Overview

Retrieval augmented generation (RAG) has become a popular paradigm for enabling LLMs to access external data and also as a mechanism for grounding to mitigate against hallucinations.

In this notebook, you will learn how to perform RAG where you will perform Q&A over a document filled with both text and images.

Gemini

Gemini is a family of generative AI models developed by Google DeepMind that is designed for multimodal use cases. The Gemini API gives you access to the Gemini 1.0 Pro Vision and Gemini 1.0 Pro models.

Comparing text-based and multimodal RAG

Multimodal RAG offers several advantages over text-based RAG:

1. **Enhanced knowledge access:** Multimodal RAG can access and process both textual and visual information, providing a richer and more comprehensive knowledge base for the LLM.
2. **Improved reasoning capabilities:** By incorporating visual cues, multimodal RAG can make better informed inferences across different types of data modalities.

This notebook shows you how to use multimodal RAG with Vertex AI Gemini API, [text embeddings] to build a question answering system for a PDF document.

Objectives

This notebook provides a guide to building a questions answering system using multimodal retrieval augmented generation (RAG).

You will complete the following tasks:

1. Extract data from documents containing both text and images using Gemini Vision Pro, and generate embeddings of the data, store it in vector store
2. Search the vector store with text queries to find similar text data
3. Using Text data as context, generate answer to the user query using Gemini Pro Model.

Getting Started

Install Vertex AI SDK and other required packages

```
!pip install --upgrade --quiet pymupdf langchain gradio google-cloud-aiplatform lan
```

Restart runtime

To use the newly installed packages in this Jupyter runtime, you must restart the runtime. You can do this by running the cell below, which restarts the current kernel.

The restart might take a minute or longer. After its restarted, continue to the next step.

```
import IPython

app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

```
{'status': 'ok', 'restart': True}
```

⚠️ **Wait for the kernel to finish restarting before you continue.** ⚠️

Authenticate your notebook environment (Colab only)

If you are running this notebook on Google Colab, run the cell below to authenticate your environment.

This step is not required if you are using [Vertex AI Workbench](#).

```
import sys

# Additional authentication is required for Google Colab
if "google.colab" in sys.modules:
    # Authenticate user to Google Cloud
    from google.colab import auth

    auth.authenticate_user()
```

Define Google Cloud project information and initialize Vertex AI

To get started using Vertex AI, you must have an existing Google Cloud project and enable the Vertex AI API.

Learn more about setting up a project and a development environment.

```
# Define project information
PROJECT_ID = "genai-projects-429119"  # @
LOCATION = "us-east1"  # @param {type:"st

# Initialize Vertex AI
import vertexai

vertexai.init(project=PROJECT_ID, locatio
```

PROJECT_ID: " genai-projects-42' "

LOCATION: " us-east1 "

```
!pip install langchain_community
```

Show hidden output

Import libraries

Let's start by importing the libraries that we will need for this tutorial

```python
# File system operations and displaying images
import os

# Import utility functions for timing and file handling
import time

# Libraries for downloading files, data manipulation, and creating a user interface
import uuid
from datetime import datetime

import fitz
import gradio as gr
import pandas as pd

# Initialize Vertex AI libraries for working with generative models
from google.cloud import aiplatform
from PIL import Image as PIL_Image
from vertexai.generative_models import GenerativeModel, Image
from vertexai.language_models import TextEmbeddingModel

# Print Vertex AI SDK version
print(f"Vertex AI SDK version: {aiplatform.__version__}")

# Import LangChain components
import langchain

print(f"LangChain version: {langchain.__version__}")
from langchain.text_splitter import CharacterTextSplitter
from langchain_community.document_loaders import DataFrameLoader
```

```
Vertex AI SDK version: 1.59.0
LangChain version: 0.2.7
```

Initializing Gemini Vision Pro and Text Embedding models

```python
# Loading Gemini Pro Vision Model
multimodal_model = GenerativeModel("gemini-1.0-pro-vision")

# Initializing embedding model
text_embedding_model = TextEmbeddingModel.from_pretrained("textembedding-gecko@003"

# Loading Gemini Pro Model
model = GenerativeModel("gemini-1.0-pro")
```

Download from internet a sample PDF file and default image to be shown when no results are found

[Skip this step if you have uploaded your PDF file]

This document describes the importance of stable power grids in Japan, highlighting the recent failure of a generator step-up transformer at the Nakoso Power Station and the rapid restoration response undertaken to maintain power supply stability.

