动手实验：使用 Chroma 向量数据库进行文本相似性搜索

预计时间： 30 分钟

您将学习的内容

在本实验中，您将使用 Chroma DB 实现文本相似性搜索，该数据库系统旨在高效管理和查询文本数据。

您将学习如何创建一个集合并用一组示例文本文档填充数据库，每个文档都与一个唯一的 ID 相关联，并且其对应的嵌入是使用 SentenceTransformer 嵌入函数生成的。随后，您将对指定的查询词执行相似性搜索，检索与查询词最相似的前三个文档。

学习目标

完成本实验后，您将能够：

- 将 Chroma DB 集成到应用程序中，以存储和查询文本数据。
- 使用 Chroma DB 的 SentenceTransformer 嵌入函数为文本文档生成文本嵌入。
- 实现相似性搜索功能，计算查询嵌入与文档嵌入之间的相似性分数。
- 显示与给定查询词相似的高排名文档。

关于 Skills Network Cloud IDE

Skills Network Cloud IDE 提供了一个用于课程和项目相关实验的动手实验环境。它是一个开源的集成开发环境（IDE）。

关于此实验环境的重要通知

请注意，此实验环境的会话不会被保存。每次您连接到此实验时，都会为您创建一个新的环境。您在之前会话中保存的任何数据将会丢失。请计划在单个会话中完成这些实验，以避免丢失数据。

先决条件

- 对向量数据库的基本知识
- 对 Chroma 向量数据库的理解
- 在完成本实验的任务之前，请使用 设置 Chroma DB 环境 实验。记得保持终端窗口打开。

任务 1：设置 Chroma DB 环境

1. 在主菜单中，选择 **终端** 选项卡，然后选择 **新建终端**。

2. 现在打开一个新终端，并在环境中安装 `chromadb` 以便使用 Chroma DB 向量数据库。在终端窗口中执行以下命令并按 **Enter**。

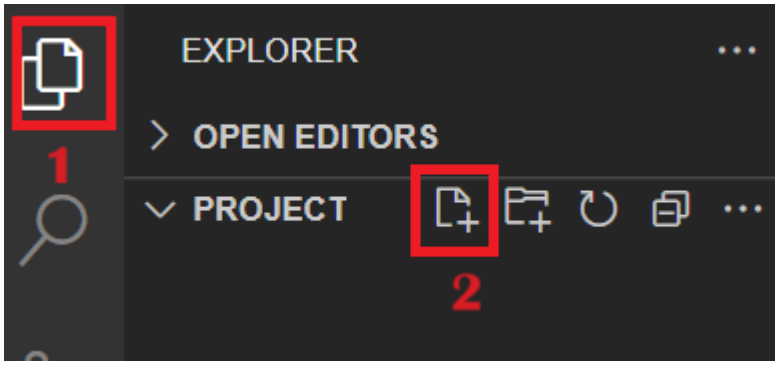
```
pip install chromadb==1.0.12
```

3. 然后在同一个终端窗口中安装另一个依赖项，通过执行以下命令，您之前安装的依赖项也在此窗口中。

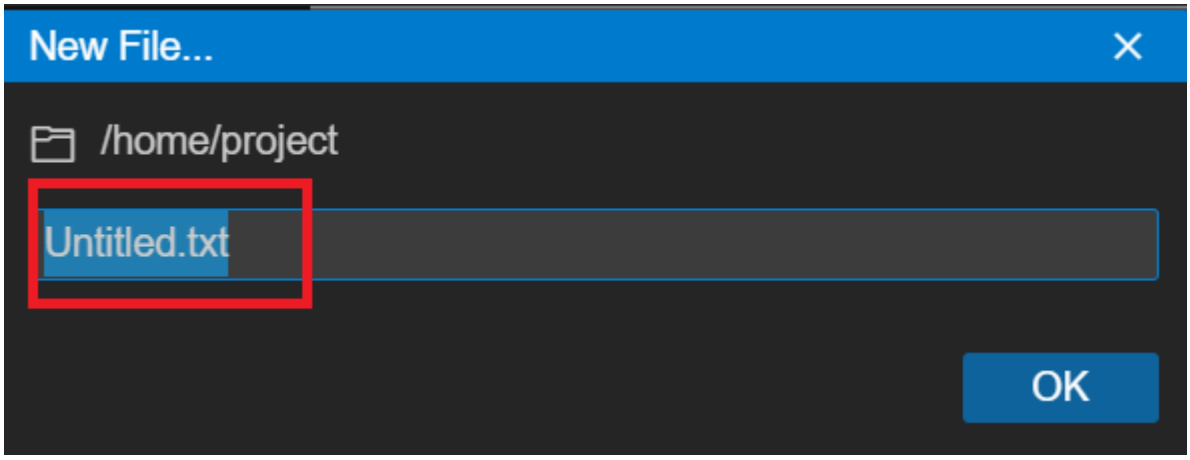
```
pip install sentence-transformers==4.1.0
```

