Quick tip: Review the prerequisites before you run the lab

End Lab 00:24:11

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. Learn more.

Open Google Console

Username
student-00-dee3a0d3ba6a@

Password
2B9MvBoCENhN

GCP Project ID
qwiklabs-gcp-04-e3fbc7d

Improving RAG Solutions

Lab · 1 hour · No cost · Intermediate

Rate Lab

This lab may incorporate AI tools to support your learning.

Overview

In this lab, we'll develop a Retrieval Augmented Generation (RAG) system using embeddings and Large Language Models (LLMs). We'll explore common issues and techniques to improve the overall recall and accuracy of results. Our toolkit includes Google Cloud technologies such as Vertex AI Workbench, the latest Gemini LLM model, and the Google Cloud Embeddings APIs.

Objectives

In this lab, you will:

- Set up a RAG system.
- Explore the distribution of embeddings with user queries.
- Implement query augmentation.
- Perform result re-ranking.
- Utilize Embedding Adapters.

Setup and requirements

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This Qwiklabs hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

What you need

To complete this lab, you need:

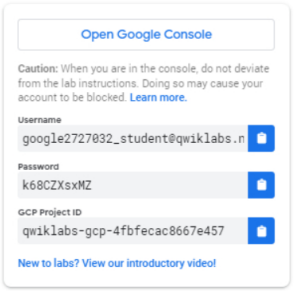
- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

Note: If you already have your own personal Google Cloud account or project, do not use it for this lab.

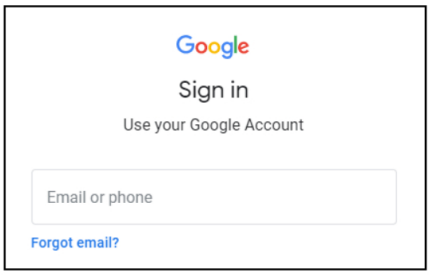
Note: If you are using a Pixelbook, open an Incognito window to run this lab.

How to start your lab and sign in to the Google Cloud Console

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.

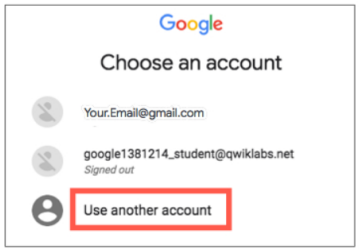


2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.



Tip: Open the tabs in separate windows, side-by-side.

If you see the **Choose an account** page, click **Use Another Account**.



3. In the **Sign in** page, paste the username that you copied from the Connection Details panel. Then copy and paste the password.

Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

4. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

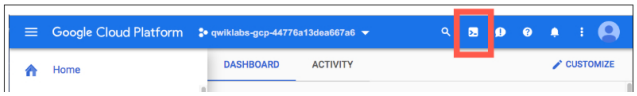
After a few moments, the Cloud Console opens in this tab.

Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.

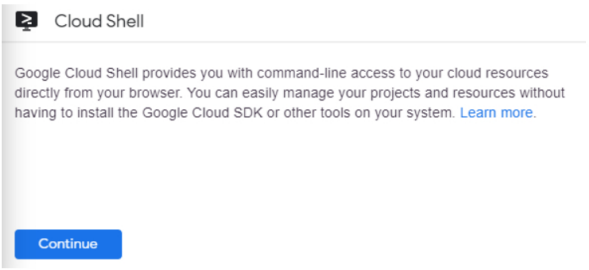
Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

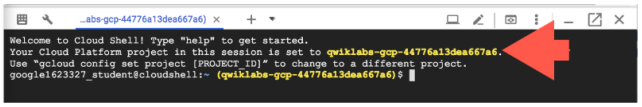
In the Cloud Console, in the top right toolbar, click the **Activate Cloud Shell** button.



Click **Continue**.