```
!wget https://www.hitachi.com/rev/archive/2023/r2023_04/pdf/04a02.pdf
!wget https://img.freepik.com/free-vector/hand-drawn-no-data-illustration_23-215069(

# Create an "Images" directory if it doesn't exist
Image_Path = "./Images/"
if not os.path.exists(Image_Path):
    os.makedirs(Image_Path)

!mv hand-drawn-no-data-illustration_23-2150696455.jpg {Image_Path}/blank.jpg
```

Show hidden output

Split PDF to images and extract data using Gemini Vision Pro

This module processes a set of images, extracting text and tabular data using a multimodal model (Gemini Vision Pro). It handles potential errors, stores the extracted information in a DataFrame, and saves the results to a CSV file.

```
# Run the following code for each file
PDF_FILENAME = "04a02.pdf"  # Replace with your filename
```

```python
# To get better resolution
zoom_x = 2.0  # horizontal zoom
zoom_y = 2.0  # vertical zoom
mat = fitz.Matrix(zoom_x, zoom_y)  # zoom factor 2 in each dimension

doc = fitz.open(PDF_FILENAME)  # open document
for page in doc:  # iterate through the pages
    pix = page.get_pixmap(matrix=mat)  # render page to an image
    outpath = f"./Images/{PDF_FILENAME}_{page.number}.jpg"
    pix.save(outpath)  # store image as a PNG

# Define the path where images are located
image_names = os.listdir(Image_Path)
Max_images = len(image_names)

# Create empty lists to store image information
page_source = []
page_content = []
page_id = []

p_id = 0  # Initialize image ID counter
rest_count = 0  # Initialize counter for error handling

while p_id < Max_images:
    try:
        # Construct the full path to the current image
        image_path = Image_Path + image_names[p_id]

        # Load the image
        image = Image.load_from_file(image_path)

        # Generate prompts for text and table extraction
        prompt_text = "Extract all text content in the image"
        prompt_table = (
            "Detect table in this image. Extract content maintaining the structure"
        )

        # Extract text using your multimodal model
        contents = [image, prompt_text]
        response = multimodal_model.generate_content(contents)
        text_content = response.text

        # Extract table using your multimodal model
        contents = [image, prompt_table]
        response = multimodal_model.generate_content(contents)
        table_content = response.text

        # Log progress and store results
        print(f"processed image no: {p_id}")
        page_source.append(image_path)
        page_content.append(text_content + "\n" + table_content)
```

```
        page_id.append(p_id)
        p_id += 1

    except Exception as err:
        # Handle errors during processing
        print(err)
        print("Taking Some Rest")
        time.sleep(1)  # Pause execution for 1 second
        rest_count += 1
        if rest_count == 5:  # Limit consecutive error handling
            rest_count = 0
            print(f"Cannot process image no: {image_path}")
            p_id += 1  # Move to the next image

# Create a DataFrame to store extracted information
df = pd.DataFrame(
    {"page_id": page_id, "page_source": page_source, "page_content": page_content}
)
del page_id, page_source, page_content  # Conserve memory
df.head()  # Preview the DataFrame
```

Show hidden output

Next steps: Generate code with df View recommended plots

Generate Text Embeddings

Leverage a powerful language model textembedding-gecko to generate rich text embeddings that helps us find relevant information from a dataset.

```python
def generate_text_embedding(text) -> list:
    """Text embedding with a Large Language Model."""
    embeddings = text_embedding_model.get_embeddings([text])
    vector = embeddings[0].values
    return vector


# Create a DataFrameLoader to prepare data for LangChain
loader = DataFrameLoader(df, page_content_column="page_content")

# Load documents from the 'page_content' column of your DataFrame
documents = loader.load()

# Log the number of documents loaded
print(f"# of documents loaded (pre-chunking) = {len(documents)}")

# Create a text splitter to divide documents into smaller chunks
text_splitter = CharacterTextSplitter(
    chunk_size=10000,  # Target size of approximately 10000 characters per chunk
    chunk_overlap=200,  # overlap between chunks
)

# Split the loaded documents
doc_splits = text_splitter.split_documents(documents)

# Add a 'chunk' ID to each document split's metadata for tracking
for idx, split in enumerate(doc_splits):
    split.metadata["chunk"] = idx

# Log the number of documents after splitting
print(f"# of documents = {len(doc_splits)}")

texts = [doc.page_content for doc in doc_splits]
text_embeddings_list = []
id_list = []
page_source_list = []
for doc in doc_splits:
    id = uuid.uuid4()
    text_embeddings_list.append(generate_text_embedding(doc.page_content))
    id_list.append(str(id))
    page_source_list.append(doc.metadata["page_source"])
    time.sleep(1)  # So that we don't run into Quota Issue

