

Enhancing AI Chatbots with Multimodal Capabilities Using ChromaDB and OpenAI CLIP

Introduction

A picture is worth a thousand words, so when we are building AI Agents or chatbots, if we enhance them with multimodal capabilities, users will have both text and images to understand the response easily. In this blog, we're going to use ChromaDB as our multimodal vector database with OpenAI CLIP embedding to vectorize both text and images and save them in the same vector DB. I'll demonstrate how to query text-to-image.

When building a vector store, we need to add **IDs** for each image, image **URIs**, and **captions**. ChromaDB depends on these captions to build images with semantic search capabilities.

Chroma supports multi-modal embedding functions, which can be used to embed data from multiple modalities into a **single embedding space**.

Chroma has the **OpenCLIP** embedding function built in, which supports both text and images.

Embeddings: Bridging Images and Text with Multimodal Learning

CLIP embeddings are numeric representations of images and text that enable comparison between the two. Developed by OpenAI, CLIP (Contrastive Language-Image Pretraining) uses separate image and text encoders to learn a shared embedding space, where related images and texts are brought closer together. The model is trained to align these two modalities, allowing it to determine if an image and text match by measuring the proximity of their respective embeddings.

Work

CLIP trains an image encoder and a text encoder to create a shared space for both images and text. The model optimizes the embeddings so that related images and texts are positioned closer together, while unrelated pairs are pushed further apart. This allows CLIP to compare images and text for similarity and relevance.

Embeddings Are Used For

- **Image Captioning:** CLIP can generate captions for images by matching textual inputs to image embeddings.
- **Similarity Search:** CLIP embeddings allow for efficient image similarity search, where images can be compared directly based on their embeddings.
- **Multimodal Retrieval-Augmented Generation (RAG):** CLIP-based embeddings can be used to enhance question-answering systems by providing additional context from images.

CLIP Embeddings

- **Abstract Tasks:** CLIP may struggle with more abstract or systematic tasks that require deeper contextual understanding.
- **Generalization:** CLIP may not generalize well to images outside of its pre-training dataset.
- **Sensitivity to Wording:** CLIP's zero-shot classifiers can be sensitive to slight changes in wording or phrasing, affecting performance.

So, CLIP embeddings serve as a powerful tool for **connecting images and text in a unified space**, enabling efficient comparisons and a range of multimodal applications.

