Quick tip: Review the prerequisites before you run the lab

Start Lab 03:00:00

Using Vertex AI Multimodal Embeddings and Vector Search

Lab 3 hours No cost Intermediate

★★★★☆

This lab may incorporate AI tools to support your learning.

GENAI025

Previous Next

Overview

This lab demonstrates how to create text-to-image embeddings using the DiffusionDB dataset and the Vertex AI Multimodal Embeddings model. The embeddings are uploaded to the Vertex AI Vector Search service, which is a high scale, low latency solution to find similar vectors for a large corpus. Moreover, it is a fully managed offering, further reducing operational overhead. It is built upon Approximate Nearest Neighbor (ANN) technology developed by Google Research.

To learn more about Vertex AI Multimodal Embeddings, and Vertex AI Vector Search.

Objective

In this lab, you learn how to encode custom text embeddings, create an Approximate Nearest Neighbor (ANN) index, and query against indexes.

This lab uses the following Google Cloud ML services:

- Vertex AI Multimodal Embeddings
- Vertex AI Vector Search

- Convert an image dataset to embeddings
- Create an index
- Upload embeddings to the index
- Create an index endpoint
- Deploy the index to the index endpoint
- Perform an online query

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

of the lab.

What you need

To complete this lab, you need:

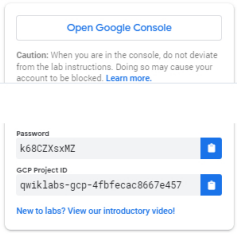
- Access to a standard internet browser (Chrome browser recommended).
- Time to complete the lab.

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

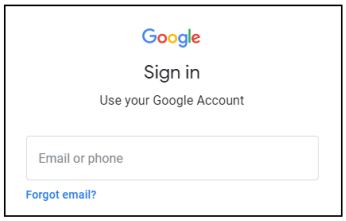
Note: If you are using a Pixelbook, open an Incognito window to run this lab.

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

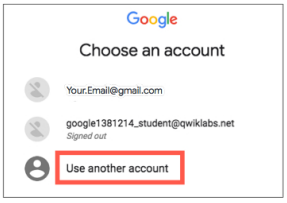
1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.



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.



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

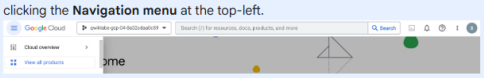
Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

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).

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.



Dataset

The dataset used for this tutorial is the DiffusionDB dataset.

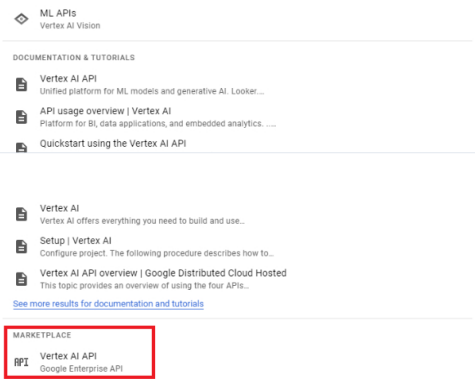
DiffusionDB is the first large-scale text-to-image prompt dataset. It contains 14 million images generated by Stable Diffusion using prompts and hyperparameters specified by real users. The unprecedented scale and diversity of this human-actuated dataset provide exciting research opportunities in understanding the interplay between prompts and generative models, detecting deepfakes, and designing human-AI interaction tools to help users more easily use these models.

Task 1. Enable APIs

In this task, you need to enable the Vertex AI Conversation APIs and Service Networking API.

To enable the Vertex AI Conversation API follow these steps:

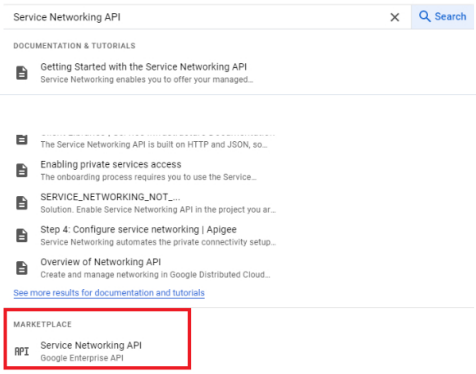
1. Type **Vertex AI api** into the top search bar of the Google Cloud console, choose the selected result as below, and continue.



2. If not already enabled, click **Enable**, read and agree to the Terms of Service, then click **Continue and activate the API**.

To enable the Service Networking API, follow these steps:

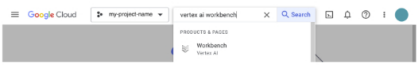
3. Type **Service Networking API** into the top search bar of the Google Cloud console, choose the first result.



