Guide to Chroma DB | A Vector Store for Your Generative AI LLMs

ALGORITHM AUDIO DATABASE IMAGE INTERMEDIATE PYTHON

Introduction

Generative Large Language Models like GPT, PaLM, etc are trained on large amounts of data. These models do not take the texts from the dataset as it is, because computers do not understand text, they only understand numbers. Embeddings are the representation of the text but in a numerical format. All the information to and from the Large Language Models is through these embeddings. Accessing these embeddings directly is time-consuming. Hence, what is called Vector Databases stores these embeddings specifically designed for efficient storage and retrieval of vector embeddings. In this guide, we will focus on one such vector store/database, Chroma DB, which is widely used and open-source.

Learning Objectives

- Generating embeddings with ChromaDB and Embedding Models
- Creating collections within the Chroma Vector Store
- Storing documents, images, and embeddings within the collections
- Performing Collection Operations like deleting and updating data, renaming of Collections
- Finally, querying the collections to extract relevant information

This article was published as a part of the Data Science Blogathon.

Table of contents

Short Introduction to Embeddings

Embeddings or Vector Embeddings is a way of representing data (be it text, images, audio, videos, etc) in the numerical format, to be precise it's a way of representing data in the form of numbers in an n-dimensional space(a numerical vector). This way, embeddings allow us to cluster similar data together. There are models, that take these inputs and convert them into vectors. One such example is the Word2Vec, which is a popular embedding model developed by Google, that converts words to vectors(vectors are points having n-dimensions). All the Large Language Models have their respective embedding models, which create embeddings for their LLM.

What are Embeddings Used for?

The good thing about converting words to vectors is we can compare them. A computer cannot compare two words as they are, but if we give them in the form of numerical inputs, i.e. vector embeddings it can compare them. We can create a cluster of words having similar embeddings. The words King, Queen, Prince, and Princess will appear in a cluster because they are related to other.

This way embeddings allow us to get find words similar to a given word. We can incorporate this into sentences, where we input a sentence and obtain the related sentences from the provided data. This is the base for Semantic Search, Sentence Similarity, Anomaly Detection, chatbot, and many more use cases. The Chatbots we build to perform Question Answering from a given PDF, Doc, leverage this very concept of embeddings. All the Generative Large Language Models use this approach to get similarly related content to the queries provided to them.

Vector Store and the Need for Them

As discussed, embeddings are representations of any kind of data usually, the unstructured ones in the numerical format in an n-dimensional space. Now where do we store them? Traditional RDMS (Relational Database Management Systems) cannot be used to store these vector embeddings. This is where the Vector Store / Vector Dabases come into play. Vector Databases are designed to store and retrieve vector embeddings in an efficient manner. There are many Vector Stores out there, which differ by the embedding models they support and the kind of search algorithm they use to get similar vectors.

Why do we need them? We need them because they provide fast access to the data we need. Let's consider a Chatbot based on a PDF. Now when a user enters a query, the first thing will be to fetch related content from PDF to that query and feed this information to the Chatbot. So that the Chatbot can take this information related to the query and proved the relevant answer to the User. Now how do we get the relevant content from PDF related to the User query? The answer is a simple similarity search

When data is represented in vector embeddings, we can find similarities between different parts of the data and extract the data similar to a particular embedding. The query is first converted to embeddings by

an embedding model and then the Vector Store takes this vector embedding and then performs a similarity search (through search algorithms) between other embeddings that it has stored in its database and fetches all the relevant data. These relevant vector embeddings are then passed to the Large Language Model which is the chatbot that uses this information to generate a final answer to the User.

What is Chroma DB?

Chroma is a Vector Store / Vector DB by the company Chroma. Chroma DB like many other Vector Stores out there, is for storing and retrieving vector embeddings. The good part is that Chroma is a Free and Open Source project. This gives other skilled developers out there in the world the to give suggestions and make tremendous improvements to the Database and even one can expect a quick reply to an issue when dealing with Open Source software, as the whole Open Source community is out there to see and resolve that issue.

At present Chroma does not provide any hosting services. Store the data locally in the local file system when creating applications around Chroma. Though Chroma is planning to build a hosting service in the near future. Chroma DB offers different ways to store vector embeddings. You can store them In-memory, you can save and load them In-memory, you can just run Chroma a client to talk to the backend server. Overall Chroma DB has only 4 functions in the API, thus making it short, simple, and easy to get started with.