It takes a few moments to provision and connect to the environment. When you are connected, you are already authenticated, and the project is set to your *PROJECT_ID*. For example:



`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

You can list the active account name with this command:

```
gcloud auth list
```

(Output)

```
Credentialed accounts:
 - <myaccount>@<mydomain>.com (active)
```

(Example output)

```
Credentialed accounts:
 - google1623327_student@qwiklabs.net
```

You can list the project ID with this command:

```
gcloud config list project
```

(Output)

```
[core]
project = <project_ID>
```

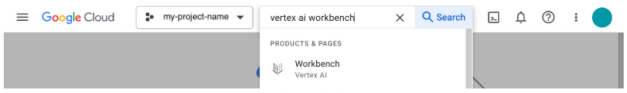
(Example output)

```
[core]
project = qwiklabs-gcp-44776a13dea667a6
```

For full documentation of `gcloud` see the gcloud command-line tool overview.

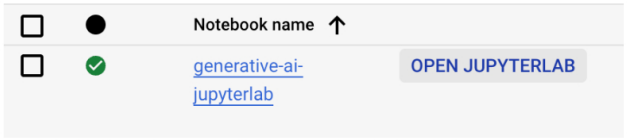
Task 1. Open a Jupyter notebook in Vertex AI Workbench

1. In your Google Cloud project, navigate to Vertex AI Workbench. In the top search bar of the Google Cloud console, enter **Vertex AI Workbench**, and click on the first result.



2. Under **User-Managed Notebooks**, click on **Open Jupyterlab** for **generative-ai-jupyterlab** notebook.

The JupyterLab will run in a new tab.



3. On the **Launcher**, under **Notebook**, click on **Python 3** to open a new python notebook.

Task 2. Setting up a RAG solution

In this first task, we will setup the Workbench environment, create a RAG solution from Alphabets 2022 financial annual report and perform some queries to the report. To store the embeddings we use Chroma, an open-source embeddings database that makes it straightforward to store embeddings.

1. In the first cell run the following command to install the Google Cloud Vertex AI SDKs. Either click the play ▶ button at the top or enter **SHIFT+ENTER** on your keyboard to execute the cell.

```
!pip3 install --upgrade --user google-cloud-aiplatform umap-learn
tqdm pypdf
```

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

```
import IPython
from IPython.display import Markdown, display
import time

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

3. Initialize the environment variables for the current project.

```
PROJECT = !gcloud config get-value project
PROJECT_ID = PROJECT[0]
LOCATION = "us-east1"
```

4. Import python display utilities and Google Cloud's Embeddings and Gemini models.

```
from vertexai.preview.generative_models import GenerativeModel
from vertexai.language_models import TextEmbeddingModel
text_embedding_model = TextEmbeddingModel.from_pretrained("text-
embedding-004")
model = GenerativeModel('gemini-pro')
```

5. Download Alphabet's financial annual report for 2022 so that we can query it.

```
import urllib.request

# download alphabet's annual pdf report
url = "https://abc.xyz/assets/d4/4f/a48b94d548d0b2fdc029a95e8c63"
file = "2022-alphabet-annual-report.pdf"

urllib.request.urlretrieve(f"{url}/{file}", filename=f"{file}")
```

6. Convert the PDF into text.

```
!pip install PyPDF2
import PyPDF2

# Open the PDF file in binary mode
with open('2022-alphabet-annual-report.pdf', 'rb') as file:
    # Create a PdfFileReader object
    pdf_reader = PyPDF2.PdfReader(file)
    pdf_texts = [p.extract_text().strip() for p in
pdf_reader.pages]

    # Filter the empty strings
    pdf_texts = [text for text in pdf_texts if text]

    print(pdf_texts[0])
```

7. Split the PDF into chunks so that we can create embedding out of them. The embedding's APIs and the LLM have token limits, that's why we need to split the text into smaller units as opposed to sending the whole text.

```
!pip install langchain sentence-transformers
from langchain.text_splitter import
RecursiveCharacterTextSplitter,
SentenceTransformersTokenTextSplitter
character_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", ". ", " ", ""],
    chunk_size=1000,
    chunk_overlap=0
)
character_split_texts =
character_splitter.split_text('\n\n'.join(pdf_texts))

print(character_split_texts[10])
print(f"\nTotal chunks: {len(character_split_texts)}")
```

8. Create more chunks if the size of the existing chunk is too big. In this case, we define too big as 256 tokens.

```
token_splitter =
SentenceTransformersTokenTextSplitter(chunk_overlap=0,
tokens_per_chunk=256)

token_split_texts = []
for text in character_split_texts:
    token_split_texts += token_splitter.split_text(text)

print(token_split_texts[10])
print(f"\nTotal chunks: {len(token_split_texts)}")
```

9. Install and initialize the embeddings database with an embeddings function.