4. If not already enabled, click **Enable**.

Task 2. Open a Jupyter notebook in Vertex AI Workbench

search bar of the Google Cloud console, enter **Vertex AI Workbench**, and click on the first result.



2. Under **Instances**, click on **Open JupyterLab** for **generative-ai-jupyterlab** notebook.

   The JupyterLab will run in a new tab.

notebook.

Task 3. Set up the Jupyter notebook enviornment

1. In the first cell, enter the following command to install the latest version of Cloud Storage and the Gen AI SDK for Python. To run the command, hit **SHIFT+ENTER**, or click on play button at the top.

```
# Install the packages
! pip install --upgrade google-genai google-cloud-storage
requests matplotlib pillow tqdm tenacity
```

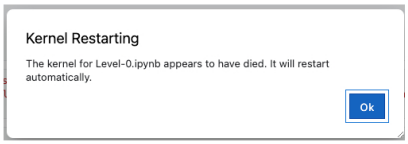
Output:

2. Install the latest version of google-cloud-vision for filtering for safe images.

```
# Install the packages
! pip install google-cloud-vision
```

3. Finally, restart kernel after installs so that your environment can access the new packages.

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



4. Setup the environment values for your project.

```
PROJECT = !gcloud config get-value project
PROJECT_ID = PROJECT[0]
REGION = "Lab Default Region"
```

Task 4. Create a Cloud Storage bucket

Let's create a Cloud Storage bucket to store intermediate artifacts such as datasets.

1. To create a Cloud Storage bucket, use the following commands.

```
BUCKET_URI = f"gs://artifacts-{PROJECT_ID}-unique"  #
@param {type:"string"}
```

```
! gsutil mb -l {REGION} -p {PROJECT_ID} {BUCKET_URI}
```

Task 5. Prepare the data

You'll use DiffusionDB dataset of image prompt and image pairs.

```
! git clone https://github.com/poloclub/diffusiondb
```

Output:

```
Cloning into 'diffusiondb'...
remote: Enumerating objects: 437, done.
remote: Counting objects: 100% (68/68), done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 437 (delta 24), reused 54 (delta 15), pack-reused 369
Receiving objects: 100% (437/437), 7.02 MiB | 17.03 MiB/s, done.
Resolving deltas: 100% (238/238), done.
```

2. Install the dependencies for downloading the dataset

```
! pip install -r diffusiondb/requirements.txt
```

5. Download image files

```
# Download image files from 1. The file contains of 1000
images.
! python diffusiondb/scripts/download.py -i 2 -r 3
```

Output:

```
Downloading files |████████████████████████████████| 1/1 [100%] in 2.9s
```

4. Extract image archives

```
# Unzip all image files
image_directory = "extracted"

! unzip -n 'images/*.zip' -d '{image_directory}'
```

Note: This code snippet may run for a few minutes.

Output:

```
inflating: extracted/a59fd1e8-42ee-4aac-b7b9-f130fda0cb0.png
inflating: extracted/e60cbfe1-bfff-4c71-b01a-50ce6d413354.png
inflating: extracted/dd6cf0b3-e688-4911-bfe1-b93f809fb085.png
inflating: extracted/11b30384-1954-4335-bbc1-e985d25bc652.png
inflating: extracted/52345567-b55f-41af-9ecc-ef44b887cc8e.png
inflating: extracted/5bec8396-f043-49d7-a503-85e5e3a1c3b3.png
inflating: extracted/dbeba0e4-584c-4ce2-b8bc-5bd83a88130e.png
inflating: extracted/cb4d8f7f-c454-45f2-a6bf-d9cb71963bde.png
inflating: extracted/9c96ecbe-a749-4c47-bd30-8853606c3c87.png
inflating: extracted/323d69d8-0d4f-425c-8716-bc941021ca10.png
inflating: extracted/4a99d1d5-f7da-41de-b92a-e27567d49421.png
inflating: extracted/6ac3ef00-5d58-48d0-8875-84388a171b2d.png
inflating: extracted/3ad4a6ac-a23e-4098-9d5a-a44242df5873.png
inflating: extracted/2b79ac47-479a-411a-a05c-96492192124d.png
inflating: extracted/dfd79709-26b3-4ce7-a9cf-9c3d47d5378c.png
inflating: extracted/9b0589f1-c4d9-4985-8243-491fa0195072.png
```

5. Load image metadata

```
import json
import os

metadatas = {}
for file_name in os.listdir(image_directory):
    if file_name.endswith(".json"):
        with open(os.path.join(image_directory, file_name))
```

```
            metadatas.update(metadata_chunk)

# Get image names and paths from the loaded metadata
image_names_all = list(metadatas.keys())
image_paths_all = [os.path.join(image_directory,
image_name) for image_name in image_names_all]

len(metadatas)
```