Let's dive into the code that implements our multimodal AI system using ChromaDB and OpenAI CLIP:

```
# Import necessary libraries
import chromadb # ChromaDB: A vector database for storing and querying embeddings
from chromadb.utils.embedding_functions.open_clip_embedding_function import
OpenCLIPEmbeddingFunction # OpenCLIP embedding function from ChromaDB
from transformers import CLIPProcessor, CLIPModel # CLIP model and processor (imported but
not used directly in this script)
from matplotlib import pyplot as plt # Matplotlib for displaying images
import os # To interact with the file system (used for image folder management)
import torch # PyTorch for tensor computations (imported but not used directly in this
script)

# Initialize Chroma client with persistent storage
# This initializes the ChromaDB client which allows the creation of vector collections
vectore_db_client = chromadb.PersistentClient(path="sreeni-multi-embeddings") # Specify a
path for persistent storage

# Use Chroma's default embedding function (OpenCLIP model)
# This is a pre-built embedding function that will be used to convert documents into
embeddings for storage
default_ef = OpenCLIPEmbeddingFunction()

# Create or get the collection named "sreeni_albums"
# If the collection already exists, it is retrieved. If not, it will be created.
multi_embedding_db = vectore_db_client.get_or_create_collection(
    name="sreeni_albums", # Name of the collection in ChromaDB
    embedding_function=default_ef # Define the embedding function to use
)

# Captions for the images - Descriptive texts about each image to be added as metadata
captions = [
    'Captain - A leader in a heroic pose, symbolizing strength and courage.',
    'Gita - Lord Krishna teaching Arjuna the Bhagavad Gita on the battlefield of
Kurukshetra.',
    'Krishna - A divine depiction of Lord Krishna, representing love and wisdom.',
    'Mari - A scene from the Tamil movie "Mariyadhai," starring Vijayakanth in a dual role
as father and son.',
    'Ramana - Vijayakanth as a college professor who leads an Anti-Corruption Force to
fight against corruption in society.',
    'Ulvanmagan - A dramatic moment from the Tamil movie "Ulvanmagan," highlighting
emotions and family values.',
    'Vanathapole - A serene scene from the Tamil movie "Vanathapole," emphasizing rural
life and relationships.',
    'With Modi - An image of a significant moment featuring Prime Minister Narendra Modi.'
]

# Define the path to the image folder containing images to be processed
# The images folder must be located in the same directory as the script
image_folder = os.path.join(os.path.dirname(__file__), 'images')
```

```

# Create empty lists to store information about each image (ID, filename, metadata)
ids = []
documents = []
metadatas = []

# Loop through all files in the image folder
# Filter out non-image files (only .jpg, .jpeg, .png)
for idx, filename in enumerate(os.listdir(image_folder)):
    if filename.endswith(('.jpg', '.jpeg', '.png')): # Only process image files
        ids.append(str(idx)) # Use the index as a unique identifier for each image
        documents.append(filename) # Store the filename as the document content

        # Add metadata with the caption for the image; if no caption, use 'Unknown'
        caption = captions[idx] if idx < len(captions) else 'Unknown'
        metadatas.append({
            'item_id': str(idx), # Store the unique ID
            'img_category': 'sreeni_albums', # Image category (used for organization)
            'item_name': caption # Caption associated with the image
        })

# Add all the processed items to the Chroma collection
multi_embedding_db.add(
    ids=ids, # List of image IDs
    documents=documents, # List of filenames (image document contents)
    metadatas=metadatas # List of metadata for each image (captions)
)

# Output the total count of items in the collection to verify successful addition
print(f"Total items in collection: {multi_embedding_db.count()}")

# Define a function to display query results along with images
# It will print out relevant metadata and display the image
def print_query_results(query_list: list, query_results: dict) -> None:
    for i, query in enumerate(query_list): # Loop over each query in the list
        print(f'Results for query: {query}')
        found_results = False

        # Loop over the results for the current query
        for j in range(len(query_results['ids'][i])):
            id = query_results["ids"][i][j]
            distance = query_results['distances'][i][j]
            document = query_results['documents'][i][j]
            metadata = query_results['metadatas'][i][j]

            # Check if the query is found in the item's metadata (case-insensitive)
            if query.lower() in metadata['item_name'].lower():
                found_results = True
                print(f'id: {id}, distance: {distance}, metadata: {metadata}, document:
{document}')

        # Construct full image path and display it using Matplotlib
        img_path = os.path.join(image_folder, document)
        if os.path.exists(img_path):
            img = plt.imread(img_path) # Load the image
            plt.imshow(img) # Display the image
            plt.axis("off") # Hide axis labels
            plt.show() # Show the image
        else:

```

```

        print(f"Image not found at path: {img_path}") # Handle missing image
files

# If no results found for the query, output this information
if not found_results:
    print(f"No results found for query: {query}")

# Define the list of queries to search for in the Chroma collection
query_texts = [ 'professor'] # Example query looking for 'professor'

# Query the Chroma vector database with the queries
query_results = multi_embedding_db.query(
    query_texts=query_texts, # List of query texts to search for
    n_results=len(ids), # Retrieve as many results as there are items in the collection
    include=['documents', 'distances', 'metadatas'] # Include relevant fields in the query
    result
)

# Call the function to print and display query results
print_query_results(query_texts, query_results)

```

****Here is my query , I am looking for Movie Album Actor Vijayakanth acted as professor.**

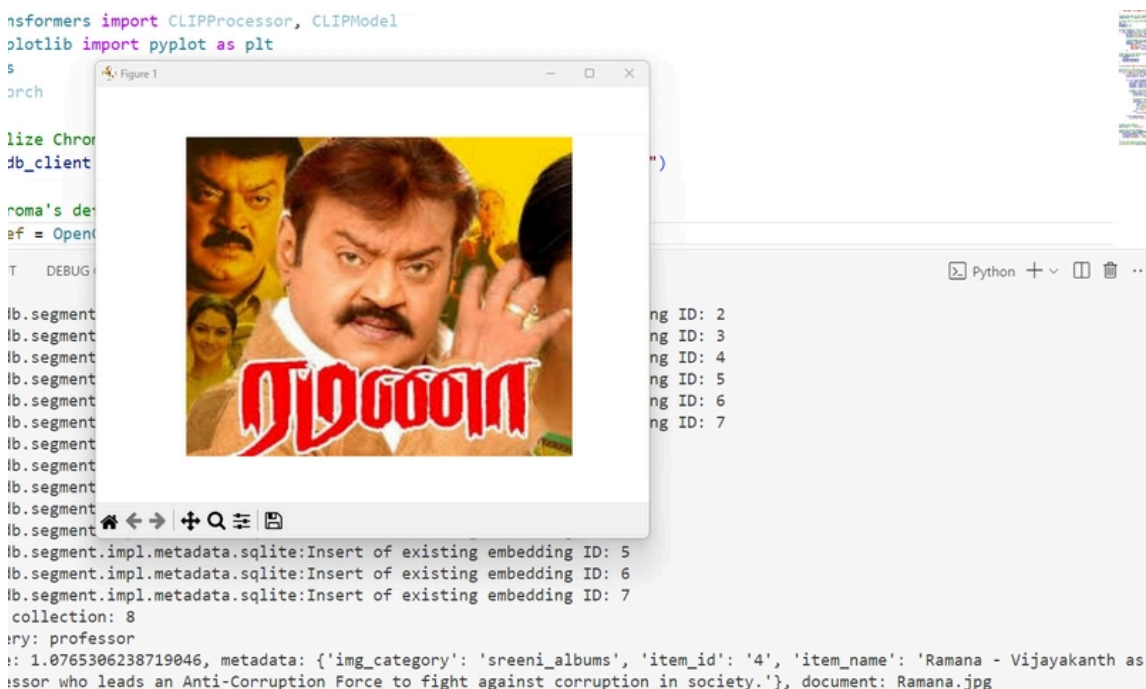
`query_texts = ['professor'] # Example query looking for 'professor'`

