Hands-on Lab: Similarity Search on Text Using a Chroma Vector Database

Skills Network

Estimated time: 30 minutes

What you will learn

In this lab, you will work with implementing text similarity search using embeddings in Chroma DB, a database system designed for managing and querying text data efficiently.

You will learn how to create a collection and populate the database with a set of sample text documents, where each document is associated with a unique ID and its corresponding embedding generated using the SentenceTransformer embedding function. Subsequently, you will perform a similarity search on a specified query term, retrieving the top three documents that exhibit the highest similarity to the query term.

Learning objectives

After completing this lab, you will be able to:

- Integrate Chroma DB into applications for storing and querying text data.
- Generate text embeddings for text documents using Chroma DB's SentenceTransformer embedding function.
- Implement similarity search functionality to compute similarity scores between query embeddings and document embeddings.
- Display top-ranked documents similar to a given query term.

About Skills Network Cloud IDE

Skills Network Cloud IDE provides an environment for hands-on labs for course and project-related labs. It is an open source IDE (Integrated Development Environment).

Important notice about this lab environment

Please be aware that sessions for this lab environment are not persisted. Every time you connect to this lab, a new environment is created for you. Any data you may have saved in the earlier session will be lost. Plan to complete these labs in a single session to avoid losing your data.

Prerequisites

- Basic knowledge of vector databases
- Understanding of Chroma vector databases
- Use the Set up a Chroma DB environment lab before completing this lab's tasks. Remember to keep your terminal window open.

Task 1: Set up a Chroma DB environment

1. On the main menu, select the **Terminal** tab and then select **New Terminal**.

2. Now open a new terminal and install `chromadb` in the environment to work with the Chroma DB vector database. Execute the following command in the terminal window and press **Enter**.

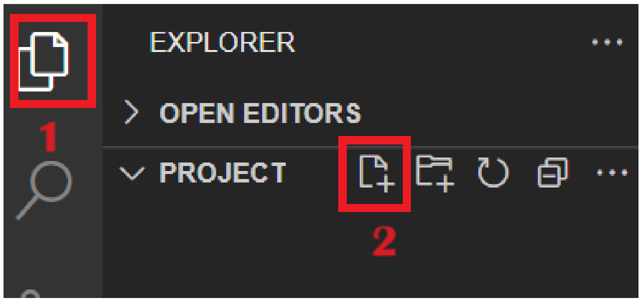
```
pip install chromadb==1.0.12
```

3. Then install one more dependency by executing the following command in the same terminal window where you installed the previous dependency.

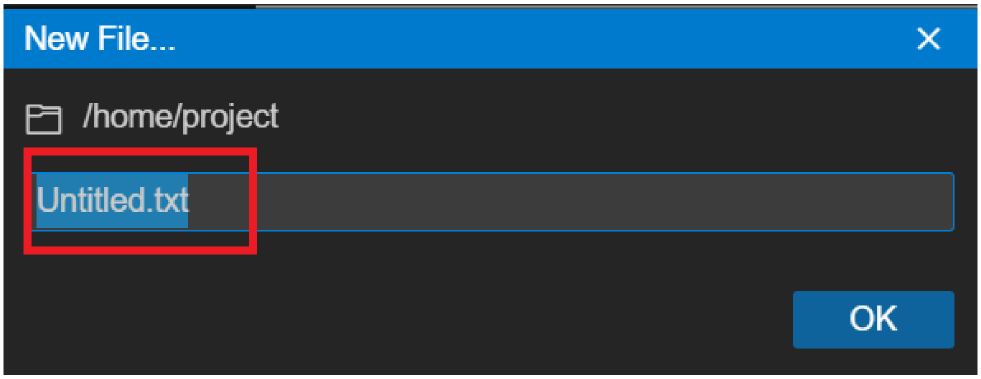
   ```
   pip install sentence-transformers==4.1.0
   ```

   Note: Do not close the terminal where Docker is running.

