Google Cloud – Langchain usage for Prompt and Response

-------------------------------------------------------------------------------------

%%capture --no-stderr

!pip install --quiet langchain chromadb==0.5.3

!pip install langchain-community

!pip install langchain-google-vertexai

!pip install --upgrade --quiet langchain-google-genai

# Restart kernel after installs so that your environment can access the new packages

import IPython

import time

app = IPython.Application.instance()

app.kernel.do\_shutdown(True)

# Define project information

import sys

PROJECT\_ID = "qwiklabs-gcp-00-b8db9c969c7d" # @param {type:"string"} Please set your PROJECT\_ID

LOCATION = "us-central1" # @param {type:"string"}

# if not running on colab, try to get the PROJECT\_ID automatically

if "google.colab" not in sys.modules:

import subprocess

PROJECT\_ID = subprocess.check\_output(

["gcloud", "config", "get-value", "project"], text=True

).strip()

print(f"Your project ID is: {PROJECT\_ID}")

from langchain import PromptTemplate

from langchain import hub

from langchain.docstore.document import Document

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.document\_loaders import WebBaseLoader

from langchain.schema import StrOutputParser

from langchain.schema.prompt\_template import format\_document

from langchain.schema.runnable import RunnablePassthrough

from langchain.vectorstores import Chroma

from langchain\_google\_vertexai import VertexAIEmbeddings

from langchain\_google\_vertexai import VertexAI

from langchain.chains import RetrievalQA, ConversationalRetrievalChain

# Task 1. Use the WebBaseLoader to load documents related to queries

loader = WebBaseLoader("https://blog.google/technology/ai/google-gemini-ai/")

docs = loader.load()

# Task 2. Use the RecursiveCharacterTextSplitter class to split the documents into chunks for embedding

text\_splitter = RecursiveCharacterTextSplitter(

chunk\_size = 800,

chunk\_overlap = 100,

length\_function = len,

)

chunks = text\_splitter.split\_documents(docs)

# Look at the first two chunks

chunks[0:2]

# Task 3. Create vector store using embeddings

from langchain\_google\_genai import GoogleGenerativeAIEmbeddings

gemini\_embeddings = VertexAIEmbeddings(model="text-embedding-004")

# Save to disk

vectorstore = Chroma(embedding\_function=gemini\_embeddings, persist\_directory="./vectorstore")

for chunk in chunks:

vectorstore.add\_documents([chunk])

# Load from disk

vectorstore\_disk = Chroma(

embedding\_function=gemini\_embeddings, # Embedding model

persist\_directory="./vectorstore" # Directory to save the embeddings

)

retriever = vectorstore\_disk.as\_retriever(search\_kwargs={"k": 1})

print(len(retriever.invoke("MMLU")))

# vector store

retriever = vectorstore.as\_retriever(

search\_type="similarity",

search\_kwargs={"k":2} # number of nearest neighbors to retrieve

)

# You can also set temperature, top\_p, top\_k

llm = VertexAI(

model\_name="text-bison",

max\_output\_tokens=1024

)

# q/a chain

qa = RetrievalQA.from\_chain\_type(

llm=llm,

chain\_type="stuff",

retriever=retriever,

return\_source\_documents=True

)

def ask\_question(question: str):

response = qa.invoke({"query": question})

print(f"Response: {response['result']}\n")

citations = {doc.metadata['source'] for doc in response['source\_documents']}

print(f"Citations: {citations}\n")

# uncomment below to print source chunks used

# print(f"Source Chunks Used: {response['source\_documents']}")

def ask\_question(question: str):

response = qa.invoke({"query": question})

print(f"Response: {response['result']}\n")

citations = {doc.metadata['source'] for doc in response['source\_documents']}

print(f"Citations: {citations}\n")

# uncomment below to print source chunks used

# print(f"Source Chunks Used: {response['source\_documents']}")

ask\_question("What is MMLU?")

ask\_question("What is a TPU?")