# Creating a dataframe of ID, embeddings, page_source and text
embedding_df = pd.DataFrame(
    {
        "id": id_list,
        "embedding": text_embeddings_list,
        "page_source": page_source_list,
        "text": texts,
    }
```

```
)
embedding_df.head()
```

Show hidden output

Next steps: Generate code with `embedding_df` View recommended plots

Creating Vertex AI: Vector Search

The code configures and deploys a vector search index on Google Cloud, making it ready to store and search through embeddings.

Embedding size : The number of values used to represent a piece of text in vector form. Larger dimensions mean a denser and potentially more expressive representation.

Dimensions vs. Latency

- Search: Higher-dimensional embeddings can make vector similarity searches slower, especially in large databases.
- Computation: Calculations with larger vectors generally take more time during model training and inference.

```
VECTOR_SEARCH_REGION = "us-central1"
VECTOR_SEARCH_INDEX_NAME = f"{PROJECT_ID}-vector-search-index-ht"
VECTOR_SEARCH_EMBEDDING_DIR = f"{PROJECT_ID}-vector-search-bucket-ht"
VECTOR_SEARCH_DIMENSIONS = 768
```

Save the embeddings in a JSON file

To load the embeddings to Vector Search, we need to save them in JSON files with JSONL format. See more information in the docs at [Input data format and structure](#).

First, export the `id` and `embedding` columns from the DataFrame in JSONL format, and save it.

Then, create a new Cloud Storage bucket and copy the file to it.

```
# save id and embedding as a json file
jsonl_string = embedding_df[["id", "embedding"]].to_json(orient="records", lines=Tru
with open("data.json", "w") as f:
    f.write(jsonl_string)

# show the first few lines of the json file
! head -n 3 data.json
```

Show hidden output

```
# Generates a unique ID for session
UID = datetime.now().strftime("%m%d%H%M")

# Creates a GCS bucket
BUCKET_URI = f"gs://{VECTOR_SEARCH_EMBEDDING_DIR}-{UID}"
! gsutil mb -l $LOCATION -p {PROJECT_ID} {BUCKET_URI}
! gsutil cp data.json {BUCKET_URI}
```

Show hidden output

Create an Index

Now it's ready to load the embeddings to Vector Search. Its APIs are available under the [aiplatform] package of the SDK.

Create an [MatchingEngineIndex] with its `create_tree_ah_index` function (Matching Engine is the previous name of Vector Search).

```
# create index
my_index = aiplatform.MatchingEngineIndex.create_tree_ah_index(
    display_name=f"{VECTOR_SEARCH_INDEX_NAME}",
    contents_delta_uri=BUCKET_URI,
    dimensions=768,
    approximate_neighbors_count=20,
    distance_measure_type="DOT_PRODUCT_DISTANCE",
)
```

Show hidden output

By calling the `create_tree_ah_index` function, it starts building an Index. This will take under a few minutes if the dataset is small, otherwise about 50 minutes or more depending on the size of the dataset. You can check status of the index creation on the Vector Search Console > INDEXES tab.

The parameters for creating index

- `contents_delta_uri`: The URI of Cloud Storage directory where you stored the embedding JSON files
- `dimensions`: Dimension size of each embedding. In this case, it is 768 as we are using the embeddings from the Text Embeddings API.

- `approximate_neighbors_count` : how many similar items we want to retrieve in typical cases
- `distance_measure_type` : what metrics to measure distance/similarity between embeddings. In this case it's `DOT_PRODUCT_DISTANCE`

See the document for more details on creating Index and the parameters.

Create Index Endpoint and deploy the Index

To use the Index, you need to create an Index Endpoint. It works as a server instance accepting query requests for your Index.

```
# create IndexEndpoint
my_index_endpoint = aiplatform.MatchingEngineIndexEndpoint.create(
    display_name=f"{VECTOR_SEARCH_INDEX_NAME}",
    public_endpoint_enabled=True,
)
print(my_index_endpoint)
```

Show hidden output

This tutorial utilizes a Public Endpoint and does not support Virtual Private Cloud (VPC). Unless you have a specific requirement for VPC, we recommend using a Public Endpoint. Despite the term "public" in its name, it does not imply open access to the public internet. Rather, it functions like other endpoints in Vertex AI services, which are secured by default through IAM. Without explicit IAM permissions, as we have previously established, no one can access the endpoint.

With the Index Endpoint, deploy the Index by specifying an unique deployed index ID.