4. Select **Explorer** on the left side of the terminal window, as shown at number 1 in the following screenshot. Then, within the **Project** folder, select **New File** as shown at number 2 in the following screenshot.



5. After performing the above step, a popup box displays with default file name **Untitled.txt** as shown in the following screenshot. Replace its default name with `similarity_search.py`.



Task 2: Create a Collection and Embed Data

Now, you will learn how to create a collection and embed data.

1. First, import the Chroma DB library and embedding functions from the `chromadb` package. By importing these classes, you can create an instance of the client that interacts with the Chroma DB database and define the embedding model. Include the following command in your Python file.

   ```
   # Importing the necessary modules from the chromadb package:
   # chromadb is used to interact with the Chroma DB database,
   # embedding_functions is used to define the embedding model
   import chromadb
   from chromadb.utils import embedding_functions
   ```

```
# Define the embedding function using SentenceTransformers
ef = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="all-MiniLM-L6-v2"
)
```

The code imports necessary modules from Chroma DB, including the main `chromadb` library for interacting with the database and `embedding_functions` to define the embedding model using SentenceTransformers.

2. Then create a Chroma DB client and initialize the `collection_name` variable, specifying the name of the collection.

```
# Create a new instance of ChromaClient to interact with the Chroma DB
client = chromadb.Client()
# Define the name for the collection to be created or retrieved
collection_name = "my_grocery_collection"
```

3. Create the main function, which will contain the code for interacting with the Chroma DB database.

```
# Define the main function to interact with the Chroma DB
def main():
    try:
        # Place your database operations inside this block
        pass
    except Exception as error:  # Catch any errors and log them to the console
        print(f"Error: {error}")
```

4. Then create a collection inside the database and perform the following command inside the `try` block of the `main` function.

```
# Create a collection in the Chroma database with a specified name,
# distance metric, and embedding function. In this case, we are using
# cosine distance
collection = client.create_collection(
    name=collection_name,
    metadata={"description": "A collection for storing grocery data"},
    configuration={
        "hnsw": {"space": "cosine"},
        "embedding_function": ef
    }
)
print(f"Collection created: {collection.name}")
```

   - This code creates a collection named "my_grocery_collection" from the database using the client's `create_collection` method.
   - This method creates a new collection in the database with the specified name and configuration.
   - The collection variable holds a reference to this collection for further operations and is configured to use the SentenceTransformer embedding function to generate embeddings for text data.

5. Next, after creating a collection inside the `try` block, define the sample data. The following array contains sample text strings for grocery items.

```
# Array of grocery-related text items
texts = [
    'fresh red apples',
    'organic bananas',
    'ripe mangoes',
    'whole wheat bread',
    'farm-fresh eggs',
    'natural yogurt',
    'frozen vegetables',
    'grass-fed beef',
    'free-range chicken',
    'fresh salmon fillet',
    'aromatic coffee beans',
    'pure honey',
    'golden apple',
    'red fruit'
]
```

6. Now, generate unique IDs for each text string in the array by including the following command after the text array:

```
# Create a list of unique IDs for each text item in the 'texts' array
 # Each ID follows the format 'food_<index>', where <index> starts from 1
ids = [f"food_{index + 1}" for index, _ in enumerate(texts)]
```

   - This code generates unique IDs for the documents based on their index in the `texts` array. Each ID follows the format `"food_<index>"`, where the `<index>` starts with the number 1.

Task 3: Generate Embeddings and Add Texts to a Collection

1. Next, add the IDs and texts to the collection using the following code:

```
# Add documents and their corresponding IDs to the collection
# The `add` method inserts the data into the collection
# The documents are the actual text items, and the IDs are unique identifiers
# ChromaDB will automatically generate embeddings using the configured embedding function
collection.add(
    documents=texts,
    metadatas=[{"source": "grocery_store", "category": "food"} for _ in texts],
    ids=ids
)
```

- By passing the documents, metadata, and IDs using the `add` method of the collection, the code adds the documents along with their corresponding information to the Chroma DB database collection.
- This code supports efficient storage and retrieval of text data, along with their automatically generated embeddings, to facilitate natural language processing tasks such as similarity search and document clustering.