```
!pip3 install chromadb==0.5.3
!pip3 install google-generativeai
import chromadb
import os
from chromadb.utils.embedding_functions import
SentenceTransformerEmbeddingFunction
# import chromadb.utils.embedding_functions as
embedding_functions

# import getpass
# import os

# getpass will prompt for an API Key
# The API Key is needed for Chroma DB
# API_KEY = getpass.getpass("Provide your Google API Key")

# embedding_function =
embedding_functions.GooglePalmEmbeddingFunction(api_key=creds.toker

embedding_function = SentenceTransformerEmbeddingFunction()
print(embedding_function([token_split_texts[10]]))
```

10. Create embedings and store them in the chroma database.

```
chroma_client = chromadb.Client()
print("collection")
chroma_collection =
```

```
chroma_client.create_collection("alphabet_annual_report_2022",
embedding_function=embedding_function)
print("created")
ids = [str(i) for i in range(len(token_split_texts))]
print("adding")
chroma_collection.add(ids=ids, documents=token_split_texts)
chroma_collection.count()
```

11. Ask a questions to perform an embeddings search on the chroma db database.

```
query = "What was the total revenue?"

results = chroma_collection.query(query_texts=[query],
n_results=5)
retrieved_documents = results['documents'][0]

for document in retrieved_documents:
    print(document)
    print('\n')
```

12. Use a RAG to answer the question given the information that was looked up from the database.

```
def rag(query, retrieved_documents, model):
    information = "\n\n".join(retrieved_documents)

    prompt = (f'You are a helpful expert financial research
assistant.\n'
    f'Your users are asking questions about information contained
in an annual report.\n'
    f'You will be shown the user\'s question, and the relevant
information from the annual report.\n'
    f'Answer the user\'s question using only this
information.\n\n'
    f'Question: {query}. \n Information: {information}')

    responses = model.generate_content(prompt, stream=False)
    return responses.text

response = rag(query=query,
retrieved_documents=retrieved_documents, model=model)
print(response)
```

Task 3. Explore embeddings distribution with user queries

In this task, we use the python library `umap` to reduce the dimensions of our embeddings to two dimensions, so that we can see how close they are represented in a graph. Then we try different queries to see how different queries relate to the existing embedding space. This can give us an intuition about whether the answer might be found in the embeddings space.

1. Transform embeddigns into a two-dimensional space.

```
import umap.umap_ as umap
import numpy as np
from tqdm import tqdm

embeddings = chroma_collection.get(include=['embeddings'])
['embeddings']
umap_transform = umap.UMAP(random_state=0,
transform_seed=0).fit(embeddings)

def project_embeddings(embeddings, umap_transform):
    umap_embeddings = np.empty((len(embeddings),2))
    for i, embedding in enumerate(tqdm(embeddings)):
        umap_embeddings[i] =
umap_transform.transform([embedding])
    return umap_embeddings

projected_dataset_embeddings = project_embeddings(embeddings,
umap_transform)
```

2. Plot the two-dimensional embeddings.

```
import matplotlib.pyplot as plt

def plot(title='Projected Embeddings',
projected_dataset_embeddings=[], projected_query_embedding=[],
projected_retrieved_embeddings=[]):
    # Plot the projected query and retrieved documents in the
embedding space
    plt.figure()
    plt.scatter(projected_dataset_embeddings[:, 0],
projected_dataset_embeddings[:, 1], s=10, color='gray')
    if len(projected_query_embedding) > 0:
        plt.scatter(projected_query_embedding[:, 0],
projected_query_embedding[:, 1], s=150, marker='X', color='r')
    if len(projected_retrieved_embeddings) > 0:
        plt.scatter(projected_retrieved_embeddings[:, 0],
projected_retrieved_embeddings[:, 1], s=100, facecolors='none',
edgecolors='g')

    plt.gca().set_aspect('equal', 'datalim')
    plt.title(f'{title}')
    plt.axis('off')

plot(projected_dataset_embeddings=projected_dataset_embeddings)
```

3. Ask the same question to the database again, but this time retrieve the embeddings in addition to the documents.

