Hands-on Lab: Similarity Search on Employee Records using Python and Chroma DB

Estimated time: 60 minutes

Objectives

After completing this lab, you will be able to:

- Create a collection for employee data and store related data in a Chroma DB collection.
- Generate numeric representations for specific key-value pairs of the employee dataset and convert them into vector embeddings.
- Store related data in a Chroma DB collection.
- Perform similarity search on vector embeddings that you create and display similar results based on a query.

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

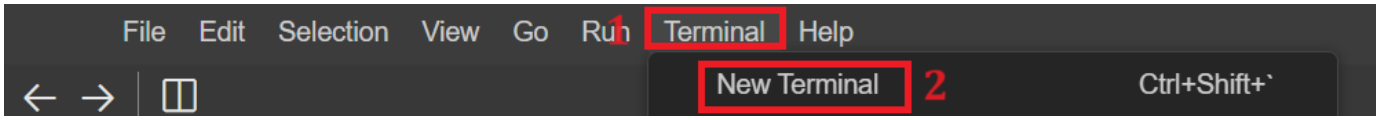
Please be aware that sessions for this lab environment are not persisted. Every time you connect to this lab, a new environment is created for you. Any data you may have saved in an earlier session will be lost. Plan to complete these labs in a single session to avoid losing your data.

Prerequisites

- Basic knowledge of vector embeddings
- An understanding of Chroma vector databases
- Basic understanding of Python programming

Task 1: Set up the Environment

1. In your IDE environment, select the **Terminal** tab at the top-right of the window shown at number 1 in the following screenshot, and then select **New Terminal** from the drop-down menu as shown at number 2 in the same screenshot.



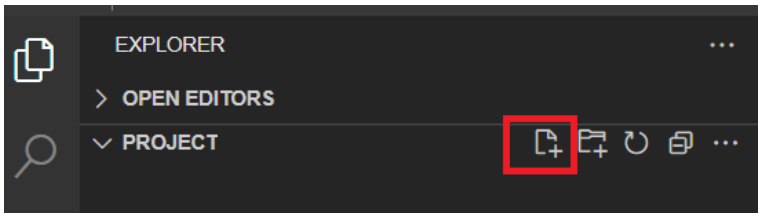
2. Install the required Python packages to work with Chroma DB. First, install `chromadb` in the environment to work with the Chroma DB vector database. Execute the following command in the terminal window and press **Enter**.

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

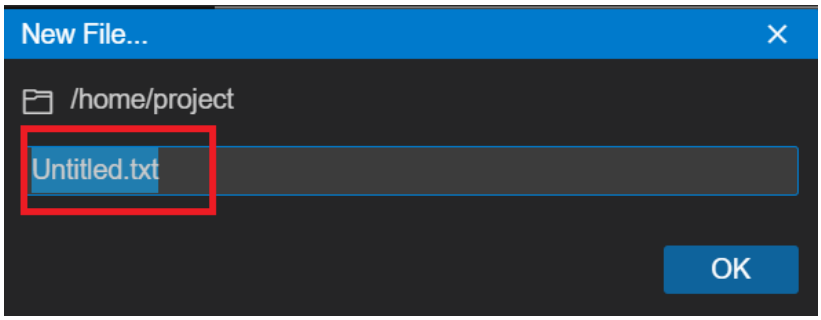
3. Then install the SentenceTransformers dependency by executing the following command in the same terminal window where you installed the previous dependency.

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

4. Now, within the IDE environment, create a Python file. For this, first create a file within **Explorer** by clicking the **New File** symbol highlighted in the red box in the given screenshot.



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



6. Now, let's start creating the code logic to analyze employee data.

Task 2: Create a Collection and Embed Data

In **Task 2** you will create a collection within Chroma DB and embed data within it.

1. First, import the necessary modules 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 the `similarity_employeedata.py` file.

```
# Importing 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
# This function will be used to generate embeddings (vector representations) for the data
ef = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="all-MiniLM-L6-v2"
)
# Creating an instance of ChromaClient to establish a connection with the Chroma database
client = chromadb.Client()
```

   This 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 initialize the `collection_name` variable and specify the name of the collection.

```
# Defining a name for the collection where data will be stored or accessed
# This collection is likely used to group related records, such as employee data
collection_name = "employee_collection"
```

3. Now create the main function, which you must define as a function. This function will contain the code to create collection, generate embeddings and perform similarity search.

```
# Defining a function named 'main'
# This function is used to encapsulate the main operations for creating collections,
# generating embeddings, and performing similarity search
def main():
    try:
        # Code for database operations will be placed here
        # This includes creating collections, adding data, and performing searches
        pass
    except Exception as error:
        # Catching and handling any errors that occur within the 'try' block
        # Logs the error message to the console for debugging purposes
        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.