Output:

```
1000
```

Task 6. Define function to detect explicit images

1. Run the following code in the next cell of the notebook to use Google Cloud Vision API to perform safe search detection of images.

```
from typing import Optional

from google.cloud import vision
from google.cloud.vision_v1.types.image_annotator import
SafeSearchAnnotation

client = vision.ImageAnnotatorClient()


def detect_safe_search(path: str) ->
Optional[SafeSearchAnnotation]:
    """Detects unsafe features in the file."""
```

```
    with open(path, "rb") as image_file:
        content = image_file.read()

    image = vision.Image(content=content)

    response = client.safe_search_detection(image=image)
```

```
        return None

    return response.safe_search_annotation
```

2. Define a function to convert SafeSearch annotation results to a boolean value.

```
from google.cloud.vision_v1.types.image_annotator import
Likelihood


# Returns true if some annotations have a potential safety
issues
def convert_annotation_to_safety(safe_search_annotation:
SafeSearchAnnotation) -> bool:
    return all(
        [
            (safe_level == Likelihood.VERY_UNLIKELY)
            or (safe_level == Likelihood.UNLIKELY)
            for safe_level in [
                safe_search_annotation.adult,
                safe_search_annotation.medical,
                safe_search_annotation.violence,
                safe_search_annotation.racy,
            ]
```

Perform rate-limited explicit image detection

Cloud Vision has a rate limit for API requests. Use a rate limiter to ensure the requests go under this limit. For better performance, use a ThreadPool to make parallel requests.

3. Run the following code snippet in the next available cell to introduce a rate limiter, and process images concurrently using a `ThreadPoolExecutor`.

```
import time
from concurrent.futures import ThreadPoolExecutor
from typing import Optional

import numpy as np
from tqdm import tqdm

# Create a rate limiter with a limit of 1800 requests per
minute
seconds_per_job = 1 / (1800 / 60)
```

```
    try:
        annotation = detect_safe_search(image_path)

        if annotation:
            return
convert_annotation_to_safety(safe_search_annotation=annotatio
        else:
            return None
    except Exception:
        return None


# Process images using ThreadPool
is_safe_values_cloud_vision = []
with ThreadPoolExecutor() as executor:
    futures = []
    for img_url in tqdm(image_paths_all,
total=len(image_paths_all), position=0):
        futures.append(executor.submit(process_image,
img_url))
        time.sleep(seconds_per_job)

    for future in futures:
        is_safe_values_cloud_vision.append(future.result())

# Set Nones to False
```

```
is_safe_values_cloud_vision
]

# Print number of safe images found
print(
    f"Safe images =
{np.array(is_safe_values_cloud_vision).sum()} out of
{len(is_safe_values_cloud_vision)} images"
)
```

Output:

```
100%|██████████| 1000/1000 [00:34<00:00, 29.22it/s]
Safe images = 804 out of 1000 images
```

4. Filter a list of images based on their safety classification using the Cloud Vision

API.

```
# Filter images by safety
metadatas = [
    metadata
    for metadata, is_safe in zip(metadatas,
```

```
]
image_names = [
    image_name
    for image_name, is_safe in zip(image_names_all,
is_safe_values_cloud_vision)
    if is_safe
]
image_paths = [
    image_path
    for image_path, is_safe in zip(image_paths_all,
is_safe_values_cloud_vision)
    if is_safe
]
```