```
query = "What was the total revenue?"

results = chroma_collection.query(query_texts=query, n_results=5,
include=['documents', 'embeddings'])

retrieved_documents = results['documents'][0]

for document in results['documents'][0]:
    print(document)
    print('')
```

4. Embed the question and answer and transform it into two dimentions.

```
query_embedding = embedding_function([query])[0]
retrieved_embeddings = results['embeddings'][0]

projected_query_embedding = project_embeddings([query_embedding],
umap_transform)
projected_retrieved_embeddings =
project_embeddings(retrieved_embeddings, umap_transform)
```

5. Plot the projected query and retrieved documents in the embedding space.

```
plot(title=query,
projected_dataset_embeddings=projected_dataset_embeddings,
projected_query_embedding=[],
projected_retrieved_embeddings=projected_retrieved_embeddings)
```

6. Let's try again with a different question to see another example.

```
query = "What is the strategy around artificial intelligence (AI)
?"
results = chroma_collection.query(query_texts=query, n_results=5,
include=['documents', 'embeddings'])

retrieved_documents = results['documents'][0]

for document in results['documents'][0]:
    print(document)
    print('')
```

7. Create embeddings and projections.

```
query_embedding = embedding_function([query])[0]
retrieved_embeddings = results['embeddings'][0]
```

```
retrieved_embeddings = results['embeddings'][0]

projected_query_embedding = project_embeddings([query_embedding],
umap_transform)
projected_retrieved_embeddings =
project_embeddings(retrieved_embeddings, umap_transform)
```

8. Plot the projected query and retrieved documents in the embedding space. Notice how close results are, since the answer seems to be contained in the database.

```
plot(title=query,
projected_dataset_embeddings=projected_dataset_embeddings,
projected_query_embedding=[],
projected_retrieved_embeddings=projected_retrieved_embeddings)
```

9. Let's do all these again with another question that might be contained in the dataset to see that results are still close to one another.

```
query = "What are the company's long-term financial goals?"
results = chroma_collection.query(query_texts=query, n_results=5,
include=['documents', 'embeddings'])

retrieved_documents = results['documents'][0]

for document in results['documents'][0]:
    print(document)
    print('')

query_embedding = embedding_function([query])[0]
retrieved_embeddings = results['embeddings'][0]

projected_query_embedding = project_embeddings([query_embedding],
umap_transform)
projected_retrieved_embeddings =
project_embeddings(retrieved_embeddings, umap_transform)

plot(title=query,
projected_dataset_embeddings=projected_dataset_embeddings,
projected_query_embedding=[],
projected_retrieved_embeddings=projected_retrieved_embeddings)
```

10. Now, let's use an unrelated query and we'll observe how the results are more scattered around. Notice that the algorithm always returns close neighbors even if they are far appart. This is an indication that is returning results that are not necesseraly related to the query.

```
query = "What has Tom Brady done for us lately?"
results = chroma_collection.query(query_texts=query, n_results=5,
include=['documents', 'embeddings'])

retrieved_documents = results['documents'][0]

for document in results['documents'][0]:
    print(document)
    print('')

query_embedding = embedding_function([query])[0]
retrieved_embeddings = results['embeddings'][0]

projected_query_embedding = project_embeddings([query_embedding],
umap_transform)
projected_retrieved_embeddings =
project_embeddings(retrieved_embeddings, umap_transform)

plot(title=query,
projected_dataset_embeddings=projected_dataset_embeddings,
projected_query_embedding=[],
projected_retrieved_embeddings=projected_retrieved_embeddings)
```

Task 4. Query augmentation

One way of improving the results from queries is by getting more results of similar queries. This can be done either by creating similar queries or by sending an answer of what you might expect, so that you attach more context. In this task, you learn to expand your queries leveraging LLMs to do the heavy lifting.

1. Augment the query with a probable answer to give more context to the vector search.

```
def augment_query_generated(query, model):
    information = "\n\n".join(retrieved_documents)

    prompt = (f'You are a helpful expert financial research
assistant.\n'
    f'Provide an example answer to the given question, that might
be found in a document like an annual report.\n'
    f'Question: {query}.')

    responses = model.generate_content(prompt, stream=False)
    return responses.text

original_query = "Was there significant turnover in the executive
team?"
hypothetical_answer = augment_query_generated(original_query,
model)

joint_query = f"{original_query} {hypothetical_answer}"
print(joint_query)
```

2. Perform the query with the new augmented query.