2. You can use the following code, which uses the `get` method, to access all items.

```
# Retrieve all the items (documents) stored in the collection
# The `get` method fetches all data from the collection
all_items = collection.get()
# Log the retrieved items to the console for inspection
# This will print out all the documents, IDs, and metadata stored in the collection
print("Collection contents:")
print(f"Number of documents: {len(all_items['documents'])}")
```

- `all_items` This variable stores the result of the `collection.get()` method. This variable holds the retrieved items from the collection, including the documents, IDs, and metadata.

- `collection.get()` This `get` method retrieves all items stored within the collection. These items include the documents and their corresponding IDs.

- The method returns all data stored in the collection as a dictionary containing the documents, IDs, and metadata.

Task 4: Perform a Similarity Search

1. Use the following code to create a function named `perform_similarity_search` after the main function. You will write your query to find similar items.

```
# Function to perform a similarity search in the collection
def perform_similarity_search(collection, all_items):
    try:
        # Place your similarity search code inside this block
        pass
    except Exception as error:
        print(f"Error in similarity search: {error}")
```

2. Within the `try` block, initialize a variable named `query_term`. Here's an example:

```
# Define the query term you want to search for in the collection
query_term = "apple"
```

3. Then, create the code that calls the `query` method. The query method searches for documents similar to the query term, which also includes the *n_results* parameter. The *n_results* parameter specifies the number of top results to retrieve.

```
# Perform a query to search for the most similar documents to the 'query_term'
results = collection.query(
    query_texts=[query_term],
    n_results=3  # Retrieve top 3 results
)
print(f"Query results for '{query_term}':")
print(results)
```

Task 5: Handle the Results and Display Highly Similar Text

1. Next, create code that handles the results if no similar documents to the query term are found and logs a message to the console.

```
# Check if no results are returned or if the results array is empty
if not results or not results['ids'] or len(results['ids'][0]) == 0:
    # Log a message indicating that no similar documents were found for the query term
    print(f'No documents found similar to "{query_term}"')
    return
```

2. You'll also want to display the top documents whose distance is closest to the query compared to other text data.

```
print(f'Top 3 similar documents to "{query_term}":')
# Access the nested arrays in 'results["ids"]' and 'results["distances"]'
for i in range(min(3, len(results['ids'][0]))):
    doc_id = results['ids'][0][i]  # Get ID from 'ids' array
    score = results['distances'][0][i]  # Get score from 'distances' array
    # Retrieve text data from the results
    text = results['documents'][0][i]
    if not text:
        print(f' - ID: {doc_id}, Text: "Text not available", Score: {score:.4f}')
    else:
        print(f' - ID: {doc_id}, Text: "{text}", Score: {score:.4f}')
```

3. Next, call the `perform_similarity_search` function at the end of the `try` block of the `main` function using the following code:

```
perform_similarity_search(collection, all_items)
```

4. Then call the `main` function to create a collection and generate embeddings.

```
if __name__ == "__main__":
    main()
```

Click on the button below to see the full `similarity_search.py` script. You can copy-paste the solution into your `similarity_search.py` file.