Defining encoding functions

5. Create an `EmbeddingPredictionClient` which encapsulates the logic to call the embedding API.

```
import base64
import time
import typing
```

```
import numpy as np

from google.cloud import aiplatform
from google.protobuf import struct_pb2


class EmbeddingResponse(typing.NamedTuple):
    text_embedding: typing.Sequence[float]
    image_embedding: typing.Sequence[float]


def load_image_bytes(image_uri: str) -> bytes:
    """Load image bytes from a local URI."""
    # This lab only uses local files, so simplify.
    with open(image_uri, "rb") as image_file:
        image_bytes = image_file.read()
    return image_bytes


class EmbeddingPredictionClient:
    """Wrapper around Prediction Service Client."""

    def __init__(
        self,
        project: str,
        location: str = "us-central1",
        api_regional_endpoint: str = "us-central1
```

```
        client_options = {"api_endpoint":
api_regional_endpoint}
        self.client =
aiplatform.gapic.PredictionServiceClient(
            client_options=client_options
        )
        self.location = location
        self.project = project

    def get_embedding(self, text: str = None, image_file:
str = None):
        if not text and not image_file:
            raise ValueError("At least one of text or
image_file must be specified.")

        # Load image file if provided
        image_bytes = None
        if image_file:
            image_bytes = load_image_bytes(image_file)

        instance = struct_pb2.Struct()
        if text:
            instance.fields["text"].string_value = text

        if image_bytes:
            encoded_content =
```

```
instance.fields["image"].struct_value

image_struct.fields["bytesBase64Encoded"].string_value =
encoded_content

        instances = [instance]
        endpoint = (

f"projects/{self.project}/locations/{self.location}"

"/publishers/google/models/multimodalembedding@001"
        )
        response = self.client.predict(endpoint=endpoint,
instances=instances)

        text_embedding = None
        if text:
            # Check if textEmbedding field exists and is
not None
```

```
            if "textEmbedding" in response.predictions[0]
and response.predictions[0]["textEmbedding"]:
                text_emb_value = response.predictions[0]
["textEmbedding"]
                text_embedding = [v for v in
text_emb_value]
```

```
        if image_bytes:
            # Check if imageEmbedding field exists and is
not None
            if "imageEmbedding" in response.predictions[0]
and response.predictions[0]["imageEmbedding"]:
                image_emb_value = response.predictions[0]
["imageEmbedding"]
                image_embedding = [v for v in
image_emb_value]

        return EmbeddingResponse(
            text_embedding=text_embedding,
image_embedding=image_embedding
        )
```

Task 7. Create helper functions to process data in batches

memory using a generator.

```
import time
from concurrent.futures import ThreadPoolExecutor
from typing import Callable, Generator, List, Optional
from tqdm.auto import tqdm # Use tqdm.auto for notebook
compatibility
from tenacity import retry, stop_after_attempt, wait_fixed
# Import wait_fixed

def generate_batches(
    inputs: List[str], batch_size: int
) -> Generator[List[str], None, None]:
    """
    Generator function that takes a list of strings and a
batch size, and yields batches of the specified size.
    """
    for i in range(0, len(inputs), batch_size):
        yield inputs[i : i + batch_size]

ACTUAL_API_RATE_RPM = 10
API_ITEMS_PER_SECOND = ACTUAL_API_RATE_RPM / 60

def encode_to_embeddings_chunked(
    process_function: Callable[[List[str]],
```

```
    batch_size: int = 1, # Multimodal embedding model
currently supports batch_size=1
) -> List[Optional[List[float]]]:
    """
    Function that encodes a list of strings (or image
paths) into embeddings using a process function.
    It takes a list of items and returns a list of optional
lists of floats.
    The data is processed in chunks with rate limiting.
    """
    embeddings_list: List[Optional[List[float]]] = []

    # Prepare the batches using a generator (batch size is
1 for this model)
    batches = generate_batches(items, batch_size)

    # Calculate sleep time between batches based on the
*actual* API rate
    seconds_per_batch = batch_size / API_ITEMS_PER_SECOND
if API_ITEMS_PER_SECOND > 0 else 0 # Avoid division by zero

    print(f"Processing {len(items)} items in batches of
{batch_size}...")
    print(f"Rate limit: {ACTUAL_API_RATE_RPM} RPM
({API_ITEMS_PER_SECOND:.2f} req/sec)")
    print(f"Sleeping for {seconds_per_batch:.2f} seconds
```

```
    with ThreadPoolExecutor(max_workers=5) as executor: #
Limit concurrent workers
        futures = []
        # Use tqdm for progress bar
        for batch in tqdm(batches, total=len(items) //
batch_size + (len(items) % batch_size > 0), desc="Encoding
Batches"):
            # Note: The multimodalembedding model often
processes 1 item per request
            # The batch argument here is typically a list
of length 1 for this model.

futures.append(executor.submit(process_function, batch))
            time.sleep(seconds_per_batch) # Respect the
rate limit

        # Collect results
        for future in tqdm(futures, desc="Collecting
```

```
            embeddings_list.extend(future.result())

    return embeddings_list
```