```
results = chroma_collection.query(query_texts=joint_query,
n_results=5, include=['documents', 'embeddings'])
retrieved_documents = results['documents'][0]

for doc in retrieved_documents:
    print(doc)
    print('')
```

3. Search for embeddings for the augmented query.

```
results = chroma_collection.query(query_texts=joint_query,
n_results=5, include=['documents', 'embeddings'])
retrieved_documents = results['documents'][0]

for doc in retrieved_documents:
    print(doc)
    print('')
```

4. Get the embeddings for the queries and results and then map them to a 2-dimensional space.

```
retrieved_embeddings = results['embeddings'][0]
original_query_embedding = embedding_function([original_query])
augmented_query_embedding = embedding_function([joint_query])

projected_original_query_embedding =
project_embeddings(original_query_embedding, umap_transform)
projected_augmented_query_embedding =
project_embeddings(augmented_query_embedding, umap_transform)
projected_retrieved_embeddings =
project_embeddings(retrieved_embeddings, umap_transform)
```

5. Map the results to get a visual understanding of the outcome. Notice that the seleted answers are closer to the augmented query datapoint.

```
plt.figure()
plt.scatter(projected_dataset_embeddings[:, 0],
projected_dataset_embeddings[:, 1], s=10, color='gray')
plt.scatter(projected_retrieved_embeddings[:, 0],
projected_retrieved_embeddings[:, 1], s=100, facecolors='none',
edgecolors='g')
plt.scatter(projected_original_query_embedding[:, 0],
projected_original_query_embedding[:, 1], s=150, marker='X',
color='r')
plt.scatter(projected_augmented_query_embedding[:, 0],
projected_augmented_query_embedding[:, 1], s=150, marker='X',
color='orange')
```

```
plt.gca().set_aspect('equal', 'datalim')
plt.title(f'{original_query}')
plt.axis('off')
```

6. Augment the query with a additional queries.

```
def augment_multiple_query(query, model):
    information = "\n\n".join(retrieved_documents)

    prompt = (f'You are a helpful expert financial research
assistant.\n'
    f'Your users are asking questions about an annual report.\n'
    f'Suggest up to five additional related questions to help
them find the information they need, for the provided
question.\n'
    f'Suggest only short questions without compound sentences.
Suggest a variety of questions that cover different aspects of
the topic.\n'
    f'Make sure they are complete questions, and that they are
related to the original question.\n'
    f'Output one question per line. Do not number the
questions.\n'
    f'Question: {query}.')

    responses = model.generate_content(prompt, stream=False)
    return responses.text

original_query = "What were the most important factors that
contributed to increases in revenue?"
augmented_queries = augment_multiple_query(original_query, model)

joint_query = f"{original_query} \n{augmented_queries}"
print(joint_query)
```

7. Retrieve answers for all of these queries.

```
queries = [original_query] + augmented_queries.split('\n')
results = chroma_collection.query(query_texts=queries,
n_results=5, include=['documents', 'embeddings'])

retrieved_documents = results['documents']

# Deduplicate the retrieved documents
unique_documents = set()
for documents in retrieved_documents:
    for document in documents:
        unique_documents.add(document)

for i, documents in enumerate(retrieved_documents):
    print(f"Query: {queries[i]}")
    print('')
    print("Results:")
    for doc in documents:
        print(doc)
        print('')
    print('-'*100)
```

8. Project the queries embeddings into a 2-dimensional space.

```
original_query_embedding = embedding_function([original_query])
augmented_query_embeddings =
embedding_function(augmented_queries)

project_original_query =
project_embeddings(original_query_embedding, umap_transform)
project_augmented_queries =
project_embeddings(augmented_query_embeddings, umap_transform)
```

9. Project the answer embeddings into a 2-dimensional space.

```
result_embeddings = results['embeddings']
result_embeddings = [item for sublist in result_embeddings for
item in sublist]
projected_result_embeddings =
project_embeddings(result_embeddings, umap_transform)
```

10. Plot the queries and answers to visualize their representation in space.