```
# Creating a collection using the ChromaClient instance
# The 'create_collection' method creates a new collection with the specified configuration
collection = client.create_collection(
    # Specifying the name of the collection to be created
    name=collection_name,
    # Adding metadata to describe the collection
    metadata={"description": "A collection for storing employee data"},
    # Configuring the collection with cosine distance and embedding function
    configuration={
        "hnsw": {"space": "cosine"},
        "embedding_function": ef
    }
)
print(f"Collection created: {collection.name}")
```

- This code creates a collection named "employee_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 of employees. Include the given `employees` array after you have declared the collection.

```
# Defining a list of employee dictionaries
# Each dictionary represents an individual employee with comprehensive information
employees = [
    {
        "id": "employee_1",
        "name": "John Doe",
        "experience": 5,
        "department": "Engineering",
        "role": "Software Engineer",
        "skills": "Python, JavaScript, React, Node.js, databases",
        "location": "New York",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_2",
        "name": "Jane Smith",
        "experience": 8,
        "department": "Marketing",
        "role": "Marketing Manager",
        "skills": "Digital marketing, SEO, content strategy, analytics, social media",
        "location": "Los Angeles",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_3",
        "name": "Alice Johnson",
        "experience": 3,
        "department": "HR",
        "role": "HR Coordinator",
        "skills": "Recruitment, employee relations, HR policies, training programs",
        "location": "Chicago",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_4",
        "name": "Michael Brown",
        "experience": 12,
        "department": "Engineering",
        "role": "Senior Software Engineer",
        "skills": "Java, Spring Boot, microservices, cloud architecture, DevOps",
        "location": "San Francisco",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_5",
        "name": "Emily Wilson",
        "experience": 2,
        "department": "Marketing",
        "role": "Marketing Assistant",
        "skills": "Content creation, email marketing, market research, social media management",
        "location": "Austin",
        "employment_type": "Part-time"
    },
    {
```

```
        "id": "employee_6",
        "name": "David Lee",
        "experience": 15,
        "department": "Engineering",
        "role": "Engineering Manager",
        "skills": "Team leadership, project management, software architecture, mentoring",
        "location": "Seattle",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_7",
        "name": "Sarah Clark",
        "experience": 8,
        "department": "HR",
        "role": "HR Manager",
        "skills": "Performance management, compensation planning, policy development, conflict resolution",
        "location": "Boston",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_8",
        "name": "Chris Evans",
        "experience": 20,
        "department": "Engineering",
        "role": "Senior Architect",
        "skills": "System design, distributed systems, cloud platforms, technical strategy",
        "location": "New York",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_9",
        "name": "Jessica Taylor",
        "experience": 4,
        "department": "Marketing",
        "role": "Marketing Specialist",
        "skills": "Brand management, advertising campaigns, customer analytics, creative strategy",
        "location": "Miami",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_10",
        "name": "Alex Rodriguez",
        "experience": 18,
        "department": "Engineering",
        "role": "Lead Software Engineer",
        "skills": "Full-stack development, React, Python, machine learning, data science",
        "location": "Denver",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_11",
        "name": "Hannah White",
        "experience": 6,
        "department": "HR",
        "role": "HR Business Partner",
        "skills": "Strategic HR, organizational development, change management, employee engagement",
        "location": "Portland",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_12",
        "name": "Kevin Martinez",
        "experience": 10,
        "department": "Engineering",
        "role": "DevOps Engineer",
        "skills": "Docker, Kubernetes, AWS, CI/CD pipelines, infrastructure automation",
        "location": "Phoenix",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_13",
        "name": "Rachel Brown",
        "experience": 7,
        "department": "Marketing",
        "role": "Marketing Director",
        "skills": "Strategic marketing, team leadership, budget management, campaign optimization",
        "location": "Atlanta",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_14",
        "name": "Matthew Garcia",
        "experience": 3,
        "department": "Engineering",
        "role": "Junior Software Engineer",
        "skills": "JavaScript, HTML/CSS, basic backend development, learning frameworks",
        "location": "Dallas",
        "employment_type": "Full-time"
    },
    {
        "id": "employee_15",
        "name": "Olivia Moore",
        "experience": 12,
        "department": "Engineering",
        "role": "Principal Engineer",
        "skills": "Technical leadership, system architecture, performance optimization, mentoring",
        "location": "San Francisco",
        "employment_type": "Full-time"
    },
]
```

Task 3: Generate Embeddings and Add Texts to a Collection

1. Next, create meaningful text documents from employee data that will enable actual similarity searches. Instead of just using experience numbers, we'll create rich text descriptions that combine multiple employee attributes.