注意：请不要关闭正在运行 Docker 的终端。

4. 在终端窗口的左侧选择 **资源管理器**，如以下截图中编号 1 所示。然后，在 **项目** 文件夹中，选择 **新建文件**，如以下截图中编号 2 所示。



5. 执行完上述步骤后，会弹出一个默认文件名为 **Untitled.txt** 的框，如以下截图所示。将其默认名称替换为 `similarity_search.py`。



任务 2：创建集合并嵌入数据

现在，您将学习如何创建集合并嵌入数据。

1. 首先，从 `chromadb` 包中导入 Chroma DB 库和嵌入函数。通过导入这些类，您可以创建一个与 Chroma DB 数据库交互的客户端实例，并定义嵌入模型。在您的 Python 文件中包含以下命令。

```
# 从 chromadb 包中导入必要的模块：
# chromadb 用于与 Chroma DB 数据库交互，
# embedding_functions 用于定义嵌入模型
import chromadb
from chromadb.utils import embedding_functions
# 使用 SentenceTransformers 定义嵌入函数
ef = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="all-MiniLM-L6-v2"
)
```

   这段代码从 Chroma DB 导入必要的模块，包括用于与数据库交互的主 `chromadb` 库和用于使用 SentenceTransformers 定义嵌入模型的 `embedding_functions`。

2. 然后创建一个 Chroma DB 客户端并初始化 `collection_name` 变量，指定集合的名称。

```
# 创建一个新的 ChromaClient 实例以与 Chroma DB 交互
client = chromadb.Client()
# 定义要创建或检索的集合名称
collection_name = "my_grocery_collection"
```

3. 创建主函数，其中将包含与 Chroma DB 数据库交互的代码。

```
# 定义与 Chroma DB 交互的主函数
def main():
    try:
        # 将您的数据库操作放在此块中
        pass
    except Exception as error:  # 捕获任何错误并记录到控制台
        print(f"错误: {error}")
```

4. 然后在数据库中创建一个集合，并在 `main` 函数的 `try` 块中执行以下命令。

```
# 在 Chroma 数据库中创建一个指定名称、距离度量和嵌入函数的集合。
# 在这种情况下，我们使用余弦距离
collection = client.create_collection(
    name=collection_name,
    metadata={"description": "用于存储杂货数据的集合"},
    configuration={
        "hnsw": {"space": "cosine"},
        "embedding_function": ef
    }
)
print(f"集合已创建: {collection.name}")
```

- 这段代码使用客户端的 `create_collection` 方法从数据库创建名为“my_grocery_collection”的集合。
- 此方法在数据库中创建一个具有指定名称和配置的新集合。
- `collection` 变量保存对该集合的引用，以便进行进一步操作，并配置为使用 SentenceTransformer 嵌入函数为文本数据生成嵌入。

5. 接下来，在 `try` 块中创建集合后，定义示例数据。以下数组包含杂货项目的示例文本字符串。

```
# 与杂货相关的文本项目数组
texts = [
    '新鲜红苹果',
    '有机香蕉',
    '成熟芒果',
    '全麦面包',
    '农场新鲜鸡蛋',
```

```
    '天然酸奶',
    '冷冻蔬菜',
    '草饲牛肉',
    '自由放养鸡',
    '新鲜三文鱼片',
    '芳香咖啡豆',
    '纯蜂蜜',
    '金色苹果',
    '红色水果'
]
```

6. 现在，通过在文本数组之后包含以下命令，为每个文本字符串生成唯一 ID：

```
# 为 'texts' 数组中的每个文本项目创建唯一 ID 列表
 # 每个 ID 的格式为 'food_<index>'，其中 <index> 从 1 开始
ids = [f"food_{index + 1}" for index, _ in enumerate(texts)]
```

   - 这段代码根据 `texts` 数组中的索引为文档生成唯一 ID。每个 ID 的格式为 `"food_<index>"`，其中 `<index>` 从 1 开始。

