nd:**  **Setting the Google API key**:   |  | | --- | | os.environ["GOOGLE\_API\_KEY"] = userdata.get('GOOGLE\_API\_KEY') |  * This sets an environment variable GOOGLE\_API\_KEY using a value fetched from userdata. This API key is necessary to authenticate the application and allow access to Google's services (such as Google Cloud APIs). * userdata.get('GOOGLE\_API\_KEY') **assumes that** userdata is some dictionary or configuration object that contains your API key.   **3rd:**  **Creating the embeddings object**:   |  | | --- | | embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001") |  * This line **initializes** the GoogleGenerativeAIEmbeddings **class with a model** identifier ("models/embedding-001") which likely **refers to a specific embedding model** hosted by Google or a Google service. * The embeddings object **can now be used to generate** embeddings for text data.  |  | | --- | | **Summary,** the code sets up the environment with a Google API key, initializes an embedding model, and can be used to generate embeddings for text data using Google's generative AI services. | |

**Step-5:**

vector = embeddings.embed\_query("We are building a RAG text")

|  |  |
| --- | --- |
| The code vector = embeddings.embed\_query("We are building a RAG text") appears to be related to embedding **a text query** ("We are building a RAG text") **into a vector** **using** **an embedding model. Let's break it down step by step:**  **1st:**  **embeddings.embed\_query:**   * embeddings likely refers to an object or module responsible for embedding operations. In machine learning, especially in natural language processing (NLP), embeddings are mathematical representations of words, sentences, or entire documents. * The method .embed\_query() is probably used to generate a vector (a list of numbers) representing the semantic meaning of the input text query.   **2nd:**  **"We are building a RAG text":**   * This is the input query. In this case, the text appears to refer to "building a RAG text." RAG could refer to a Retrieval-Augmented Generation model or another context-specific meaning (such as "RAG" being an acronym for something else in a specific domain).   **3rd:**  **vecto**r**:**   * The result of embed\_query is stored in the vector variable. This vector is a numerical representation of the input text. The length and values of this vector depend on the embedding model used. It essentially encodes the semantic meaning of the sentence "We are building a RAG text."  |  | | --- | | **Use of the Vector:**  The generated vector can be used for various purposes **like** semantic search, similarity comparison, **or** as input to other machine learning models.  **In RAG** (Retrieval-Augmented Generation) **models**, the vector might be used to **retrieve relevant documents** or passages from a knowledge base **to enhance the model's ability** to generate contextually **relevant responses**. | |

**Step-6:**

from langchain\_pinecone import PineconeVectorStore

vector\_store = PineconeVectorStore(index=index, embedding=embeddings)

|  |  |
| --- | --- |
| The **code is using** the langchain\_pinecone **module** to **interact with Pinecone**, a vector database **used for** storing and retrieving **vector embeddings efficiently**. **Let's break it down step by step:**  **1st:**  **from langchain\_pinecone import PineconeVectorStore:**  This line imports the PineconeVectorStore class from the langchain\_pinecone module, which is part of LangChain, a framework for building applications with LLMs (Large Language Models). The PineconeVectorStore class is used to interface with Pinecone, allowing you to store, search, and manage vector data (i.e., embeddings of documents or text).  **2nd:**  **vector\_store = PineconeVectorStore(index=index, embedding=embeddings):**  This line creates an instance of the PineconeVectorStore class. Here:  index: This represents a Pinecone index, which is a pre-configured data structure that Pinecone uses to store vectors. It is typically initialized elsewhere in your code or configuration.  **3rd:**  **embedding:**  This is the embedding model or method used to convert text into vectors. This could be a model from OpenAI, HuggingFace, or another source that produces high-dimensional vectors representing semantic information about your text.   |  | | --- | | **Summary :** Essentially,this code initializes a connection between LangChain and Pinecone, allowing you to use Pinecone as a vector store to manage embeddings for tasks like similarity search, document retrieval, and more. | |