```
# Create comprehensive text documents for each employee
# These documents will be used for similarity search based on skills, roles, and experience
employee_documents = []
for employee in employees:
    document = f"{employee['role']} with {employee['experience']} years of experience in {employee['department']}. "
    document += f"Skills: {employee['skills']}. Located in {employee['location']}. "
    document += f"Employment type: {employee['employment_type']}."
    employee_documents.append(document)
```

2. Next, add the IDs, documents, and comprehensive metadata to the collection.

```
# Adding data to the collection in the Chroma database
# The 'add' method inserts or updates data into the specified collection
collection.add(
    # Extracting employee IDs to be used as unique identifiers for each record
    ids=[employee["id"] for employee in employees],
    # Using the comprehensive text documents we created
    documents=employee_documents,
    # Adding comprehensive metadata for filtering and search
    metadatas=[{
        "name": employee["name"],
        "department": employee["department"],
        "role": employee["role"],
        "experience": employee["experience"],
        "location": employee["location"],
        "employment_type": employee["employment_type"]
    } for employee in employees]
)
```

   - By passing these pieces of information using the `add` method of the collection, the code adds the documents along with their corresponding IDs and comprehensive metadata to the Chroma DB database collection.
   - This approach enables both semantic similarity search (through the text documents) and precise filtering (through the metadata), making it suitable for realistic employee search scenarios.

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

```
# Retrieving all items from the specified collection
# The 'get' method fetches all records stored in the collection
all_items = collection.get()
# Logging the retrieved items to the console for inspection or debugging
print("Collection contents:")
print(f"Number of documents: {len(all_items['documents'])}")
```

   - `all_items` This line of code declares a variable named `all_items` to store the result of the `collection.get()` method. This variable holds the items retrieved from the collection, which includes the documents, IDs, and metadata.
   - `collection.get()` This method retrieves all data items stored within the collection. These items include the documents, their corresponding IDs, and their metadata.

Task 4: Perform Similarity Search, Metadata Filtering, and Full-Text Search

1. Next, create a function named `perform_advanced_search` after the `main` function, where you will demonstrate different types of searches including similarity search, metadata filtering, and combinations of both.

```
# Function to perform various types of searches within the collection
def perform_advanced_search(collection, all_items):
    try:
        # Advanced search operations will be placed here
        pass
    except Exception as error:
        print(f"Error in advanced search: {error}")
```

2. Inside this function, implement similarity search using natural language queries. This demonstrates how Chroma DB can find semantically similar content.

```
print("=== Similarity Search Examples ===")
# Example 1: Search for Python developers
print("\n1. Searching for Python developers:")
query_text = "Python developer with web development experience"
results = collection.query(
    query_texts=[query_text],
    n_results=3
)
print(f"Query: '{query_text}'")
for i, (doc_id, document, distance) in enumerate(zip(
    results['ids'][0], results['documents'][0], results['distances'][0]
)):
    metadata = results['metadatas'][0][i]
    print(f"  {i+1}. {metadata['name']} ({doc_id}) - Distance: {distance:.4f}")
    print(f"     Role: {metadata['role']}, Department: {metadata['department']}")
    print(f"     Document: {document[:100]}...")
# Example 2: Search for leadership roles
print("\n2. Searching for leadership and management roles:")
query_text = "team leader manager with experience"
results = collection.query(
    query_texts=[query_text],
    n_results=3
)
print(f"Query: '{query_text}'")
for i, (doc_id, document, distance) in enumerate(zip(
    results['ids'][0], results['documents'][0], results['distances'][0]
)):
    metadata = results['metadatas'][0][i]
    print(f"  {i+1}. {metadata['name']} ({doc_id}) - Distance: {distance:.4f}")
    print(f"     Role: {metadata['role']}, Experience: {metadata['experience']} years")
```

3. Now implement metadata filtering to find employees based on specific criteria.