Output: Image and associated metadata

```

Total items in collection: 8
Results for query: professor
id: 4, distance: 1.0765306238719046, metadata: {'img_category': 'sreeni_albums', 'item_id': '4', 'item_name': 'Ramana - Vijayakanth as
a college professor who leads an Anti-Corruption Force to fight against corruption in society.'}, document: Ramana.jpg

```



The screenshot shows a Jupyter Notebook environment. On the left, a code editor contains Python code that imports necessary libraries, defines a query, and uses ChromaDB to search for images. The code includes comments and function calls. On the right, the console displays the output of the query, showing the search results for the query 'professor'. The output includes the item ID, distance, metadata, and the document name. A large image of the movie poster for 'Ramana' is displayed in the center of the notebook, showing the actor Vijayakanth in a white shirt with a red sash, set against a yellow background with a red border. The word 'Ramana' is written in large red letters at the bottom of the poster.

Key Components

1. **ChromaDB:** We use ChromaDB as our vector database to store and query embeddings.
2. **OpenCLIP Embedding Function:** This function generates embeddings for both text and images using the CLIP model.

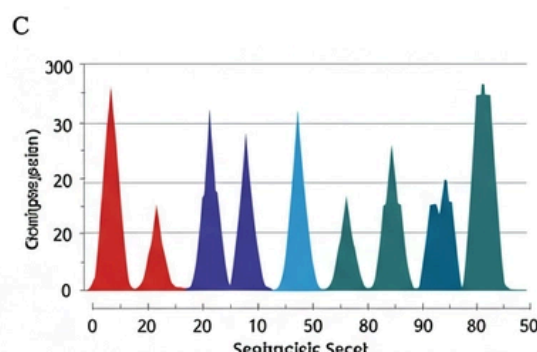
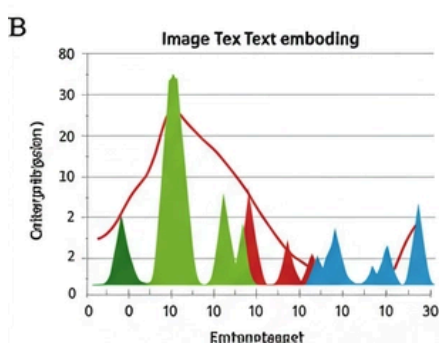
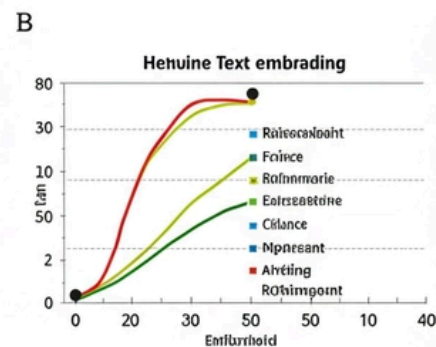
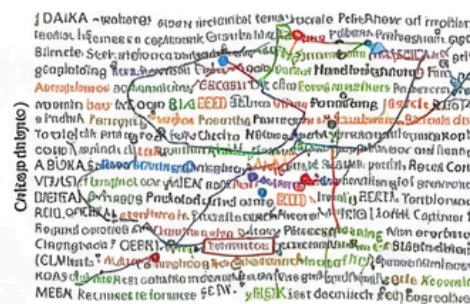
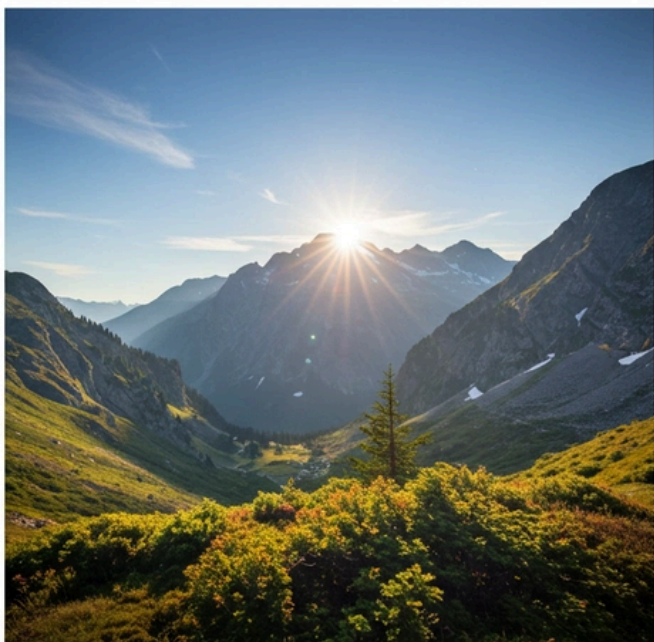
3. **Matplotlib:** Used for displaying the retrieved images.