**Step-7:**

from uuid import uuid4

from langchain\_core.documents import Document

document\_1 = Document(

page\_content="I had chocalate chip pancakes and scrambled eggs for breakfast this morning.",

metadata={"source": "tweet"},

)

document\_2 = Document(

page\_content="The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees.",

metadata={"source": "news"},

)

documents = [

document\_1,

document\_2,

]

|  |  |
| --- | --- |
| The code demonstrates how to create documents **using** langchain\_core and populate them **with** content and metadata. **Let's break it down step by step:**  **1st:**  **Importing** uuid4 **from** uuid**:**  This imports a function used for generating random UUIDs (universally unique identifiers). However, the function is not actually used in this snippet, so it might be included for potential future use, or it was part of a larger codebase.  **2nd:**  **Importing** Document from langchain\_core.documents**:**  The Document class is imported from the langchain\_core.documents module. This class is used to create documents that consist of page\_content (the text of the document) and metadata (a dictionary holding additional information about the document).  **3rd:**  **Creating document\_1:** The first Document is created with the following properties:  page\_content: "I had chocolate chip pancakes and scrambled eggs for breakfast this morning."  metadata: The metadata for this document is specified as {"source": "tweet"}, which may indicate that this text was sourced from a tweet.  **Creating document\_2:** The second Document is created with the following properties:  page\_content: "The weather forecast for tomorrow is cloudy and overcast, with a high of 62 degrees."  metadata: The metadata for this document is specified as {"source": "news"}, indicating the text comes from a news source.  **4th:**  **Storing Documents in a List:** Both document\_1 and document\_2 are added to a list named documents.   |  | | --- | | Purpose:  This code is part of a larger program that processes or analyzes text documents. By storing these documents in a list, the program can manipulate, analyze, or index the documents, using the content and metadata for various purposes, such as natural language processing (NLP), document retrieval, or indexing in a system like langchain. The metadata can be used to categorize, tag, or filter the documents. | |

**Step-8:**

len(documents)

|  |  |
| --- | --- |
| The len() function is a built-in Python function used to get the length of various objects. The behavior depends on the type of object passed as an argument:  In Python, len(documents) is a function call that returns the number of items in the object documents.   |  | | --- | | # List example  documents = ["doc1", "doc2", "doc3"]  print(len(documents)) # Output: 3  # String example  documents = "This is a document."  print(len(documents)) # Output: 19  # Dictionary example  documents = {"doc1": "content1", "doc2": "content2"}  print(len(documents)) # Output: 2  # Set example  documents = {"doc1", "doc2", "doc3"}  print(len(documents)) # Output: 3 | |

**Step-9:**

uuids = [str(uuid4()) for \_ in range(len(documents))]

vector\_store.add\_documents(documents=documents, ids=uuids)

|  |  |  |  |
| --- | --- | --- | --- |
| The code snippet you provided is using the uuid4 function to generate unique identifiers (UUIDs) and then using them to add documents to a vector store.  **Let's break it down step by step:**  **1st:**  **UUID Generation:**   |  | | --- | | uuids = [str(uuid4()) for \_ in range(len(documents))] |  * **uuid4()** generates a random UUID (Universally Unique Identifier) using random numbers. These UUIDs are typically used to uniquely identify items across distributed systems. * **The str(uuid4())** converts the UUID object to a string representation. * The list comprehension **[str(uuid4()) for \_ in range(len(documents))]** creates a list of UUID strings. The number of UUIDs generated is equal to the number of documents (i.e., len(documents)).   So, if you have 10 documents, this will generate 10 unique UUIDs, each in string form.  **2nd:**  **Adding Documents to a Vector Store:**   |  | | --- | | vector\_store.add\_documents(documents=documents, ids=uuids) |  * **vector\_store.add\_documents()** is a function that adds the documents into a vector store, which is a data structure used to store and search vector representations of documents, often used in machine learning or natural language processing tasks. * **documents is assumed to be a list** or collection of documents (it could be text, data points, or other entities). * **ids=uuids** passes the list of UUIDs that you just generated as the unique identifiers for each document. This means each document will be indexed with its corresponding UUID, allowing you to reference, retrieve, or update the document by this unique ID later.  |  | | --- | | **Summary:**   * **Purpose:** This code assigns unique IDs (UUIDs) to each document and then adds them to a vector store for future use (e.g., retrieval, indexing). * **UUIDs:** These unique identifiers help to manage documents in the vector store without risking any duplication or confusion. * **Vector Store**: This could be a data structure or a database that stores vector representations of documents, typically used in similarity searches or machine learning models. | |