RAG

A diagram of a diagram

Description automatically generated

A white text on a white background

Description automatically generated

Gemini langchain RAG

%%capture

!pip -q install langchain\_experimental langchain\_core

!pip -q install google-generativeai==0.3.1

!pip -q install google-ai-generativelanguage==0.4.0

!pip -q install langchain-google-genai

!pip -q install wikipedia

!pip -q install docarray

!pip -q install --upgrade protobuf google.protobuf

import os

import google.generativeai as genai

from IPython.display import display

from IPython.display import Markdown

from google.protobuf.empty\_pb2 import Empty

key\_name = !gcloud services api-keys list --filter="gemini-api-key" --format="value(name)"

key\_name = key\_name[0]

api\_key = !gcloud services api-keys get-key-string $key\_name --location="us-central1" --format="value(keyString)"

api\_key = api\_key[0]

os.environ["GOOGLE\_API\_KEY"] = api\_key

genai.configure(api\_key=os.environ["GOOGLE\_API\_KEY"])

models = [m for m in genai.list\_models()] ------------🡪 all models listed

models

# generate text

prompt = 'Who are you and what can you do?'

model = genai.GenerativeModel('gemini-pro')

response = model.generate\_content(prompt)

Markdown(response.candidates[0].content.parts[0].text)

Basic LLM chain

from langchain\_core.messages import HumanMessage

from langchain\_google\_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-pro",

temperature=0.7)

result = llm.invoke("What is a LLM?")

Markdown(result.content)

for chunk in llm.stream("Write a haiku about LLMs."):

print(chunk.content)

print("---")

Basic Multi Chain

from langchain\_google\_genai import ChatGoogleGenerativeAI

from langchain\_google\_genai import GoogleGenerativeAIEmbeddings

from langchain.prompts import ChatPromptTemplate

from langchain.schema.output\_parser import StrOutputParser

model = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.7)

prompt = ChatPromptTemplate.from\_template(

"tell me a short joke about {topic}"

)

output\_parser = StrOutputParser()

chain = prompt | model | output\_parser

chain.invoke({"topic": "machine learning"})

A more complicated RAG

from langchain\_google\_genai import ChatGoogleGenerativeAI

from langchain\_google\_genai import GoogleGenerativeAIEmbeddings

from langchain.vectorstores import DocArrayInMemorySearch

model = ChatGoogleGenerativeAI(model="gemini-pro",

temperature=0.7)

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

from langchain.document\_loaders import WikipediaLoader

# use Wikipedia loader to create some docs to use..

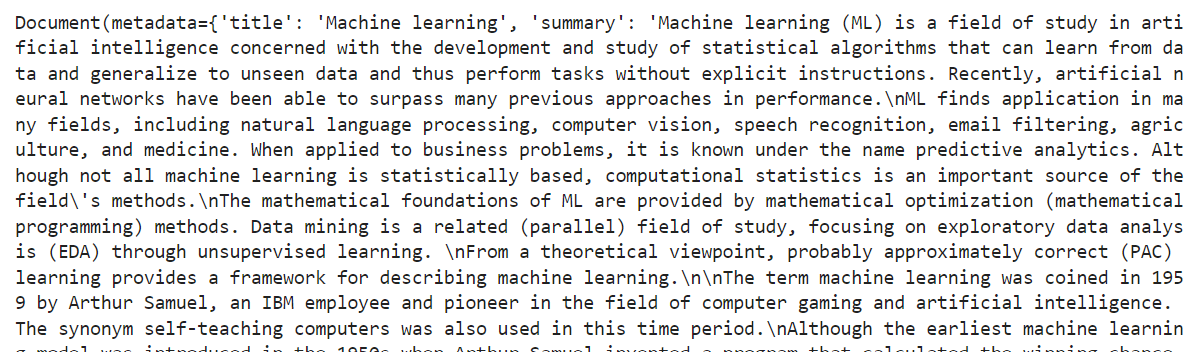
docs = WikipediaLoader(query="Machine Learning", load\_max\_docs=10).load()

docs += WikipediaLoader(query="Deep Learning", load\_max\_docs=10).load()

docs += WikipediaLoader(query="Neural Networks", load\_max\_docs=10).load()

# Take a look at a single document

docs[0]



vectorstore = DocArrayInMemorySearch.from\_documents(

docs,

embedding=embeddings # passing in the model to embed documents..