```
import matplotlib.pyplot as plt

plt.figure()
plt.scatter(projected_dataset_embeddings[:, 0],
projected_dataset_embeddings[:, 1], s=10, color='gray')
plt.scatter(project_augmented_queries[:, 0],
project_augmented_queries[:, 1], s=150, marker='X',
color='orange')
plt.scatter(projected_result_embeddings[:, 0],
projected_result_embeddings[:, 1], s=100, facecolors='none',
edgecolors='g')
plt.scatter(project_original_query[:, 0],
project_original_query[:, 1], s=150, marker='X', color='r')

plt.gca().set_aspect('equal', 'datalim')
plt.title(f'{original_query}')
plt.axis('off')
```

Task 5. Re-ranking results

With the query expansion technique that we just discussed, we increased the cardinality of responses. We have a new problem now. The high amount of responses might be larges than the LLM's input token limit and, it will be more expensive to process such a high amount of tokens, since we are billed per token. One solution to this problem is to rerank the results and only send the most relevant ones. Now, instead of using the same cosine-similarity method as before, you could use a cross-encoder, a method that usually yields better results.

1. Import a cross encoder model.

```
from sentence_transformers import CrossEncoder
cross_encoder = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-
v2')
```

2. Create unique pairs of query to document.

```
pairs = []
for doc in unique_documents:
    pairs.append([original_query, doc])
```

3. Compute an print the similarity between the pairs.

```
scores = cross_encoder.predict(pairs)

print("Scores:")
for score in scores:
    print(score)
```

4. Reorder the pairs in terms of relevance.

```
print("New Ordering:")
ranked_docs=['']*len(scores)
i = 0
for position in np.argsort(scores)[::-1]:
    ranked_docs[position] = pairs[i][1]
    i+=1
    print(position)
```

5. Print the 5 most relevant documents.

```
ranked_docs[:5]
```

6. Only send the most relevant answers to the LLM.

```
response = rag(query=original_query
```

```
response = rag(query=original_query,
retrieved_documents=ranked_docs[:5], model=model)
print(original_query)
print(response)
```

Task 6. Embedding Adapters

Another improvement that we can make is instead of getting more answers and reranking, train the embedding model to work with our dataset based on user queries. This is a form a of fine tuning that you can do on your own.

1. Generate queries that might be asked to a financial statement. If you had real users, you would use those queries instead of making them up.

```
def generate_queries(model):

    prompt = (f'You are a helpful expert financial research
assistant.\n'
    f'You help users analyze financial statements to better
understand companies.\n'
    f'Suggest 10 to 15 short questions that are important to ask
when analyzing an annual report.\n'
    f'Do not output any compound questions (questions with
multiple sentences or conjunctions).\n'
    f'Output each question on a separate line divided by a
newline.')

    responses = model.generate_content(prompt, stream=False)
    return responses.text.split('\n')

generated_queries = generate_queries(model)
for query in generated_queries:
    print(query)
```

2. Search answers for those queries.

```
results = chroma_collection.query(query_texts=generated_queries,
n_results=10, include=['documents', 'embeddings'])
retrieved_documents = results['documents']
```

3. Evaluate the results in 1 for relevant and -1 for irrelevant answers.

```
def evaluate_results(query, statement, model):

    prompt = (f'You are a helpful expert financial research
assistant.\n'
    f'You help users analyze financial statements to better
understand companies.\n'
    f'For the given query, evaluate whether the following
satement is relevant.\n'
    f'Output only \'yes\' or \'no\'.\n'
    f'Question: {query}, Statement: {statement}.')

    responses = model.generate_content(prompt, stream=False)
    if responses.text == 'yes':
        return 1
    return -1

retrieved_embeddings = results['embeddings']
query_embeddings = embedding_function(generated_queries)
```

4. Extract the documents and queries into their own lists.

```
adapter_query_embeddings = []
adapter_doc_embeddings = []
adapter_labels = []

for q, query in enumerate(tqdm(generated_queries)):
    for d, document in enumerate(retrieved_documents[q]):
        adapter_query_embeddings.append(query_embeddings[q])
        adapter_doc_embeddings.append(retrieved_embeddings[q][d])
```

```
        adapter_labels.append(evaluate_results(query, document,
model))

len(adapter_labels)
```

5. Install PyTorch so that we can train the embedding model based on relevant answers.