```
print("\n=== Metadata Filtering Examples ===")
# Example 1: Filter by department
print("\n3. Finding all Engineering employees:")
results = collection.get(
    where={"department": "Engineering"}
)
print(f"Found {len(results['ids'])} Engineering employees:")
for i, doc_id in enumerate(results['ids']):
    metadata = results['metadatas'][i]
    print(f"  - {metadata['name']}: {metadata['role']} ({metadata['experience']} years)")
# Example 2: Filter by experience range
print("\n4. Finding employees with 10+ years experience:")
results = collection.get(
    where={"experience": {"$gte": 10}}
)
print(f"Found {len(results['ids'])} senior employees:")
for i, doc_id in enumerate(results['ids']):
    metadata = results['metadatas'][i]
    print(f"  - {metadata['name']}: {metadata['role']} ({metadata['experience']} years)")
# Example 3: Filter by location
print("\n5. Finding employees in California:")
results = collection.get(
    where={"location": {"$in": ["San Francisco", "Los Angeles"]}}
)
print(f"Found {len(results['ids'])} employees in California:")
```

```
for i, doc_id in enumerate(results['ids']):
    metadata = results['metadatas'][i]
    print(f"  - {metadata['name']}: {metadata['location']}")
```

4. Finally, combine similarity search with metadata filtering for more precise results.

```
print("\n=== Combined Search: Similarity + Metadata Filtering ===")
# Example: Find experienced Python developers in specific locations
print("\n6. Finding senior Python developers in major tech cities:")
query_text = "senior Python developer full-stack"
results = collection.query(
    query_texts=[query_text],
    n_results=5,
    where={
        "$and": [
            {"experience": {"$gte": 8}},
            {"location": {"$in": ["San Francisco", "New York", "Seattle"]}}
        ]
    }
)
print(f"Query: '{query_text}' with filters (8+ years, major tech cities)")
print(f"Found {len(results['ids'][0])} matching employees:")
for i, (doc_id, document, distance) in enumerate(zip(
    results['ids'][0], results['documents'][0], results['distances'][0]
)):
    metadata = results['metadatas'][0][i]
    print(f"  {i+1}. {metadata['name']} ({doc_id}) - Distance: {distance:.4f}")
    print(f"     {metadata['role']} in {metadata['location']} ({metadata['experience']} years)")
    print(f"     Document snippet: {document[:80]}...")
```

Task 5: Handle the Results and Display Highly Similar Text

1. Create code that handles the results if no documents are found that are similar to the query term. This code also logs a message to the console.

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

2. Use the following code to display the top documents whose distance meets the query criteria based on the query data threshold.

```
# Log the header for the top 3 similar documents based on the query term
print(f'Top 3 similar documents to "{query_text}":')
# Loop through the top 3 results and log the document details
for i in range(min(3, len(results['ids'][0]))):
    # Extract the document ID and similarity score from the results
    doc_id = results['ids'][0][i]
    score = results['distances'][0][i]
    # Retrieve the document text corresponding to the current ID from the results
    text = results['documents'][0][i]
    # Check if the text is available; if not, log 'Text not available'
    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_advanced_search` function at the end of the `try` block of the `main` function.

```
# Call the perform_advanced_search function with the collection and all_items as arguments
perform_advanced_search(collection, all_items)
```

4. Then use the `main` function to create a collection, generate embeddings and call the `perform_advanced_search` function, which you included in the preceding instruction.

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

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

▶ Click for the script

Task 6: Check the output

Next, check the output so that you can verify that the advanced search functionality is working correctly across all search types.

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

```
python3.11 similarity_employeedata.py
```

2. Next view the resulting output. The output will display comprehensive search results demonstrating different types of queries and filtering capabilities.