How does Chroma DB work?

Here are the steps describing how Chroma DB works:

1. **Data Structure**: Chroma DB organizes chromatic data in a structured format optimized for efficient storage and retrieval.
2. **Storage**: It stores color-related information such as RGB values, color names, and associated metadata in the database.
3. **Indexing**: Chroma DB creates indexes to facilitate fast lookup of colors based on various criteria like RGB values, color names, or other attributes.
4. **Querying**: Users can query Chroma DB using specific criteria such as color codes, names, or properties to retrieve relevant color information.
5. **Analysis**: Chroma DB enables analysis of color data for various applications such as image processing, design, and color matching.
6. **Optimization**: The database is optimized for speed and efficiency, allowing for quick retrieval and processing of color-related information.
7. **Integration**: It can be integrated into different software applications and platforms to provide color-related functionalities seamlessly.
8. **Continued Improvement**: Chroma DB may undergo updates and improvements to enhance its capabilities and accommodate evolving requirements in color management and analysis.

Let's Start with Chroma DB

In this section, we will install Chroma and see all the functionalities it provides. Firstly, we will install the library through the pip command

```
$ pip install chromadb
```

Chroma Vector Store API

This will download the Chroma Vector Store API for Python. With this package, we can perform all tasks like storing the vector embeddings, retrieving them, and performing a semantic search for a given vector embedding.

```
import chromadb from chromadb.config import Settings client = chromadb.Client(Settings(chroma_db_impl="duckdb+parquet", persist_directory="/content/" ))
```

Memory Database

We will start off with creating a persistent in-memory database. The above code will create one for us. To create a client we take the **Client()** object from the Chroma DB. Now to create an in-memory database, we configure our client with the following parameters

- **chroma_db_impl** = “duckdb+parquet”
- **persist_directory** = “/content/”

This will create an in-memory DuckDB database with the parquet file format. And we provide the directory for where this data is to be stored. Here we are saving the database in the /content/ folder. So whenever we connect to a Chroma DB client with this configuration, the Chroma DB will look for an existing database in the directory provided and will load it. If it is not present then it will create it. And when we close the connection, the data will be saved to this directory.

Now, we will create a collection. Collection in Vector Store is where we save the set of vector embeddings, documents, and any metadata if present. Collection in a vector database can be thought of as a Table in Relational Database.

Create Collection and Add Documents

We will now create a collection and add documents to it.

```
collection = client.create_collection("my_information") collection.add( documents=["This is a document containing car information", "This is a document containing information about dogs", "This document contains four wheeler catalogue"], metadatas=[{"source": "Car Book"},{"source": "Dog Book"},{'source':'Vechile Info'}], ids=["id1", "id2", "id3"] )
```

- Here we start by creating a collection first. Here we name the collection “**my_information**”.
- To this collection, we will be adding documents. Here we are adding 3 documents, in our case, we are just adding three sentences as three documents. The first document is about cars, the second one is about dogs and the final one is about four-wheelers.
- We are even adding the metadata. Metadata for all three documents is provided.
- Every document needs to have a unique ID to it, hence we are giving id1, id2, and id3 to them
- All these are like the variables to the **add()** function from the collection
- After running the code, add these documents to our collection “**my_information**”

Vector Databases

We learned that the information stored in Vector Databases is in the form of Vector Embeddings. But here, we provided text/text files i.e. documents. So how does it store them? Chroma DB by default, uses an **all-MiniLM-L6-v2** vector embedding model to create the embeddings for us. This model will take our documents and convert them into vector embeddings. If we want to work with a specific embedding function like other sentence-transformer models from HuggingFace or OpenAI embedding model, we can specify it under the **embeddings_function=embedding_function_name** variable name in the **create_collection()** method.

We can also provide embeddings directly to the Vector Store, instead of passing the documents to it. Just like the document parameter in **create_collection**, we have an **embedding** parameter, to which we pass on the embeddings that we want to store in the Vector Database.