How It Works

1. We initialize a ChromaDB client with persistent storage.
2. We create a collection using the OpenCLIP embedding function.
3. We define captions for our images and process the image files in the specified folder.
4. We add the images, their IDs, and metadata to the ChromaDB collection.
5. We define a function to query the database and display results.
6. Finally, we perform a query and display the results.

Here are 3-4 potential use cases or applications:

Multimodal re Embleading



1. Image-Based Search Engine for a Collection

- **Application:** Imagine you have a large collection of images and each image has a descriptive caption (like movie scenes, artwork, product images, etc.). This code can be used to create a search engine where users can search for images based on keywords from the captions. For example, searching for "professor" would return images of the "Ramana" movie scene where Vijayakanth plays a college professor.
- **Use Case:** Useful for media libraries, movie databases, or e-commerce platforms where customers can search for images based on descriptions or tags.

2. Content-Based Image Retrieval (CBIR) System

- **Application:** The system allows users to find similar images based on the content of a query (e.g., a text description). For example, if you have a library of artwork images, a user could input a query like "love and wisdom" to find images depicting similar themes. The embeddings of the images and their metadata (captions) enable efficient similarity-based searches.
- **Use Case:** This is highly beneficial for art galleries, stock photo websites, or collections of visual media, enabling users to find images that match the content of their queries even if they don't know the exact image name.

3. Educational or Training Content Search

- **Application:** This code can be used to build an image repository with educational content, like images related to historical events, scientific concepts, or literature. For example, teachers or students can search for "Krishna" and get relevant images related to the deity's depictions across different contexts, helping them better visualize and understand the subject matter.
- **Use Case:** This could be a great tool for schools, educational content libraries, or museums to allow for searching and interacting with educational materials in a visually enhanced manner.

4. Personalized Image Recommendations Based on Descriptions

- **Application:** This system could be adapted for recommending images from a user's personal collection based on their preferences or search history. For example, if a user frequently searches for images related to leadership or heroism (like the "Captain" description), the system could recommend more images that contain similar themes.
- **Use Case:** Personalized recommendation systems for galleries, social media platforms, or photo storage apps. This can be a great way to surface new content that aligns with a user's interests based on semantic descriptions rather than just tags or metadata.

In Summary:

- **Image Search Engine** for collections of images (e.g., movies, art, or products).
- **Content-Based Image Retrieval (CBIR)** for finding similar images based on textual descriptions.
- **Educational Content Search** for images in learning environments.
- **Personalized Image Recommendations** based on query history or preferences.

These applications can be customized for specific domains where image content needs to be easily searchable or recommended based on descriptive metadata.

Conclusion

This implementation demonstrates how to create a powerful multimodal AI system using ChromaDB and OpenAI's CLIP model. By leveraging these tools, we can build robust systems capable of handling both text-to-image and image-to-text queries seamlessly. This opens up exciting possibilities for building more intuitive and engaging AI applications across various domains.

As you experiment with these multimodal capabilities, you'll discover new ways to enhance user interactions and create more powerful AI agents and chatbots. The future of AI is multimodal - start exploring these capabilities today!