)

retriever = vectorstore.as\_retriever()

retriever.get\_relevant\_documents("what is machine learning?")

A close-up of a computer screen

Description automatically generated

retriever.get\_relevant\_documents("what is gemini pro?")

A close up of text

Description automatically generated

template = """Answer the question a a full sentence, based only on the following context:

{context}

Return you answer in three back ticks

Question: {question}

"""

prompt = ChatPromptTemplate.from\_template(template)

from langchain.schema.runnable import RunnableMap

retriever.get\_relevant\_documents("What is a graident boosted tree?")

A close-up of a computer screen

Description automatically generated

chain = RunnableMap({

"context": lambda x: retriever.get\_relevant\_documents(x["question"]),

"question": lambda x: x["question"]

}) | prompt | model | output\_parser

chain.invoke({"question": "What is machine learning?"})

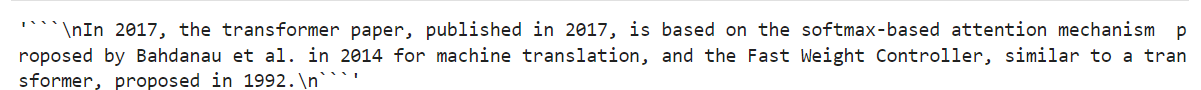
A close-up of a computer screen

Description automatically generated

chain.invoke({"question": "When was the transformer invented?"})

1st time – no answer

2nd time



3rd time

A close up of text

Description automatically generated

Traditional data storage

A screenshot of a computer

Description automatically generated

AI Embeddings

A screenshot of a computer

Description automatically generated

A screenshot of a text

Description automatically generated

A diagram of a movie

Description automatically generated

A white text on a white background

Description automatically generated

A white background with black text

Description automatically generated

A diagram of a function

Description automatically generated with medium confidence

A white text on a white background

Description automatically generated

Gemini langchain vector search RAG

%%capture --no-stderr

!pip3 install -q --upgrade pip

!pip3 install -q google-cloud-aiplatform

!pip3 install -q langchain

!pip3 install -q langchain-community

!pip3 install -q lxml

!pip3 install -q requests

!pip3 install -q beautifulsoup4

!pip3 install -q unstructured

!pip3 install -q langchain-google-genai

!pip3 install -q google-generativeai

!pip3 install -q tqdm

# restart the kernel

import IPython

app = IPython.Application.instance()

app.kernel.do\_shutdown(True)

from IPython.display import display

from IPython.display import Markdown

import textwrap

def to\_markdown(text):

text = text.replace('•', ' \*')

return Markdown(textwrap.indent(text, '> ', predicate=lambda \_: True))

# source API key from GCP project and configure genai client

import os

import pathlib

import textwrap

import google.generativeai as genai

from IPython.display import display

from IPython.display import Markdown

key\_name = !gcloud services api-keys list --filter="gemini-api-key" --format="value(name)"

key\_name = key\_name[0]

api\_key = !gcloud services api-keys get-key-string $key\_name --location="us-central1" --format="value(keyString)"

api\_key = api\_key[0]

os.environ["GOOGLE\_API\_KEY"] = api\_key

genai.configure(api\_key=os.environ["GOOGLE\_API\_KEY"])

# Define project information

import sys

import subprocess

PROJECT\_ID = subprocess.check\_output(["gcloud", "config", "get-value", "project"], text=True).strip()

REGION = "us-central1" # @param {type:"string"}

print(f"Your project ID is: {PROJECT\_ID}")

# Set environment vars

BUCKET = f"gs://{PROJECT\_ID}/embeddings"

DIMENSIONS=768

DISPLAY\_NAME='vertex\_docs\_qa'

ENDPOINT=f"{REGION}-aiplatform.googleapis.com"

TEXT\_GENERATION\_MODEL='gemini-pro'

SITEMAP='https://docs.anthropic.com/sitemap.xml'

import os

from google.cloud import aiplatform

aiplatform.init(project=PROJECT\_ID, location=REGION)