Collection Creation and Document Count:

```
Collection created: employee_collection
Collection contents:
Number of documents: 15
```

Similarity Search Examples:
The first section demonstrates semantic similarity search using natural language queries. Notice how the system understands the meaning behind the words, not just exact matches.

```
=== Similarity Search Examples ===
1. Searching for Python developers:
Query: 'Python developer with web development experience'
  1. John Doe (employee_1) - Distance: 0.5156
     Role: Software Engineer, Department: Engineering
     Document: Software Engineer with 5 years of experience in Engineering. Skills: Python, JavaScript, React, Node...
  2. Matthew Garcia (employee_14) - Distance: 0.5724
     Role: Junior Software Engineer, Department: Engineering
     Document: Junior Software Engineer with 3 years of experience in Engineering. Skills: JavaScript, HTML/CSS, ba...
  3. Alex Rodriguez (employee_10) - Distance: 0.5967
     Role: Lead Software Engineer, Department: Engineering
     Document: Lead Software Engineer with 18 years of experience in Engineering. Skills: Full-stack development, R...
2. Searching for leadership and management roles:
Query: 'team leader manager with experience'
  1. Jane Smith (employee_2) - Distance: 0.5382
     Role: Marketing Manager, Experience: 8 years
  2. Sarah Clark (employee_7) - Distance: 0.5467
```

```
   Role: HR Manager, Experience: 8 years
3. David Lee (employee_6) - Distance: 0.5497
   Role: Engineering Manager, Experience: 15 years
```

Key Observations:

- **Semantic Understanding**: For the Python query, John Doe ranks highest (0.5156) because his skills explicitly include "Python, JavaScript, React, Node.js" - a perfect match for web development. Notice how the system found relevant candidates even though the query didn't exactly match the text.
- **Cross-Department Results**: The leadership search finds managers across different departments (Marketing, HR, Engineering), showing the system understands that "manager" roles exist in various contexts.
- **Ranking Logic**: Distance scores increase (0.5382 → 0.5467 → 0.5497) as semantic similarity decreases, demonstrating how Chroma DB quantifies relevance.
- **Skills vs. Roles**: Matthew Garcia appears in Python results despite his skills being "JavaScript, HTML/CSS" because his role as a "Software Engineer" is semantically related to development work.

Metadata Filtering Examples:
The second section shows precise filtering based on employee attributes. Unlike similarity search, these results are exact matches based on specific criteria.

```
=== Metadata Filtering Examples ===
3. Finding all Engineering employees:
Found 8 Engineering employees:
  - John Doe: Software Engineer (5 years)
  - Michael Brown: Senior Software Engineer (12 years)
  - David Lee: Engineering Manager (15 years)
  - Chris Evans: Senior Architect (20 years)
  - Alex Rodriguez: Lead Software Engineer (18 years)
  - Kevin Martinez: DevOps Engineer (10 years)
  - Matthew Garcia: Junior Software Engineer (3 years)
  - Olivia Moore: Principal Engineer (12 years)
4. Finding employees with 10+ years experience:
Found 6 senior employees:
  - Michael Brown: Senior Software Engineer (12 years)
  - David Lee: Engineering Manager (15 years)
  - Chris Evans: Senior Architect (20 years)
  - Alex Rodriguez: Lead Software Engineer (18 years)
  - Kevin Martinez: DevOps Engineer (10 years)
  - Olivia Moore: Principal Engineer (12 years)
5. Finding employees in California:
Found 3 employees in California:
  - Jane Smith: Los Angeles
  - Michael Brown: San Francisco
  - Olivia Moore: San Francisco
```

Key Observations:

- **Exact Filtering Power**: The Engineering department filter returns exactly 8 out of 15 employees, showing that over half the workforce is in Engineering - a typical distribution for tech companies.
- **Seniority Distribution**: The 10+ years filter reveals 6 senior employees (40% of workforce), indicating a good mix of experience levels. Notice how roles reflect seniority: "Senior," "Lead," "Principal," and "Manager" titles.
- **Role Hierarchy Visible**: In the senior employees list, you can see the career progression: Software Engineer → Senior Software Engineer → Lead/Principal Engineer → Engineering Manager → Senior Architect.
- **Geographic Concentration**: Only 3 employees in California highlight the distributed nature of the workforce, with San Francisco having multiple employees as expected for a tech hub.
- **Perfect Precision**: Unlike similarity search, metadata filtering gives 100% precise results - every employee returned exactly matches the specified criteria.