So now the model has successfully stored our three documents in the form of vector embeddings in the vector store. Now, we will look at retrieving relevant documents from them. We will pass a query and will fetch the documents that are relevant to it. The corresponding code for this will be

```
results = collection.query( query_texts=["Car"], n_results=2 ) print(results)
```

Query a Vector Store

- To query a vector store, we have a **query()** function provided by the collections which lets us query the vector database for relevant documents. In this function, we provide two parameters
- **query_texts** – To this parameter, we give a list of queries for which we need to extract the relevant documents.
- **n_results** – This parameter specifies how many top results should the database return. In our case we want our collection to return 2 top most relevant documents related to the query
- When we run and print the results, we get the following output

```
{'ids': [['id1', 'id3']],
 'embeddings': None,
 'documents': [['This is a document containing car information',
   'This document contains four wheeler catalogue']],
 'metadatas': [[{'source': 'Car Book'}, {'source': 'Vechile Info'}]],
 'distances': [[1.0784918069839478, 1.5855050086975098]]}
```

We see that the vector store returns two documents associated with **id1** and **id3**. The **id1** is the document about cars and the **id3** is the document amount four wheelers, which is related to a car again. So when we gave a **query**, the Chrom DB converts the query into a vector embedding with the embedding model we provided at the start. Then this vector embedding performs a semantic search(similar nearest neighbors) on all the available documents. The query here "**car**" is most relevant to the id1 and id3 documents, hence we get the following result for the query.

This is very helpful when we are trying to build a chat application that includes multiple documents. Through a vector store, we can fetch the relevant documents to the provided query by performing a semantic search and feeding only these documents to the final Generative AI model, which will then take these relevant documents and generate a response to the provided query.

Updating and Deleting Data

Not always do we add all the information at once to the Vector Store. In most cases, we have only limited data/documents at the start, which we add as is to the Vector Store. Later in point of time, when we get more data, it becomes necessary to update the existing data/vector embeddings present in the Vector Store. To update data in Chroma DB, we do the following

```
collection.update( ids=["id2"], documents=["This is a document containing information about Cats"], metadatas=[{"source": "Cat Book"}], )
```

Previously, the information in the document associated with **id2** was about Dogs. Now we are changing it to Cats. For this information to be updated within the Vector Store, we pass the id of the document, the updated document, and the updated metadata of the document to the **update()** function of the collections. This will now update the **id2** to Cats which was previously about Dogs.

Query in Database

```
results = collection.query( query_texts=["Felines"], n_results=1 ) print(results)
```

```
{'ids': [['id2']],
 'embeddings': None,
 'documents': [['This is a document containing information about Cats']],
 'metadatas': [[{'source': 'Cat Book'}]],
 'distances': [[0.9560527801513672]]}
```

We pass in Felines as the query to the Vector Store. Cats belong to the family of mammals called Felines. So the collection must return the Cat document as the relevant document to us. In the output, we get to see exactly the same. The vector store was able to perform a semantic search between the query and the contents of the documents and was able to return the perfect document to the query provided.

Upset Function

There is a similar function to the update function called the **upsert()** function. The only difference between both the **update()** and **upsert()** function is, if the document ID specified in the **update()** function does not exist, the **update()** function will raise an error. But in the case of the **upsert()** function, if the document ID doesn't exist in the collection, then it will be added to the collection similar to the **add()** function.

Sometimes, to reduce the space or remove unnecessary/ unwanted information, we might want to delete some documents from the collection in the Vector Store.

```
collection.delete(ids = ['id1']) results = collection.query( query_texts=["Car"], n_results=2 ) print(results)
```

```
{'ids': [['id3', 'id2']], 'embeddings': None, 'documents': [['This document
contains four wheeler catalogue', 'This is a document containing information about
Cats']], 'metadatas': [[{'source': 'Vechile Info'}, {'source': 'Cat Book'}]],
'distances': [[1.5855050086975098, 1.7295209169387817]]}
```

Delete Function

To delete an item from a collection, we have the **delete()** function. In the above, we are deleting the first document associated with **id1** which was about cars. Now to check, we query the collection with the "**car**"

as the query and then see the results. We see that only 2 documents **id2** and **id3** appear, where the **id2** is the document about four wheelers which are closest to cars and **id3** is the document about cats which is the least closest to cars, but as we specified **n_results** = 2 we get the **id3** as well. If we do not specify any variables to the **delete()** function, then all the items will be deleted from that collection

Collection Functions

We have seen how to create a new collection and then add documents, and embeddings to it. We have even seen how to extract relevant information to a query from the collection i.e. from the documents stored in the Vector Store. The collections object from Chroma DB is also associated with many other useful functions.