Create documents from vertex ai cloud documentation site

# Parse the xml of sitemap and get URLs of doc site

import requests

from bs4 import BeautifulSoup

def parse\_sitemap(url):

response = requests.get(url)

soup = BeautifulSoup(response.content, "xml")

urls = [element.text for element in soup.find\_all("loc")]

return urls

sites = parse\_sitemap(SITEMAP)

# Use this to filter out docs that don't have a corresponding reference page

sites\_filtered = [url for url in sites if '/en/docs' in url]

len(sites\_filtered)

Load documentation pages using Langchain unstructured URL loader

# This step will take a few minutes to complete

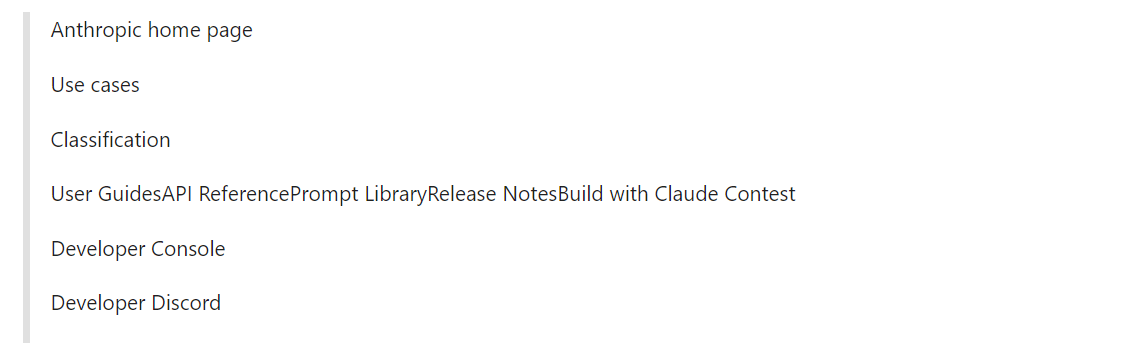
# you will see download messages below the cell after execution

from langchain.document\_loaders import UnstructuredURLLoader

loader = UnstructuredURLLoader(urls=sites\_filtered)

documents = loader.load();

to\_markdown(documents[1].page\_content + "\n\nSource: " + documents[1].metadata["source"])



len(documents)

Create Documents Chunks

# recursively loop through the text and create document chunks for embedding

import warnings

from langchain.text\_splitter import RecursiveCharacterTextSplitter

text\_splitter = RecursiveCharacterTextSplitter(

#separator = "\n",

chunk\_size = 2000,

chunk\_overlap = 100)

document\_chunks = text\_splitter.split\_documents(documents)

print(f"Number documents {len(documents)}")

print(f"Number chunks {len(document\_chunks)}")

document\_chunks=[f"content: {chunk.page\_content}, source: {chunk.metadata['source']}" for chunk in document\_chunks]

docs – 33

chunks – 197

Generate Embeddings from document chunks

# make a documents directory

!rm -rf ./documents

!mkdir ./documents

# view the document chunks in a dataframe

import pandas as pd

df = pd.DataFrame(document\_chunks, columns =['text'])

df

A screenshot of a computer

Description automatically generated

197 rows \* 1 column

# Run this cell to generate the embeddings files you will later upload to Cloud Storage

from tqdm import tqdm

import json

index\_embeddings = []

model = "models/embedding-001"

for index, doc in tqdm(df.iterrows(), total=len(df), position=0):

response = genai.embed\_content(model=model, content=doc['text'], task\_type="retrieval\_query")

doc\_id=f"{index}.txt"

embedding\_dict = {

"id": doc\_id,

"embedding": response["embedding"],

}

index\_embeddings.append(json.dumps(embedding\_dict) + "\n")

with open(f"documents/{doc\_id}", "w") as document:

document.write(doc['text'])

with open("embeddings.json", "w") as f:

f.writelines(index\_embeddings)

embeddings.json generated here

from google.cloud import storage

source\_file = '/home/jupyter/embeddings.json'

destination\_blob\_name = 'embeddings/embeddings.json' # Adjust if needed

client = storage.Client(project=PROJECT\_ID)

bucket = client.bucket(PROJECT\_ID)

blob = bucket.blob(destination\_blob\_name)

blob.upload\_from\_filename(source\_file)