Combined Search Results:
The final section demonstrates the power of combining similarity search with metadata filtering - this is where Chroma DB truly shines for real-world applications.

```
=== Combined Search: Similarity + Metadata Filtering ===
6. Finding senior Python developers in major tech cities:
Query: 'senior Python developer full-stack' with filters (8+ years, major tech cities)
Found 4 matching employees:
  1. Michael Brown (employee_4) - Distance: 0.6726
     Senior Software Engineer in San Francisco (12 years)
     Document snippet: Senior Software Engineer with 12 years of experience in Engineering. Skills: Jav...
  2. Chris Evans (employee_8) - Distance: 0.7537
     Senior Architect in New York (20 years)
     Document snippet: Senior Architect with 20 years of experience in Engineering. Skills: System desi...
  3. David Lee (employee_6) - Distance: 0.8344
     Engineering Manager in Seattle (15 years)
     Document snippet: Engineering Manager with 15 years of experience in Engineering. Skills: Team lea...
```

```
   4. Olivia Moore (employee_15) - Distance: 0.8761
      Principal Engineer in San Francisco (12 years)
      Document snippet: Principal Engineer with 12 years of experience in Engineering. Skills: Technical...
```

Key Observations:

- **Filtered Pool**: From 15 total employees, only 4 meet both the semantic similarity AND the exact criteria (8+ years in major tech cities), demonstrating how filters narrow down results effectively.
- **Semantic Ranking within Constraints**: Michael Brown ranks highest (0.6726) because his Java/Spring Boot skills are semantically closest to "senior Python developer full-stack" among the filtered candidates.
- **Interesting Discoveries**: Chris Evans (Senior Architect) ranks second despite having "System design, distributed systems" skills rather than Python - showing the system understands that senior architects often have full-stack capabilities.
- **Geographic + Experience Filtering**: All results are from San Francisco, New York, or Seattle (major tech cities) and have 10+ years experience, proving the metadata filters worked perfectly.
- **Real-World Relevance**: This type of search mimics actual HR/recruiting scenarios where you need "senior developers with specific skills in key locations" - combining fuzzy skill matching with hard requirements.
- **Distance Score Interpretation**: Higher distances (0.8761 for Olivia) indicate she's still relevant to the query but with a looser semantic match, which is valuable for finding candidates with transferable skills.

Understanding the Results:

- **Distance scores** represent semantic similarity, where lower values indicate higher similarity to the query.
- **Metadata filtering** acts as a hard constraint - only employees meeting these exact criteria are considered.
- **Combined searches** solve real business problems by balancing flexible skill matching with non-negotiable requirements.
- The system successfully **demonstrates how vector databases can power sophisticated talent search platforms** used by modern companies.

Congratulations! You have created your first application to find similar data!

Practice Exercises

In this practice exercise, you will implement advanced search capabilities for a book recommendation system. This exercise will demonstrate similarity search, metadata filtering, and combined search techniques using a book dataset.