任务 3：生成嵌入并将文本添加到集合

1. 接下来，使用以下代码将 ID 和文本添加到集合中：

```
# 将文档及其对应的 ID 添加到集合中
# `add` 方法将数据插入到集合中
# 文档是实际的文本项，ID 是唯一标识符
# ChromaDB 将使用配置的嵌入函数自动生成嵌入
collection.add(
    documents=texts,
    metadatas=[{"source": "grocery_store", "category": "food"} for _ in texts],
    ids=ids
)
```

- 通过使用集合的 `add` 方法传递文档、元数据和 ID，代码将文档及其对应的信息添加到 Chroma DB 数据库集合中。
- 这段代码支持高效存储和检索文本数据，以及其自动生成的嵌入，以便于进行自然语言处理任务，如相似性搜索和文档聚类。

2. 您可以使用以下代码，该代码使用 `get` 方法访问所有项目。

```
# 检索存储在集合中的所有项目（文档）
# `get` 方法从集合中获取所有数据
all_items = collection.get()
# 将检索到的项目记录到控制台以供检查
# 这将打印出存储在集合中的所有文档、ID 和元数据
print("集合内容：")
print(f"文档数量：{len(all_items['documents'])}")
```

- `all_items` 这个变量存储 `collection.get()` 方法的结果。该变量保存从集合中检索到的项目，包括文档、ID 和元数据。
- `collection.get()` 这个 `get` 方法检索存储在集合中的所有项目。这些项目包括文档及其对应的 ID。
- 该方法将所有存储在集合中的数据作为字典返回，包含文档、ID 和元数据。

任务 4：执行相似性搜索

1. 使用以下代码在主函数之后创建一个名为 `perform_similarity_search` 的函数。您将编写查询以查找相似项目。

```
# 在集合中执行相似性搜索的函数
def perform_similarity_search(collection, all_items):
    try:
        # 在此块中放置您的相似性搜索代码
        pass
    except Exception as error:
        print(f"相似性搜索中的错误：{error}")
```

2. 在 `try` 块内，初始化一个名为 `query_term` 的变量。以下是一个示例：

```
# 定义您想在集合中搜索的查询词
query_term = "apple"
```

3. 然后，创建调用 `query` 方法的代码。查询方法用于搜索与查询词相似的文档，同时包括 *n_results* 参数。*n_results* 参数指定要检索的顶部结果数量。

```
# 执行查询以搜索与 'query_term' 最相似的文档
results = collection.query(
    query_texts=[query_term],
    n_results=3  # 检索前 3 个结果
)
print(f"查询 '{query_term}' 的结果：")
print(results)
```

任务 5：处理结果并显示高度相似的文本

1. 接下来，创建代码以处理未找到与查询词相似的文档的结果，并在控制台记录一条消息。

```
# 检查是否没有返回结果或结果数组为空
if not results or not results['ids'] or len(results['ids'][0]) == 0:
    # 记录一条消息，指示未找到与查询词相似的文档
    print(f'没有找到与 "{query_term}" 相似的文档')
    return
```

2. 你还需要显示与查询相比，距离最近的前几个文档。

```
print(f'与 "{query_term}" 相似的前 3 个文档：')
# 访问 'results["ids"]' 和 'results["distances"]' 中的嵌套数组
for i in range(min(3, len(results['ids'][0]))):
    doc_id = results['ids'][0][i]  # 从 'ids' 数组中获取 ID
```

```
            score = results['distances'][0][i]  # 从 'distances' 数组中获取分数
            # 从结果中检索文本数据
            text = results['documents'][0][i]
            if not text:
                print(f' - ID: {doc_id}, 文本: "文本不可用", 分数: {score:.4f}')
            else:
                print(f' - ID: {doc_id}, 文本: "{text}", 分数: {score:.4f}')
```

3. 接下来，在 `main` 函数的 `try` 块末尾调用 `perform_similarity_search` 函数，使用以下代码：

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

4. 然后调用 `main` 函数以创建集合并生成嵌入。

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

点击下面的按钮查看完整的 `similarity_search.py` 脚本。你可以将解决方案复制粘贴到你的 `similarity_search.py` 文件中。

▶ 点击查看脚本

3. 修改检索顶级文档的代码块，以允许多个搜索查询。

   提示：在整个代码块周围使用一个外循环。

▶ 点击这里查看解决方案

4. 通过在终端中使用 `python` 运行当前文件名 `similarity_search.py` 来验证输出是否正确。

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

现在您已经完成了每个练习，点击下面的链接查看 `similarity_search.py` 的完整代码流程。

点击下面的按钮查看完整修改后的 `similarity_search.py` 脚本。

▶ 点击查看脚本