```
import logging

import os

import uuid

from typing import List, Optional


import chromadb

import numpy as np

from dotenv import load_dotenv


from swarms.utils.data_to_text import data_to_text

from swarms.utils.markdown_message import display_markdown_message


# Load environment variables

load_dotenv()



# Results storage using local ChromaDB

class ChromaDB:

    """


    ChromaDB database


    Args:

        metric (str): The similarity metric to use.

        output (str): The name of the collection to store the results in.

          limit_tokens (int, optional): The maximum number of tokens to use for the query. Defaults to
```

```
1000.
        n_results (int, optional): The number of results to retrieve. Defaults to 2.


    Methods:
        add: _description_
        query: _description_


    Examples:
        >>> chromadb = ChromaDB(
        >>>     metric="cosine",
        >>>     output="results",
        >>>     llm="gpt3",
        >>>     openai_api_key=OPENAI_API_KEY,
        >>> )
        >>> chromadb.add(task, result, result_id)
    """


    def __init__(
        self,
        metric: str = "cosine",
        output_dir: str = "swarms",
        limit_tokens: Optional[int] = 1000,
        n_results: int = 2,
        docs_folder: Optional[str] = None,
        verbose: bool = False,
        *args,
    **kwargs,
):
    self.metric = metric
    self.output_dir = output_dir
    self.limit_tokens = limit_tokens
    self.n_results = n_results
    self.docs_folder = docs_folder
    self.verbose = verbose

    # Disable ChromaDB logging
    if verbose:
        logging.getLogger("chromadb").setLevel(logging.INFO)

    # Create Chroma collection
    chroma_persist_dir = "chroma"
    chroma_client = chromadb.PersistentClient(
        settings=chromadb.config.Settings(
            persist_directory=chroma_persist_dir,
        ),
        *args,
        **kwargs,
    )
    # Create ChromaDB client
    self.client = chromadb.Client()

    # Create Chroma collection
    self.collection = chroma_client.get_or_create_collection(
        name=output_dir,
        metadata={"hnsw:space": metric},
        *args,
        **kwargs,
    )
    display_markdown_message(
        "ChromaDB collection created:"
        f" {self.collection.name} with metric: {self.metric} and"
        f" output directory: {self.output_dir}"
    )

    # If docs
    if docs_folder:
        display_markdown_message(
            f"Traversing directory: {docs_folder}"
        )
        self.traverse_directory()

def add(
    self,
    document: str,
    images: List[np.ndarray] = None,
    img_urls: List[str] = None,
    *args,
    **kwargs,
    ):
        """
        Add a document to the ChromaDB collection.

        Args:
            document (str): The document to be added.
              condition (bool, optional): The condition to check before adding the document. Defaults to
True.

        Returns:
            str: The ID of the added document.
        """
        try:
            doc_id = str(uuid.uuid4())
            self.collection.add(
                ids=[doc_id],
                documents=[document],
                images=images,
                uris=img_urls,
                *args,
                **kwargs,
            )
            return doc_id
        except Exception as e:
            raise Exception(f"Failed to add document: {str(e)}")
```

```
def query(
    self,
    query_text: str,
    query_images: List[np.ndarray],
    *args,
    **kwargs,
):
    """
    Query documents from the ChromaDB collection.


    Args:
        query (str): The query string.
        n_docs (int, optional): The number of documents to retrieve. Defaults to 1.


    Returns:
        dict: The retrieved documents.
    """
    try:
        docs = self.collection.query(
            query_texts=[query_text],
            query_images=query_images,
            n_results=self.n_docs,
            *args,
            **kwargs,
        )["documents"]
        return docs[0]
    except Exception as e:
        raise Exception(f"Failed to query documents: {str(e)}")


def traverse_directory(self):
    """
    Traverse through every file in the given directory and its subdirectories,
    and return the paths of all files.
    Parameters:
    - directory_name (str): The name of the directory to traverse.
    Returns:
    - list: A list of paths to each file in the directory and its subdirectories.
    """
    image_extensions = [
        ".jpg",
        ".jpeg",
        ".png",
    ]
    images = []
    for root, dirs, files in os.walk(self.docs_folder):
        for file in files:
            _, ext = os.path.splitext(file)
            if ext.lower() in image_extensions:
                images.append(os.path.join(root, file))
            else:
                data = data_to_text(file)
                added_to_db = self.add([data])
            print(f"{file} added to Database")
    if images:
        added_to_db = self.add(img_urls=[images])
        print(f"{len(images)} images added to Database ")
    return added_to_db
```