Let us look at some other functionalities provided by Chroma DB:

```
new_collections = client.create_collection("new_collection") new_collections.add( documents=["This is Python Documentation", "This is a Javascript Documentation", "This document contains Flast API Cheatsheet"], metadatas=[{"source": "Python For Everyone"}, {"source": "JS Docs"}, {'source':'Everything Flask'}], ids= ["id1", "id2", "id3"] ) print(new_collections.count()) print(new_collections.get())
```

```
(3,
 {'ids': ['id1', 'id2', 'id3'],
  'embeddings': None,
  'documents': ['This is Python Documentation',
   'This is a Javascript Documentation',
   'This document contains Flast API Cheatsheet'],
  'metadatas': [{'source': 'Python For Everyone'},
   {'source': 'JS Docs'},
   {'source': 'Everything Flask'}]})
```

Count Function

The **count()** function from the collections returns the number of items present in the collection. In our case, we have 3 documents stored in our collection, hence the output will be 3. Coming to the **get()** function, it will return all the items that are present in our collection along with the **metadata**, **ids**, and **embeddings** if any. In the output, we see that all the items that we have to our collection have to get through the **get()** command. Let's now look at modifying the collection name

```
collection.modify(name="new_collection_name")
```

Modify Function

Use the **modify()** function from collections to change the name of the collection that was given at the start of collection creation. When run, change the collection name from the old name that was defined at the start to the new name provided in the **modify()** function under the name variable. Now suppose, we have multiple collections in our Vector Store. How to work on a specific collection, that is how to get a specific collection from the Vector Store and how to delete a specific collection? Let's see this

```
my_collection = client.get_collection(name="my_information_2")
client.delete_collection(name="my_information_2")
```

Get Collection Function

The **get_collection()** function will fetch an existing collection provided the **name**, from the Vector Store. If the provided collection does not exist, then the function will raise an error for the same. Here the **get_collection()** will try to get the **my_information_2** collection and assign it to the variable **my_collection**. To delete an existing collection, we have the **delete_collection()** function, which takes the collection name as the parameter (**my_information** in this case) and then deletes it, if it exists.

Conclusion

In this guide, we have seen how to get started with Chroma, one of the Open Source Vector Databases. We initially started with learning what are vector embeddings, why they are necessary for the Generative AI models, and how Vector Stores help these Generative Large Language Models. Then we deep-dived into Chroma, and we have seen how to create collections in Chroma. Then we looked into how to add data like documents to Chroma and how the Chroma DB creates vector embeddings out of them. Finally, we have seen how to retrieve relevant information related to the given query from a particular collection present in the Vector Store.

Some of the key takeaways from this guide include:

- Vector Embeddings are numerical representations (numerical vectors) of non-numerical data like text, images, audio, etc
- Vector Stores are the databases that are used to store the vector embeddings in the form of collections
- They provide efficient storage and retrieval of information from the embeddings data
- Chroma DB can work as both an in-memory database and as a backend
- Chroma DB has the functionality to store the data upon quitting and load the data to memory upon initiating a connection, thus persisting the data
- With Vector Stores, extracting information from documents, generating recommendations, and building chatbot applications will become much simpler

Frequently Asked Questions

Q1. What are Vector Databases / Vector Stores?

A. Vector Databases are the place where vector embeddings are stored. These exist because they provide efficient retrieval of vector embeddings. They are used for extracting relevant information for the query from their database through semantic search.

Q2. What are Vector Embeddings?

A. Vector Embeddings are representations of text/image/audio/videos in a numerical format in an n-dimensional space, typically as a numerical vector. This is done because computers do not understand text or images or any other non-numerical data natively. So these embeddings allow them to understand the data well because this is presented in a numerical format.

Q3. What are Embedding Models?

A. Embedding models are the ones that turn non-numerical data like text/images into a numerical format that is vector embeddings. Chroma DB by default uses the all-MiniLM-L6-v2 model to create embeddings. Apart from these models, there are many other ones like Googles's Word2Vec, OpenAI Embedding model, other Sentence Transformers from HuggingFace, and many more.

Q4. Where might these embedding vectors/vector databases be used?

A. These Vector Stores find their applications in almost everything that involves Generative AI models. Like extracting information from documents, generating images from given prompts, building a recommendation system, clustering relevant data together, and much more.

Article Url - https://www.analyticsvidhya.com/blog/2023/07/guide-to-chroma-db-a-vector-store-for-your-generative-ai-llms/

Ajay Kumar Reddy