**Step-10:**

results = vector\_store.similarity\_search(

"LangChain provides abstractions to make working with LLMs easy",

#k=2,

#filter={"source": "tweet"},

)

for res in results:

print(f"\* {res.page\_content} [{res.metadata}]")

|  |  |  |
| --- | --- | --- |
| This code snippet is performing a similarity search in a vector store using the LangChain framework. **Let's break it down step by step:**  **1st:**  **vector\_store.similarity\_search(...):**   * This method performs a similarity search on a vector store, which is a data structure for storing and retrieving vectors. Vectors are numerical representations of text (e.g., embeddings generated from a language model). * It finds the most similar entries in the vector store to the query provided.   **2nd:**  **Query:**   * "LangChain provides abstractions to make working with LLMs easy" is the text query being searched for in the vector store. The vector store will compare this query to the stored vectors and return the most similar ones.   **3rd:**  **k=2:**   * This specifies the number of results to return. In this case, the function will return the 2 most similar entries to the query.   **4th:**  **filter={"source": "tweet"}:**   * This filter ensures that only results where the metadata source is equal to "tweet" are considered in the search. This could be useful if the vector store contains data from different sources, and you only want to focus on the "tweet" source.   **5th:**  **Results Iteration:**   * The **for res in results:** loop iterates over the results returned by the similarity search. * **res.page\_content** represents the content of the result (likely a text or document) that matched the query. * **res.metadata** holds additional information related to the result, which might include source details, date, or other metadata associated with the text.   **6th:**  **print(f"\* {res.page\_content} [{res.metadata}]"):**  For each result, this line prints the page content followed by the metadata of the result in the format:   |  | | --- | | \* [Page content here] [Metadata here]  **Example:**  If the vector\_store contains tweets, the output might look like:  \* "LangChain is revolutionizing how we work with LLMs." [{ 'source': 'tweet', 'author': 'user123', 'date': '2025-01-04' }]  \* "Working with LLMs has never been easier thanks to LangChain." [{ 'source': 'tweet', 'author': 'user456', 'date': '2025-01-03' }] |  |  | | --- | | Summary, this code is querying a vector store for the two most similar tweets to a specific text and printing the content of those tweets along with associated metadata. | |

**Step-11:**

results = vector\_store.similarity\_search\_with\_score(

"Will it be hot tomorrow?", k=1, filter={"source": "news"}

)

for res, score in results:

print(f"\* [SIM={score:3f}] {res.page\_content} [{res.metadata}]")

|  |  |
| --- | --- |
| The code is **using** a vector store **to perform** a similarity search **with scores**. Here's a breakdown of what each part does: **Let's break it down step by step:**  **1st:**  vector\_store.similarity\_search\_with\_score: This method is likely querying a vector database (e.g., Faiss, Pinecone, or another vector store). It searches for the most similar document to the query you provided (in this case, the question "Will it be hot tomorrow?").   * **Query:** The search is looking for documents related to the question "Will it be hot tomorrow?". * **k=1:** This specifies that the search will return the top 1 most similar result. * **filter=**{"source": "news"}: This adds a filter to only return documents where the source metadata field is labeled as "news". It ensures that only news-related content is considered.   **2nd:**  **Iterating through the results**: The loop goes through each of the results returned by the similarity search.   * for res, score in results: Each result (res) and its corresponding similarity score (score) are extracted in the loop.   **3rd:**  **Printing the result:** f"\* [SIM={score:3f}] {res.page\_content} [{res.metadata}]: This is formatting the output for each result. It prints:   * The similarity score (score:3f formats the score to 3 decimal places). * The content of the document (res.page\_content). * The metadata of the document (res.metadata), such as the source, date, or other contextual information.  |  | | --- | | **Summary**, this code performs a similarity search in a vector store to find documents related to the query, "Will it be hot tomorrow?", filters them by the "news" source, and then prints out the top result with its similarity score, content, and metadata. | |

**Step-12:**

def answer\_to\_user(query: str):

# Vector Search

vector\_results = vector\_store.similarity\_search(query, k=2)

print(len(vector\_results))