functions in try-except and retry logic

2. This particular embedding model can only process 1 image at a time, so inputs are validated to be equal to a length of 1.

```
import copy
from typing import List, Optional

@retry(reraise=True, stop=stop_after_attempt(3),
wait=wait_fixed(5))
def encode_texts_to_embeddings_with_retry(text: List[str])
-> List[List[float]]:
    # Assert that the input list has only one item, as
required by the function
    assert len(text) == 1,
"encode_texts_to_embeddings_with_retry expects a list with
a single text item."

    try:
        # Call the embedding client for a single text input
        embedding_response =
client.get_embedding(text=text[0], image_file=None)
        if embedding_response.text_embedding is None:
            # This might happen if the API didn't return a
text. {text[0]} ")
            return [None]
        return [list(embedding_response.text_embedding)] #
Return as List[List[float]]
    except Exception as e:
        # Catch specific API errors if needed, but general
exception is okay for lab
        print(f"Error getting text embedding for
'{text[0]}': {e}")
        raise RuntimeError(f"Error getting text embedding:
{e}") # Re-raise to trigger retry

def encode_texts_to_embeddings(text: List[str]) ->
List[Optional[List[float]]]:
    # This function handles the retry wrapper and returns
[None] on final failure
    if not text:
        return []
    try:
        results = []
        for t in text:
            # Note: We call
encode_texts_to_embeddings_with_retry with a list
containing one item
            embedding_result =
encode_texts_to_embeddings_with_retry(text=[t])
        return results
    except Exception as ex:
        print(f"Failed after retries for texts: {text}.
Error: {ex}")
        return [None for _ in range(len(text))] # Return
None for all inputs on final failure


@retry(reraise=True, stop=stop_after_attempt(3),
wait=wait_fixed(5))
def encode_images_to_embeddings_with_retry(image_uris:
List[str]) -> List[List[float]]:
    # Assert that the input list has only one item, as
required by the function
    assert len(image_uris) == 1,
"encode_images_to_embeddings_with_retry expects a list with
a single image uri."

    try:
        # Call the embedding client for a single image
input
        embedding_response =
client.get_embedding(text=None, image_file=image_uris[0])
        if embedding_response.image_embedding is None:
            # This might happen if the API didn't return an
image embedding for some reason
            return [None]
        return [list(embedding_response.image_embedding)] #
Return as List[List[float]]
    except Exception as ex:
        print(f"Error getting image embedding for
'{image_uris[0]}': {ex}")
        raise RuntimeError(f"Error getting image embedding:
{ex}") # Re-raise to trigger retry


def encode_images_to_embeddings(image_uris: List[str]) ->
List[Optional[List[float]]]:
    # This function handles the retry wrapper and returns
[None] on final failure
    if not image_uris:
        return []
    try:
        results = []
        for uri in image_uris:
            # Note: We call
```

```
encode_images_to_embeddings_with_retry with a list
containing one item
            embedding_result =
encode_images_to_embeddings_with_retry(image_uris=[uri])
            results.extend(embedding_result) # extend
because retry returns a list of results
        return results
```

```
{image_uris}. Error: {ex}")
        return [None for _ in range(len(image_uris))] #
Return None for all inputs on final failure


# Initialize the client here after defining the class
client = EmbeddingPredictionClient(project=PROJECT_ID,
location=REGION)
```

Test the encoding function

3. Encode a subset of data and see if the embeddings and distance metrics make sense.

4. Since there is no public paper describing the embedding model, assume that the embeddings are trained using cosine similarity as a loss function since that is quite common.

```
%%time
# Encode a SAMPLE subset of images for testing.
TEST_SAMPLE_SIZE = 50
```

```
print(f"Encoding a test sample of {TEST_SAMPLE_SIZE}
images...")

# Use the chunked encoder which includes the rate limiter
image_embeddings_test = encode_to_embeddings_chunked(
    process_function=encode_images_to_embeddings,
    items=image_paths_test_sample,
    batch_size=1 # Multimodal embedding model requires
batch size 1
)

# Keep only non-None embeddings from the test sample
indexes_to_keep_test, image_embeddings_test_filtered = zip(
    *[
        (index, embedding)
        for index, embedding in
enumerate(image_embeddings_test)
        if embedding is not None
    ]
) if any(e is not None for e in image_embeddings_test) else
([], []) # Handle case where all are None


print(f"Processed {len(image_embeddings_test_filtered)}
test embeddings successfully")
```

```
{len(image_embeddings_test_filtered[0])}")
else:
    print("No test embeddings were successfully
generated.")
```

Output:

```
Encoding a test sample of 50 images...
Processing 50 items in batches of 1...
Rate limit: 10 RPM (0.17 req/sec)
Sleeping for 6.00 seconds between batches.
Encoding Batches: 100% ████████████████████ 50/50 [05:00<00:00,  6.01s/it]
Collecting Results: 100% ████████████████████ 50/50 [00:00<00:00, 3970.60it/s]
Processed 50 test embeddings successfully
First test embedding dimension: 1408
CPU times: user 1.09 s, sys: 271 ms, total: 1.36 s
Wall time: 5min
```