1. First examine the following Python list that contains comprehensive book information designed for advanced search scenarios.

```
# List of book dictionaries with comprehensive details for advanced search
books = [
    {
        "id": "book_1",
        "title": "The Great Gatsby",
        "author": "F. Scott Fitzgerald",
        "genre": "Classic",
        "year": 1925,
        "rating": 4.1,
        "pages": 180,
        "description": "A tragic tale of wealth, love, and the American Dream in the Jazz Age",
        "themes": "wealth, corruption, American Dream, social class",
        "setting": "New York, 1920s"
    },
    {
        "id": "book_2",
        "title": "To Kill a Mockingbird",
        "author": "Harper Lee",
        "genre": "Classic",
        "year": 1960,
        "rating": 4.3,
        "pages": 376,
        "description": "A powerful story of racial injustice and moral growth in the American South",
        "themes": "racism, justice, moral courage, childhood innocence",
        "setting": "Alabama, 1930s"
    },
    {
        "id": "book_3",
        "title": "1984",
        "author": "George Orwell",
        "genre": "Dystopian",
        "year": 1949,
        "rating": 4.4,
        "pages": 328,
        "description": "A chilling vision of totalitarian control and surveillance society",
        "themes": "totalitarianism, surveillance, freedom, truth",
        "setting": "Oceania, dystopian future"
    },
    {
        "id": "book_4",
        "title": "Harry Potter and the Philosopher's Stone",
        "author": "J.K. Rowling",
        "genre": "Fantasy",
        "year": 1997,
        "rating": 4.5,
        "pages": 223,
        "description": "A young wizard discovers his magical heritage and begins his education at Hogwarts",
        "themes": "friendship, courage, good vs evil, coming of age",
```

```
        "setting": "England, magical world"
    },
    {
        "id": "book_5",
        "title": "The Lord of the Rings",
        "author": "J.R.R. Tolkien",
        "genre": "Fantasy",
        "year": 1954,
        "rating": 4.5,
        "pages": 1216,
        "description": "An epic fantasy quest to destroy a powerful ring and save Middle-earth",
        "themes": "heroism, friendship, good vs evil, power corruption",
        "setting": "Middle-earth, fantasy realm"
    },
    {
        "id": "book_6",
        "title": "The Hitchhiker's Guide to the Galaxy",
        "author": "Douglas Adams",
        "genre": "Science Fiction",
        "year": 1979,
        "rating": 4.2,
        "pages": 224,
        "description": "A humorous space adventure following Arthur Dent across the galaxy",
        "themes": "absurdity, technology, existence, humor",
        "setting": "Space, various planets"
    },
    {
        "id": "book_7",
        "title": "Dune",
        "author": "Frank Herbert",
        "genre": "Science Fiction",
        "year": 1965,
        "rating": 4.3,
        "pages": 688,
        "description": "A complex tale of politics, religion, and ecology on a desert planet",
        "themes": "power, ecology, religion, politics",
        "setting": "Arrakis, distant future"
    },
    {
        "id": "book_8",
        "title": "The Hunger Games",
        "author": "Suzanne Collins",
        "genre": "Dystopian",
        "year": 2008,
        "rating": 4.2,
        "pages": 374,
        "description": "A teenage girl fights for survival in a brutal televised competition",
        "themes": "survival, oppression, sacrifice, rebellion",
        "setting": "Panem, dystopian future"
    },
]
```


2. Create a new Python file named `books_advanced_search.py` and implement the following structure with comprehensive search capabilities.

3. **Exercise 1: Implement Similarity Search for Book Recommendations**

   Create meaningful text documents for each book that combine title, description, themes, and setting information for semantic search.

   Hint: Combine multiple book attributes into descriptive text documents.

▶ Click here for the solution

4. **Exercise 2: Implement Metadata Filtering**

   Add the book data to your collection with comprehensive metadata for filtering capabilities.

   Hint: Include all book attributes as metadata for flexible filtering.

▶ Click here for the solution

5. **Exercise 3: Create an Advanced Search Function**

   Implement a function that demonstrates multiple search types:

   - Similarity search for "magical fantasy adventure"
   - Filter books by genre (Fantasy or Science Fiction)
   - Filter books by rating (4.0 or higher)
   - Combined search: Find highly-rated dystopian books with similarity search

   Hint: Use the same pattern as the employee search but adapt for book attributes.

▶ Click here for the solution

6. **Exercise 4: Run Your Advanced Book Search**

Execute your script and observe the different types of search results.

Hint: Use the same command pattern as the main lab.

▶ Click here for the solution

7. **Bonus Challenge**: Extend the search to find books by specific criteria such as:
    - Books published in a specific decade
    - Books with similar page counts
    - Books that match multiple themes

Note: Ensure your environment has the required Python packages installed.

Click on the button below to see the complete `books_advanced_search.py` script solution.

▶ Click for the complete script

Conclusion

After completing this lab, you now know that:

- **You can integrate key-value pair data as vector data. You can use vector search with Chroma DB for analysis:** 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 defines an embedding function using SentenceTransformers.
- **You can easily generate embeddings for textual data:** The code creates a collection within the Chroma DB client. The code defines comprehensive employee data and creates meaningful text documents that combine multiple attributes. The SentenceTransformer embedding function generates embeddings for the text documents automatically when they are added to the collection.
- **You can implement advanced search capabilities:** You learned how to implement multiple types of searches including semantic similarity search using natural language queries, metadata filtering for precise criteria-based searches, and combined searches that leverage both similarity and filtering for powerful query capabilities.
- **You use a function named** `perform_advanced_search` to run comprehensive searches including similarity search, metadata filtering, and combined search techniques.

Author(s)

Wojciech "Victor" Fulmyk

Contributor(s)

Richa Arora