# TODO: Pass to Model vector\_results + User Query

final\_answer = llm.invoke(f"ANSWER THIS USER QUERY: {query}. Here are some references to answer: {vector\_results}")

return final\_answer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| The code defines a function **answer\_to\_user(query: str)** that processes a user's query in the context of retrieving information **from** a vector store and then **using** a language model (likely some form of large language model or LLM) to generate an answer based on the retrieved information. **Let's break it down step by step:**  **1st:**  **Vector Search**   |  | | --- | | vector\_results = vector\_store.similarity\_search(query, k=2) |  * This line performs a similarity search on the **vector\_store** using the user's query. * **vector\_store.similarity\_search(query, k=2)** performs a search to find the 2 most similar items (or documents) to the input **query** from the vector store. The results are stored in **vector\_results**. * The **vector\_store** is likely a data structure that stores vectors (numerical representations) of documents or information that can be used for similarity comparisons**.**   **2nd:**  **Print Length of Results**   |  | | --- | | print(len(vector\_results)) |  * This line prints the number of results returned by the similarity search.   **3rd:**  **Pass Results and Query to Model**   |  | | --- | | final\_answer = llm.invoke(f"ANSWER THIS USER QUERY: {query}. Here are some references to answer: {vector\_results}") |  * The function then constructs a prompt for a language model (llm) to generate an answer. The prompt includes both the original user query and the results from the similarity search (vector\_results), which might include relevant information or context. * The llm.invoke(...) method is assumed to invoke the language model with the formatted string, which will process the query and the reference results to generate a final answer.   **4th:**  **Return the Final Answer**   |  | | --- | | return final\_answer |  * Finally, the function returns the generated answer from the language model based on the query and the context **(vector\_results)**  |  | | --- | | **Overall Purpose:**   * The function is intended to take a user's query, retrieve similar content or references from a vector store, and then use a language model to formulate an answer, potentially enriched by the retrieved references. This could be useful in contexts like question-answering systems or chatbots that need to pull from a database of knowledge to provide accurate and relevant responses. |  |  | | --- | | **Potential TODO Comment:**   * There is a TODO comment in the code indicating that at some point, the code needs to pass the vector\_results and the query to the model in a more structured way, or possibly do more processing before invoking the model. | |

**Step-13:**

from langchain\_google\_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(

model="gemini-1.5-flash",

temperature=0,

max\_tokens=None,

timeout=None,

max\_retries=2,

# other params...

)

|  |  |  |
| --- | --- | --- |
| The code using **Langchain's ChatGoogleGenerativeAI** for interacting with **Google's generative AI model,** specifically the gemini-1.5-flash model.  **Let's break it down step by step:**  **1st:**  **from langchain\_google\_genai import ChatGoogleGenerativeAI:**   * This imports the ChatGoogleGenerativeAI class from the langchain\_google\_genai module, which is part of LangChain's integration with Google’s generative AI models. This class provides an interface for interacting with Google's models, like **Gemini.**   **2nd:**  **Creating an instance of ChatGoogleGenerativeAI:**   * llm = ChatGoogleGenerativeAI(...) creates an object of the ChatGoogleGenerativeAI class. This object will allow you to interact with the Google generative model using various parameters and configuration options   **3rd:**  **Parameters being set:**   * model="gemini-1.5-flash": Specifies the specific version of the Google model to use. In this case, gemini-1.5-flash is chosen, which could be a specific variant of Google's Gemini AI (likely a fast or optimized version). * temperature=0: Controls the randomness of the model's responses. A temperature of 0 makes the model's output deterministic (less creative), while higher values (e.g., 0.7 to 1) make it more creative and random. * max\_tokens=None: This indicates that there is no explicit limit on the number of tokens the model can generate. If you set a number here (e.g., 1000), it would limit the number of tokens generated by the model in one response. * timeout=None: No timeout is set, meaning the model can run indefinitely. If you set a timeout, it would limit how long the system waits for a response. * max\_retries=2: Specifies that the system will try to make up to 2 retries in case of failures or timeouts while interacting with the model.   **4th:**  **Other parameters (commented out):**  There may be other parameters that could be set based on specific needs, like controlling the response length, providing custom prompts, etc.   |  | | --- | | **Langchain Integration:**  LangChain is a framework for building applications that use language models. **It simplifies integrating different generative AI models**, like OpenAI, Hugging Face, and Google, into conversational agents or other use cases. |  |  | | --- | | **Summary,** this code defines a configuration for interacting with Google's gemini-1.5-flash model using LangChain, with specific settings for temperature, retries, and token limits. | |

**Step-14:**

answer\_to\_user("LangChain provides abstractions to make working with LLMs easy")

|  |
| --- |
| answer\_to\_user(" ") is a function call in Python language, possibly  Python, where answer\_to\_user is a function name and in round bracket (" ") is passed as an argument.  **Function name** = answer\_to\_user  **Argument** = ("LangChain provides abstractions to make working with LLMs easy") |