5. Run the below code snippet in the next cell to calculate the dot-product distance between a text embedding and a set of image embeddings. This type of calculation is common in similarity or matching tasks, where you want to measure the similarity between two vectors.

```
import numpy as np
```

```
) -> np.ndarray:
    """Compute dot-product distance between text and image
embeddings by taking the dot product"""
    # Ensure inputs are numpy arrays
    text_embedding = np.asarray(text_embedding)
    image_embeddings = np.asarray(image_embeddings)

    if text_embedding.ndim == 1:
        text_embedding = text_embedding.reshape(1, -1)

    if image_embeddings.ndim == 1:
        image_embeddings = image_embeddings.reshape(1, -1)
    elif image_embeddings.ndim > 2:
        raise ValueError("image_embeddings must be 1D or 2D
array.")

    distances = np.dot(text_embedding,
image_embeddings.T).squeeze()

    return np.atleast_1d(distances)
```

6. Run the following code snippet in the next cell, to create a visualization to display the top images based on their distances from a given text query.

```
from io import BytesIO
import matplotlib.pyplot as plt
from PIL import Image # Import Image

text_query = "Birds in flight"

# Calculate text embedding of query (uses 1 API call)
text_embedding_query_list =
encode_texts_to_embeddings(text=[text_query])

if not text_embedding_query_list or
text_embedding_query_list[0] is None:
    print(f"Failed to get embedding for query:
'{text_query}'. Cannot perform search.")
else:
    text_embedding_query = text_embedding_query_list[0]
    print(f"Successfully generated embedding for query:
'{text_query}'")

    if len(image_embeddings_test_filtered) > 0:
        print(f"Calculating distances for
{len(image_embeddings_test_filtered)} test embeddings.")
        distances = dot_product_distance(
            text_embedding=np.array(text_embedding_query),

image_embeddings=np.array(image_embeddings_test_filtered) #
```

```
        image_paths_test_filtered =
[image_paths_test_sample[i] for i in indexes_to_keep_test]

        MAX_IMAGES_DISPLAY = 12

        sorted_data = sorted(
            zip(image_paths_test_filtered, distances),
key=lambda x: x[1], reverse=True
        )[:MAX_IMAGES_DISPLAY]

        # Calculate the number of rows and columns needed
to display the images
        num_cols_display = 4
        num_rows_display = math.ceil(len(sorted_data) /
num_cols_display)


        # Create a grid of subplots to display the images
        fig, axs = plt.subplots(nrows=num_rows_display,
ncols=num_cols_display, figsize=(10, num_rows_display * 3))
        axs = axs.flatten() # Flatten the 2D array of axes
for easy iteration

        # Loop through the top images and display them
        for i, (image_path, distance) in
```

```
            try:
                # Load image - we know these are local
files
                image =
Image.open(image_path).convert('RGB') # Convert to RGB to
handle grayscale/palette

                # Display the image
                ax.imshow(image)

                # Set the title
                ax.set_title(f"Rank {i+1}, Distance =
{distance:.2f}", fontsize=8) # Reduced font size

                # Remove ticks
                ax.set_xticks([])
                ax.set_yticks([])
            except Exception as e:
                print(f"Could not load or display image
{image_path}: {e}")
                ax.set_title("Error Loading Image",
fontsize=8)
                ax.set_xticks([])
                ax.set_yticks([])
                ax.text(0.5, 0.5, 'Error',
```

```
        # Hide any unused subplots
        for j in range(i + 1, len(axs)):
            fig.delaxes(axs[j])


        # Adjust spacing and display
        plt.tight_layout() # Use tight_layout for better
spacing
        plt.show()

    else:
        print("No valid image embeddings available from the
test sample to visualize search.")
```

7. Save the dimension size for later usage when creating the index.

```
DIMENSIONS = len(image_embeddings_test_filtered[0])
```

```
print(f"Embedding Dimensions: {DIMENSIONS}")
```

8. The data must be formatted in JSONL format, which means each embedding dictionary is written as an individual JSON object on its own line.

   See more information in the docs at Input data format and structure.

9. Run the following code in the next available cells, to create a temporary file to store embeddings in JSON format.

```
import tempfile
import os

embeddings_file =
tempfile.NamedTemporaryFile(suffix=".json", delete=False)
embeddings_file.close()

embeddings_file_path = embeddings_file.name

print(f"Created temporary file path with .json suffix:
{embeddings_file_path}")
```

```
import json
```

```
image_names_for_indexing =
image_names_all[:IMAGES_FOR_INDEXING]
image_paths_for_indexing =
image_paths_all[:IMAGES_FOR_INDEXING]

print(f"\nEncoding {len(image_paths_for_indexing)} images
for the index...")

embeddings_for_indexing = encode_to_embeddings_chunked(
    process_function=encode_images_to_embeddings,
    items=image_paths_for_indexing,
    batch_size=1
)

print(f"\nWriting embeddings to {embeddings_file_path}...")
successful_embeddings_count = 0
with open(embeddings_file_path, "w") as f:
    for i, embedding in enumerate(embeddings_for_indexing):
        if embedding is not None:
            image_name = image_names_for_indexing[i]
            embedding_formatted = [float(value) for value
in embedding]
            json_entry = {
                "id": str(image_name),
                "embedding": embedding_formatted
            }
            f.write(json.dumps(json_entry) + "\n")
            successful_embeddings_count += 1

print(f"Finished encoding and writing. Successfully
generated and wrote {successful_embeddings_count}
embeddings.")
if successful_embeddings_count == 0:
    print("WARNING: No embeddings were successfully
generated. Check the API calls above for errors (quota?).")
```