▼ Click for the script

```
# Importing the necessary modules from the chromadb package:
# chromadb is used to interact with the Chroma DB database,
# embedding_functions is used to define the embedding model
import chromadb
from chromadb.utils import embedding_functions
# Define the embedding function using SentenceTransformers
ef = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="all-MiniLM-L6-v2"
)
# Create a new instance of ChromaClient to interact with the Chroma DB
client = chromadb.Client()
# Define the name for the collection to be created or retrieved
collection_name = "my_grocery_collection"
# Define the main function to interact with the Chroma DB
def main():
    try:
        # Create a collection in the Chroma database with a specified name,
        # distance metric, and embedding function. In this case, we are using
        # cosine distance
        collection = client.create_collection(
            name=collection_name,
            metadata={"description": "A collection for storing grocery data"},
            configuration={
                "hnsw": {"space": "cosine"},
                "embedding_function": ef
            }
        )
        print(f"Collection created: {collection.name}")
        # Array of grocery-related text items with professional humor
        texts = [
            'fresh red apples',
            'organic bananas',
            'ripe mangoes',
            'whole wheat bread',
            'farm-fresh eggs',
            'natural yogurt',
            'frozen vegetables',
            'grass-fed beef',
            'free-range chicken',
            'fresh salmon fillet',
            'aromatic coffee beans',
            'pure honey',
            'golden apple',
            'red fruit'
        ]
        # Create a list of unique IDs for each text item in the 'texts' array
        # Each ID follows the format 'food_<index>', where <index> starts from 1
        ids = [f"food_{index + 1}" for index, _ in enumerate(texts)]
        # Add documents and their corresponding IDs to the collection
        # The `add` method inserts the data into the collection
        # The documents are the actual text items, and the IDs are unique identifiers
                # ChromaDB will automatically generate embeddings using the configured embedding function
        collection.add(
            documents=texts,
            metadatas=[{"source": "grocery_store", "category": "food"} for _ in texts],
            ids=ids
        )
        # Retrieve all the items (documents) stored in the collection
        # The `get` method fetches all data from the collection
        all_items = collection.get()
        # Log the retrieved items to the console for inspection
        # This will print out all the documents, IDs, and metadata stored in the collection
        print("Collection contents:")
        print(f"Number of documents: {len(all_items['documents'])}")

        # Function to perform a similarity search in the collection
        def perform_similarity_search(collection, all_items):
            try:
                # Define the query term you want to search for in the collection
                query_term = "apple"
                # Perform a query to search for the most similar documents to the 'query_term'
                results = collection.query(
                    query_texts=[query_term],
                    n_results=3  # Retrieve top 3 results
                )
                print(f"Query results for '{query_term}':")
                print(results)
                # Check if no results are returned or if the results array is empty
                if not results or not results['ids'] or len(results['ids'][0]) == 0:
                    # Log a message indicating that no similar documents were found for the query term
                    print(f'No documents found similar to "{query_term}"')
                    return

                print(f'Top 3 similar documents to "{query_term}":')
                # Access the nested arrays in 'results["ids"]' and 'results["distances"]'
                for i in range(min(3, len(results['ids'][0]))):
                    doc_id = results['ids'][0][i]  # Get ID from 'ids' array
                    score = results['distances'][0][i]  # Get score from 'distances' array
                    # Retrieve text data from the results
                    text = results['documents'][0][i]
                    if not text:
                        print(f' - ID: {doc_id}, Text: "Text not available", Score: {score:.4f}')
                    else:
                        print(f' - ID: {doc_id}, Text: "{text}", Score: {score:.4f}')
            except Exception as error:
                print(f"Error in similarity search: {error}")

        perform_similarity_search(collection, all_items)
    except Exception as error:  # Catch any errors and log them to the console
        print(f"Error: {error}")
if __name__ == "__main__":
    main()
```

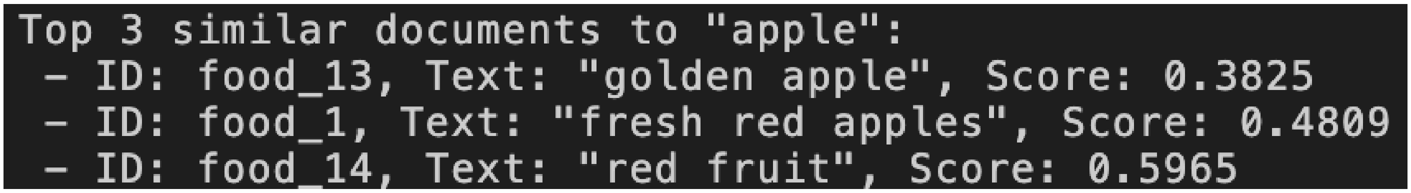
Task 6: Check the output

Next, check the output so that you can verify that the display shows the most similar items or data in the text array.