# Upload the embedding files to Cloud Storage

# This step will take a few minutes to complete

import subprocess

gsutil\_command = f"gsutil -q cp -r './documents' gs://{PROJECT\_ID}/documents"

subprocess.run(['gsutil', '-q', 'cp', '-r', './documents', f'gs://{PROJECT\_ID}/documents'])

Create a Vertex AI Vector Store Index

# Create the Vertex AI Vector Search index

# This step will take several minutes to complete

# Wait for this cell to complete before proceeding

index = aiplatform.MatchingEngineIndex.create\_tree\_ah\_index(

display\_name="vertex\_docs",

contents\_delta\_uri=f"gs://{PROJECT\_ID}/embeddings",

dimensions=768,

approximate\_neighbors\_count=150,

distance\_measure\_type="DOT\_PRODUCT\_DISTANCE"

)

index\_endpoint = aiplatform.MatchingEngineIndexEndpoint.create(

display\_name="vertex\_docs",

description="Embeddings for the documentation curated from the sitemap.",

public\_endpoint\_enabled=True,

)

# This step will take up to 20 minutes to complete

# You can view the deployment in the Vertex AI console on the "Vector Search" tab

# Wait for this cell to complete before proceeding

index\_endpoint = index\_endpoint.deploy\_index(

index=index, deployed\_index\_id="vertex\_index\_deployment"

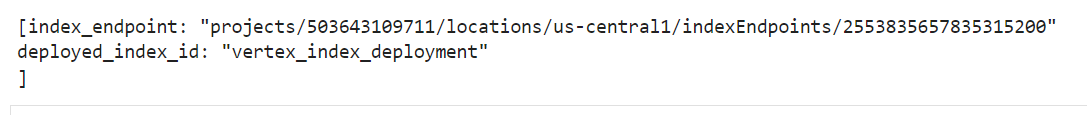
)

INDEX\_RESOURCE\_NAME=index.resource\_name

index = aiplatform.MatchingEngineIndex(index\_name=INDEX\_RESOURCE\_NAME)

deployed\_index = index.deployed\_indexes

deployed\_index



**Search Vector Store, add result as context to a query (without using a LangChain Chain)**

# In the next cells you will query the model directly using the Vertex AI python SDK

from langchain\_google\_genai import GoogleGenerativeAIEmbeddings

from langchain.vectorstores.matching\_engine import MatchingEngine

from langchain.agents import Tool

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

def search\_vector\_store(question):

vector\_store = MatchingEngine.from\_components(

index\_id=INDEX\_RESOURCE\_NAME,

region=REGION,

embedding=embeddings,

project\_id=PROJECT\_ID,

endpoint\_id=deployed\_index[0].index\_endpoint,

gcs\_bucket\_name=f"{PROJECT\_ID}")

relevant\_documentation=vector\_store.similarity\_search(question, k=8)

context = "\n".join([doc.page\_content for doc in relevant\_documentation])[:10000]

return str(context)

from vertexai.preview.generative\_models import GenerativeModel

import warnings

# filter warnings for unused libs

warnings.filterwarnings('ignore')

def ask\_question(question):

context = search\_vector\_store(question)

prompt=f"""

Follow exactly those 3 steps:

1. Read the context below and aggregrate this data

Context : {context}

2. Answer the question using only this context

3. Show the source for your answers

User Question: {question}

If you don't have any context and are unsure of the answer, reply that you don't know about this topic.

"""

model = GenerativeModel("gemini-pro")

response = model.generate\_content(prompt)

return to\_markdown(f"Question: \n{question} \n\n Response: \n {response.text}")

ask\_question("How do I reduce prompt leaks?")

A close-up of a question

Description automatically generated

ask\_question("What use cases and capabilities does Anthropic support?")