Note: This code snippet takes around 20 minutes to run.

Output:

```
Encoding 200 images for the index...
Processing 200 items in batches of 1...
Rate limit: 10 RPM (0.17 req/sec)
Sleeping for 6.00 seconds between batches.
Encoding Batches: 100% 200/200 [20:01<00:00,  6.01s/it]
Collecting Results: 100% 200/200 [00:00<00:00, 9648.95it/s]

Writing embeddings to /var/tmp/tmpl1kl86an.json...
Finished encoding and writing. Successfully generated and wrote 200 embeddings.
```

10. Upload the training data to Cloud Storage.

```
UNIQUE_FOLDER_NAME = "embeddings_folder_unique"
EMBEDDINGS_INITIAL_URI = f"
{BUCKET_URI}/{UNIQUE_FOLDER_NAME}/"
! gsutil cp {embeddings_file_path} {EMBEDDINGS_INITIAL_URI}
```

Output:

```
Copying file:///var/tmp/tmpl1kl86an.json [Content-
Type=application/json]
/ [1 files][  4.0 MiB/  4.0 MiB]
Operation completed over 1 objects/4.0 MiB.
```

Task 8. Create MatchingEngineIndex

1. Run the following code in the next cells to create an index configurations based on certain parameters and configurations. For information on configuration

```
DISPLAY_NAME = "multimodal_diffusiondb"
DESCRIPTION = "Multimodal DiffusionDB Embeddings"
```

```
aiplatform.init(project=PROJECT_ID, location=REGION,
staging_bucket=BUCKET_URI)
```

```
tree_ah_index =
aiplatform.MatchingEngineIndex.create_tree_ah_index(
    display_name=DISPLAY_NAME,
    contents_delta_uri=EMBEDDINGS_INITIAL_URI,
    dimensions=DIMENSIONS,
    approximate_neighbors_count=150,
    distance_measure_type="COSINE_DISTANCE",
    leaf_node_embedding_count=500,
    leaf_nodes_to_search_percent=7,
    description=DESCRIPTION,
    project=PROJECT_ID,
    location=REGION
)
```

Output:

2. Run the following code in next cell to print the resource name of the `MatchingEngineIndex` object (tree_ah_index).

```
INDEX_RESOURCE_NAME = tree_ah_index.resource_name
print(INDEX_RESOURCE_NAME)
```

Output:

```
projects/511733582037/locations/us-central1/indexes/1487639232380928000
```

3. Using the resource name, you can retrieve an existing MatchingEngineIndex.

```
tree_ah_index =
aiplatform.MatchingEngineIndex(index_name=INDEX_RESOURCE_NAME
```

Task 9. Create an IndexEndpoint with VPC Network

1. Run the following code snippet in the next cell, to retrieve the project number and construct the full path to a VPC (Virtual Private Cloud) network.

```
# Retrieve the project number
PROJECT_NUMBER = !gcloud projects list --
filter="PROJECT_ID:'{PROJECT_ID}'" --
format='value(PROJECT_NUMBER)'
PROJECT_NUMBER = PROJECT_NUMBER[0]

VPC_NETWORK = "default"
VPC_NETWORK_FULL =
"projects/{}/global/networks/{}".format(PROJECT_NUMBER,
VPC_NETWORK)
VPC_NETWORK_FULL
```

Create an MatchingEngineIndexEndpoint

2. Run the following code snippet in the next cell, to create a Matching Engine Index Endpoint.

```
my_index_endpoint =
aiplatform.MatchingEngineIndexEndpoint.create(
    display_name=DISPLAY_NAME,
    description=DISPLAY_NAME,
    public_endpoint_enabled=False,
    network=VPC_NETWORK_FULL,
)
```

Task 10. Deploy Indexes

```
DEPLOYED_INDEX_ID = "deployed_index_id_unique"
```

```
my_index_endpoint = my_index_endpoint.deploy_index(
    index=tree_ah_index,
deployed_index_id=DEPLOYED_INDEX_ID
)

my_index_endpoint.deployed_indexes
```

Note: This code snippet takes around 25-30 minutes to run.