1. If the terminal window is not already open, open the terminal window. Then type or enter the following command and press **Enter**.

```
python3.11 similarity_search.py
```

2. The output displayed will be similar to the following screenshot where you see your top three documents similar to "apple" and see their IDs, text, and similarity scores.



```
Top 3 similar documents to "apple":
 - ID: food_13, Text: "golden apple", Score: 0.3825
 - ID: food_1, Text: "fresh red apples", Score: 0.4809
 - ID: food_14, Text: "red fruit", Score: 0.5965
```

Congratulations! You have created your first application of similarity search!

Practice Exercises

In this lab session, you generated embeddings for the `texts` array and performed similarity search using the `query_term="apple"` query string. In this set of practice exercises, you will go through the steps needed to perform similarity search on the same `texts` array but using multiple queries at once, such as `query_term=["red", "fresh"]`.

1. Replace `query_term="apple"` with `query_term=["red", "fresh"]` in your current code file. Moreover, write code that converts `query_term` to a list if it is a string.

▼ Click here for the solution

~~~
```python
# Define the query term you want to search for in the collection
query_term = ["red", "fresh"]
if isinstance(query_term, str):
    query_term = [query_term]
```
~~~

2. Change `[query_term]` to `query_term` in the call to `collection.query()`.

▶ Click here for the solution

3. Modify the block that retrieves top documents to allow for multiple search queries.

   Hint: Use an outer loop around the whole block.

▼ Click here for the solution

```
for q in range(len(query_term)):
    print(f'Top 3 similar documents to "{query_term[q]}":')
    # Access the nested arrays in 'results["ids"]' and 'results["distances"]'
    for i in range(min(3, len(results['ids'][q]))):
        doc_id = results['ids'][q][i]  # Get ID from 'ids' array
        score = results['distances'][q][i]  # Get score from 'distances' array
        # Retrieve text data from the results
        text = results['documents'][q][i]
```

```
        if not text:
            print(f' - ID: {doc_id}, Text: "Text not available", Score: {score:.4f}')
        else:
            print(f' - ID: {doc_id}, Text: "{text}", Score: {score:.4f}')
```

4. Verify that the output is correct by running the current file name in the terminal using `python` with file name `similarity_search.py`.

```
python3.11 similarity_search.py
```

Now that you've completed each of the practice exercises, click below to see the entire code flow of `similarity_search.py`.

Click on the button below to see the fully modified `similarity_search.py` script.

▶ Click for the script

Conclusion

In this lab, you learned the following:

- **Chroma DB integration:** The code demonstrates the integration of Chroma DB with Python for similarity search and embeddings. The code imports necessary modules from Chroma DB, creates a client instance, and configures the embedding function using SentenceTransformers.
- **Data handling:** The code creates a collection named "my_grocery_collection" within the Chroma DB client. The code defines sample text documents related to grocery items and adds these text documents to this collection along with their corresponding IDs and metadata. The SentenceTransformer embedding function generates embeddings for the text documents automatically when they are added to the collection.
- **Similarity search:** The code defines a function named "perform_similarity_search" to run a similarity search based on a specified query term. In this case, the code specifies the query term "apple" to retrieve the top three similar text documents from the collection. The function iterates through the results and displays the top three texts that match the query parameter with their similarity scores.

Author(s)

Wojciech "Victor" Fulmyk

Contributor(s)

Richa Arora