A screenshot of a computer screen

Description automatically generated

Create RAG application using langchain

# To answer questions and chain together the prompt, vector search, returned context and model input use a LangChain "Chain"

# In this case you will use the RetrievalQA chain which is commonly used for Question/Answering applications

from langchain\_google\_genai import ChatGoogleGenerativeAI

from langchain.chains import RetrievalQA

# initialize model using chat

model = ChatGoogleGenerativeAI(model="gemini-pro", temperature=0.0, convert\_system\_message\_to\_human=True)

from langchain.prompts import PromptTemplate

template = """

Follow exactly those 3 steps:

1. Read the context below and aggregrate this data

Context : {context}

2. Answer the question using only this context

3. Show the source for your answers

User Question: {question}

If you don't have any context and are unsure of the answer, reply that you don't know about this topic.

"""

prompt = PromptTemplate(input\_variables=["context", "question"], template=template)

from langchain.vectorstores.matching\_engine import MatchingEngine

vector\_store = MatchingEngine.from\_components(

index\_id=INDEX\_RESOURCE\_NAME,

region=REGION,

embedding=embeddings,

project\_id=PROJECT\_ID,

endpoint\_id=deployed\_index[0].index\_endpoint,

gcs\_bucket\_name=f"{PROJECT\_ID}"

)

retriever = vector\_store.as\_retriever(

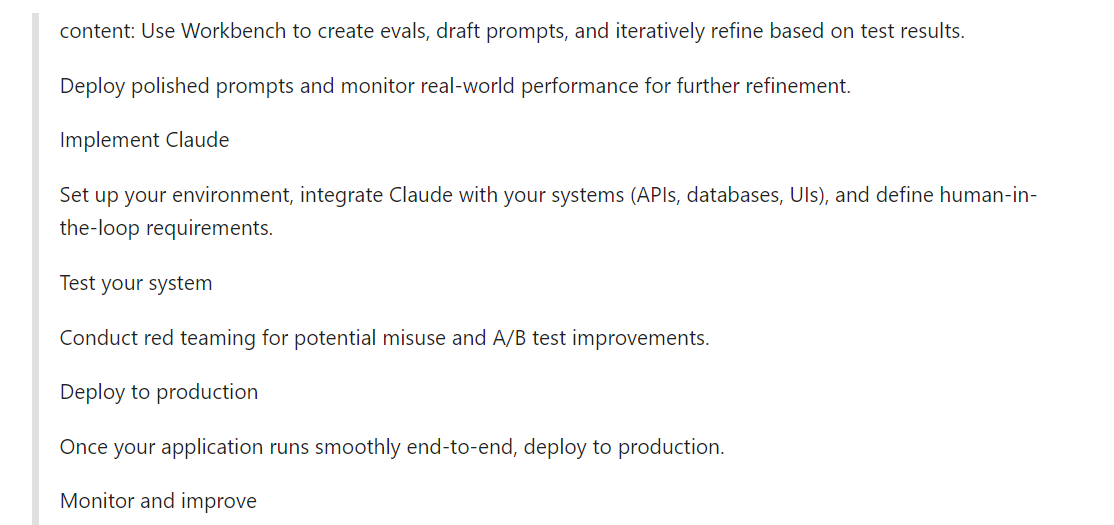
search\_type='similarity',

search\_kwargs={'k': 1}

)

# Test the retriever with a simple search performed above

to\_markdown(retriever.get\_relevant\_documents("How do I get started with Anthropic?")[0].page\_content)



chain\_type\_kwargs = {"prompt": prompt}

qa = RetrievalQA.from\_chain\_type(

llm=model,

chain\_type="stuff",

retriever=retriever,

chain\_type\_kwargs=chain\_type\_kwargs,

return\_source\_documents=True

)

def ask\_question(question: str):

response = qa({"query": question})

# since k is set to 1 only return the first source retrieved

source = response['source\_documents']

return to\_markdown(f"Response: \n\n {response['result']}")

# Note: You will see a library warning when running this step

ask\_question("How do I get started with Anthropic?")

A white background with black text

Description automatically generated