Output:

Task 11. Create Online Queries

1. After you've built your indexes, you may query against the deployed index to find nearest neighbors.

```
text_query = "New York skyline"

try:
    text_embedding_response =
encode_texts_to_embeddings(text=[text_query])

    if text_embedding_response and
isinstance(text_embedding_response, list) and
text_embedding_response[0] is not None:
        query_vector = text_embedding_response[0]
        query_successful = True
        print("Text query encoded successfully.")
        query_successful = False
        print("Failed to encode text query.")

except Exception as e:
    print(f"An error occurred during text query encoding:
{e}")
    query_vector = None
    query_successful = False
```

```
search_results = []
search_successful = False

if query_successful:
    NUM_NEIGHBORS = 20
    try:
        response = my_index_endpoint.match(
            deployed_index_id=DEPLOYED_INDEX_ID,
            queries=[query_vector],
            num_neighbors=NUM_NEIGHBORS,
        )
        if response and isinstance(response, list) and
len(response) > 0 and isinstance(response[0], list):
            search_results = response[0]
            search_successful = True
            print("Search failed: No results or unexpected
response format.")

    except Exception as e:
        print(f"Search failed: API error - {e}")

else:
    print("Search skipped: Query encoding failed.")
```

2. Plot the response and verify that images match the text query.

```
import matplotlib.pyplot as plt
from PIL import Image
import math
import os

if search_successful and search_results:
    print("\nVisualizing search results...")

    try:
```

```
        sorted_results = sorted(search_results, key=lambda
x: x.distance, reverse=True)
    except Exception as e:
        print(f"Error sorting results: {e}")
```

```
    MAX_IMAGES_DISPLAY_SEARCH = 12
    images_to_display = []

    if 'image_directory' in globals() and image_directory:
        for result in
sorted_results[:MAX_IMAGES_DISPLAY_SEARCH]:
            try:
                image_id = result.id
                score = result.distance
                original_image_path =
os.path.join(image_directory, image_id)

                if os.path.exists(original_image_path):

images_to_display.append((original_image_path, score))
                else:
                    images_to_display.append((None, score))

            except Exception as e:
                print(f"Error processing result for
display: {result} - {e}")
                images_to_display.append((None, None))

    else:
        print("Visualization skipped: 'image_directory'
```

```
    if images_to_display:
        num_cols_display_search = 4
        num_rows_display_search =
math.ceil(len(images_to_display) / num_cols_display_search)

        try:
            fig, axs =
plt.subplots(nrows=num_rows_display_search,
ncols=num_cols_display_search, figsize=(10,
num_rows_display_search * 3))

            if num_rows_display_search == 1 and
num_cols_display_search == 1:
                axs = [axs]
            else:
                axs = axs.flatten()

            for i, (image_path, score) in
enumerate(images_to_display):
                ax = axs[i]

                if image_path and
os.path.exists(image_path):
                    try:
                        image =
Image.open(image_path).convert('RGB')
```

```
{score:.4f}" if score is not None else f"Rank {i+1}"
                        ax.set_title(title, fontsize=8)
                    except Exception as e:
                        ax.set_title(f"Rank {i+1}\nError
Loading", fontsize=8)
                        ax.text(0.5, 0.5, 'Load Error',
horizontalalignment='center', verticalalignment='center',
transform=ax.transAxes, color='red')
                        print(f"Plotting Error: Could not
load/display image {image_path} - {e}")

                else:
                    title = f"Rank {i+1}\nScore =
{score:.4f}" if score is not None else f"Rank {i+1}"
                    ax.set_title(f"{title}\n(Image Not
Found)", fontsize=8)
                    ax.text(0.5, 0.5, 'Missing',
horizontalalignment='center', verticalalignment='center',
transform=ax.transAxes)
                    ax.set_facecolor('#e0e0e0')

                ax.set_xticks([])
                ax.set_yticks([])
                ax.set_xlabel('')
                ax.set_ylabel('')
```

```
            plt.tight_layout()
            plt.show()

        except Exception as e:
            print(f"Plotting failed: Error setting up plot
- {e}")

elif search_successful and not search_results:
    print("Search successful, but no results returned to
visualize.")
else:
    print("Skipping visualization because search was not
successful.")
```

Congratulations!

You've learned to create text-to-image embeddings using the DiffusionDB dataset and the Vertex AI Multimodal Embeddings model. Your embeddings are now successfully uploaded to the Vertex AI Vector Search service, providing efficient similarity searches.

Manual Last Updated May 01, 2025

Lab Last Tested May 01, 2025