```
DEPLOYED_INDEX_NAME = VECTOR_SEARCH_INDEX_NAME.replace(
    "-", "_"
)  # Can't have - in deployment name, only alphanumeric and _ allowed
DEPLOYED_INDEX_ID = f"{DEPLOYED_INDEX_NAME}_{UID}"
# deploy the Index to the Index Endpoint
my_index_endpoint.deploy_index(index=my_index, deployed_index_id=DEPLOYED_INDEX_ID)
```

Show hidden output

Next steps: Explain error

If it is the first time to deploy an Index to an Index Endpoint, it will take around 25 minutes to automatically build and initiate the backend for it. After the first deployment, it will finish in seconds. To see the status of the index deployment, open the Vector Search Console > INDEX ENDPOINTS tab and click the Index Endpoint.

Ask Questions to the PDF

This code snippet establishes a question-answering (QA) system. It leverages a vector search engine to find relevant information from a dataset and then uses the 'gemini-pro' LLM model to generate and refine the final answer to a user's query.

```
def Test_LLM_Response(txt):
    """
    Determines whether a given text response generated by an LLM indicates a lack o

    Args:
        txt (str): The text response generated by the LLM.

    Returns:
        bool: True if the LLM's response suggests it was able to generate a meaning
              False if the response indicates it could not find relevant informatio

    This function works by presenting a formatted classification prompt to the LLM
    The prompt includes the original text and specific categories indicating whethe
    The function analyzes the LLM's classification output to make the determination
    """

    classification_prompt = f""" Classify the text as one of the following categori
        -Information Present
        -Information Not Present
        Text=The provided context does not contain information.
        Category:Information Not Present
        Text=I cannot answer this question from the provided context.
        Category:Information Not Present
        Text:{txt}
        Category:"""
    classification_response = model.generate_content(classification_prompt).text

    if "Not Present" in classification_response:
        return False  # Indicates that the LLM couldn't provide an answer
    else:
        return True  # Suggests the LLM generated a meaningful response


def get_prompt_text(question, context):
    """
    Generates a formatted prompt string suitable for a language model, combining th

    Args:
        question (str): The user's original question.
        context (str): The relevant text to be used as context for the answer.

    Returns:
        str: A formatted prompt string with placeholders for the question and conte
    """
    prompt = """
      Answer the question using the context below. Respond with only from the text
      Question: {question}
      Context : {context}
      """.format(
        question=question, context=context
    )
```

```
    return prompt


def get_answer(query):
    """
    Retrieves an answer to a provided query using multimodal retrieval augmented ge

    This function leverages a vector search system to find relevant text documents 
    pre-indexed store of multimodal data. Then, it uses a large language model (LLM
    an answer, using the retrieved documents as context.

    Args:
        query (str): The user's original query.

    Returns:
        dict: A dictionary containing the following keys:
            * 'result' (str): The LLM-generated answer.
            * 'neighbor_index' (int): The index of the most relevant document used 
                                      (for fetching image path).

    Raises:
        RuntimeError: If no valid answer could be generated within the specified se
    """

    neighbor_index = 0  # Initialize index for tracking the most relevant document
    answer_found_flag = 0  # Flag to signal if an acceptable answer is found
    result = ""  # Initialize the answer string
    # Use a default image if the reference is not found
    page_source = "./Images/blank.jpg"  # Initialize the blank image
    query_embeddings = generate_text_embedding(
        query
    )  # Generate embeddings for the query

    response = my_index_endpoint.find_neighbors(
        deployed_index_id=DEPLOYED_INDEX_ID,
        queries=[query_embeddings],
        num_neighbors=5,
    )  # Retrieve up to 5 relevant documents from the vector store

    while answer_found_flag == 0 and neighbor_index < 4:
        context = embedding_df[
            embedding_df["id"] == response[0][neighbor_index].id
        ].text.values[
            0
        ]  # Extract text context from the relevant document

        prompt = get_prompt_text(
            query, context
        )  # Create a prompt using the question and context
        result = model.generate_content(prompt).text  # Generate an answer with the
```

```
        if Test_LLM_Response(result):
            answer_found_flag = 1  # Exit loop when getting a valid response
        else:
            neighbor_index += (
                1  # Try the next retrieved document if the answer is unsatisfactory
            )

    if answer_found_flag == 1:
        page_source = embedding_df[
            embedding_df["id"] == response[0][neighbor_index].id
        ].page_source.values[
            0
        ]  # Extract image_path from the relevant document
    return result, page_source


query = (
    "what is the steps of Transformer Manufacturing Flow ?")

result, page_source = get_answer(query)
print(result)
```

Show hidden output

```
result, page_source = get_answer("wh")
print(result)
```

Show hidden output

Ask Questions to the PDF using Gradio UI

this code creates a web-based frontend for your question-answering system, allowing users to easily enter queries and see the results along with relevant images.

```
import gradio as gr
from PIL import Image as PIL_Image
```