```
!pip install torch

import torch
```

6. Initialize PyTorch with the right data.

```
adapter_query_embeddings =
torch.Tensor(np.array(adapter_query_embeddings))
adapter_doc_embeddings =
torch.Tensor(np.array(adapter_doc_embeddings))
adapter_labels =
torch.Tensor(np.expand_dims(np.array(adapter_labels),1))

dataset =
torch.utils.data.TensorDataset(adapter_query_embeddings,
adapter_doc_embeddings, adapter_labels)
```

7. Set up the model with cosine similarity, so that embeddings with label 1 are similar and -1 are dissimilar.

```
def model(query_embedding, document_embedding, adaptor_matrix):
    updated_query_embedding = torch.matmul(adaptor_matrix,
query_embedding)
    return torch.cosine_similarity(updated_query_embedding,
document_embedding, dim=0)
```

8. Compute the mean squared error (MSE) loss to see how close are we to the expected output.

```
def mse_loss(query_embedding, document_embedding, adaptor_matrix,
label):
    return torch.nn.MSELoss()(model(query_embedding,
document_embedding, adaptor_matrix), label)
```

9. Initialize the adaptor matrix.

```
mat_size = len(adapter_query_embeddings[0])
adapter_matrix = torch.randn(mat_size, mat_size,
requires_grad=True)
```

10. Train the adaptor matrix using 100 steps by iteratively update a vector such that if I take that vector and multiply it by these inputs (query embeddings) then compare that result to this vector (this example's retrieved document vector), I get a score closer to (this example's -1 or 1 label).

```
min_loss = float('inf')
best_matrix = None

for epoch in tqdm(range(100)):
    for query_embedding, document_embedding, label in dataset:
        loss = mse_loss(query_embedding, document_embedding,
adapter_matrix, label)

        if loss < min_loss:
            min_loss = loss
            best_matrix = adapter_matrix.clone().detach().numpy()

        loss.backward()
        with torch.no_grad():
            adapter_matrix -= 0.01 * adapter_matrix.grad
            adapter_matrix.grad.zero_()
```

11. Print the loss. Notice there is almost a 30% increase in accuracy.

```
print(f"Best loss: {min_loss.detach().numpy()}")
```

12. Create a scaled vector based on the best matrix that we computed.

```
test_vector = torch.ones((mat_size,1))
scaled_vector = np.matmul(best_matrix, test_vector).numpy()
```

13. Plot the scale vector answers.

```
import matplotlib.pyplot as plt
plt.bar(range(len(scaled_vector)), scaled_vector.flatten())
plt.show()
```

14. Get the adapted query embeddings to compare them with the original values retrieved.

```
query_embeddings = embedding_function(generated_queries)
adapted_query_embeddings = np.matmul(best_matrix,
np.array(query_embeddings).T).T

projected_query_embeddings = project_embeddings(query_embeddings,
umap_transform)
projected_adapted_query_embeddings =
project_embeddings(adapted_query_embeddings, umap_transform)
```

15. Plot the adapted embeddings and the original values retrieved.

```
# Plot the projected query and retrieved documents in the
embedding space
plt.figure()
plt.scatter(projected_dataset_embeddings[:, 0],
projected_dataset_embeddings[:, 1], s=10, color='gray')
plt.scatter(projected_query_embeddings[:, 0],
projected_query_embeddings[:, 1], s=150, marker='X', color='r',
label="original")
plt.scatter(projected_adapted_query_embeddings[:, 0],
projected_adapted_query_embeddings[:, 1], s=150, marker='X',
color='green', label="adapted")

plt.gca().set_aspect('equal', 'datalim')
plt.title("Adapted Queries")
plt.axis('off')
plt.legend()
```

Congratulations

Congratulations! You've successfully created a Retrieval Augmented Generation (RAG) system using embeddings and LLMs.

Manual Last Updated July 10, 2024

Lab Last Tested July 10, 2024

Ready for more?

Here's another lab we think you'll like.

LAB

A Tour of Google Cloud Hands-on Labs

0% Continue