Student: Elshan Naghizade

Project: Python Library To Transform Image Datasets in Query-able SQL Tables

Date: 30 July, 2023

OLD MULTI-MODULE CODEBASE	CURRENT MONOLITHIC CODEBASE
Manual Feature Extraction: https://github.com/ADA-GWU/guidedresearchproject-Elshan-Naghizade/blob/main/Refactoring_lib_tidy.py	**Single Class Main Module:** https://github.com/ADA-GWU/guidedresearchproject-Elshan-Naghizade/blob/main/main_combine_alpha.py
PostreSQL BLOB Usage: https://github.com/ADA-GWU/guidedresearchproject-Elshan-Naghizade/blob/main/PostreSQL_blob_usage.py	
SQL3lite BLOB Usage: https://github.com/ADA-GWU/guidedresearchproject-Elshan-Naghizade/blob/main/blob_usage.py	

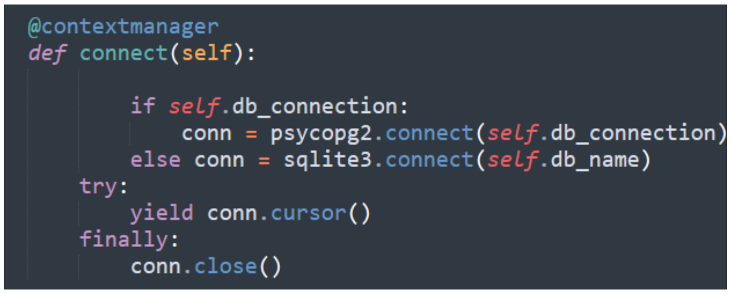
INTRODUCTION

The current codebase and the old multimodule codebase for ImageFeatureDataset have been meticulously analyzed, and a number of much needed improvements have been reached. These modifications have a direct impact on the codebase's overall functionality, usability, efficiency, and robustness.

One of the most significant distinctions that should be highlighted is the modular organization of the codebase. In the old multimodule codebase, the functionalities were scattered across three distinct modules. This separation introduced additional complexity, particularly regarding intermodule communication, function calls, and overall code management. In stark contrast, the current codebase has effectively consolidated all the functionalities into a single, coherent unit. This amalgamation of functions into one module in the current codebase significantly simplifies the code's organization. It also eases the tracking of function calls and data flow, leading to an overall reduction in management overhead and potential error points, thereby increasing the codebase's maintainability and reliability.

CONTEXT MANAGER

One of the advancements in the current codebase that is particularly noteworthy is the introduction of a context manager to handle database connections. In the old codebase, each database operation required manual opening and closing of the connection, a process which was not only tedious but also error-prone. However, the current codebase has automated this process using a context manager. The context manager's ability to automatically handle connection management ensures that connections are correctly closed after their usage, reducing the risk of connection leaks. Additionally, the automatic management of connections enhances the code readability and allows for a cleaner and more concise coding style.

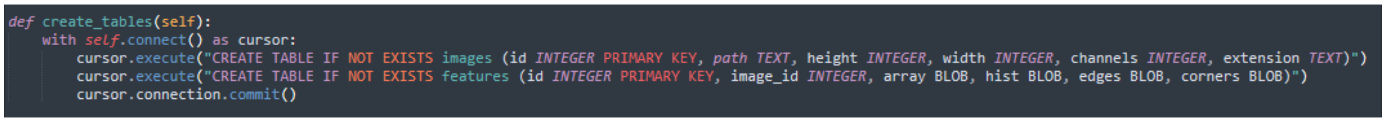


```
@contextmanager
def connect(self):

        if self.db_connection:
            conn = psycopg2.connect(self.db_connection)
        else conn = sqlite3.connect(self.db_name)
    try:
        yield conn.cursor()
    finally:
        conn.close()
```

Figure 1. Database Context Manager.

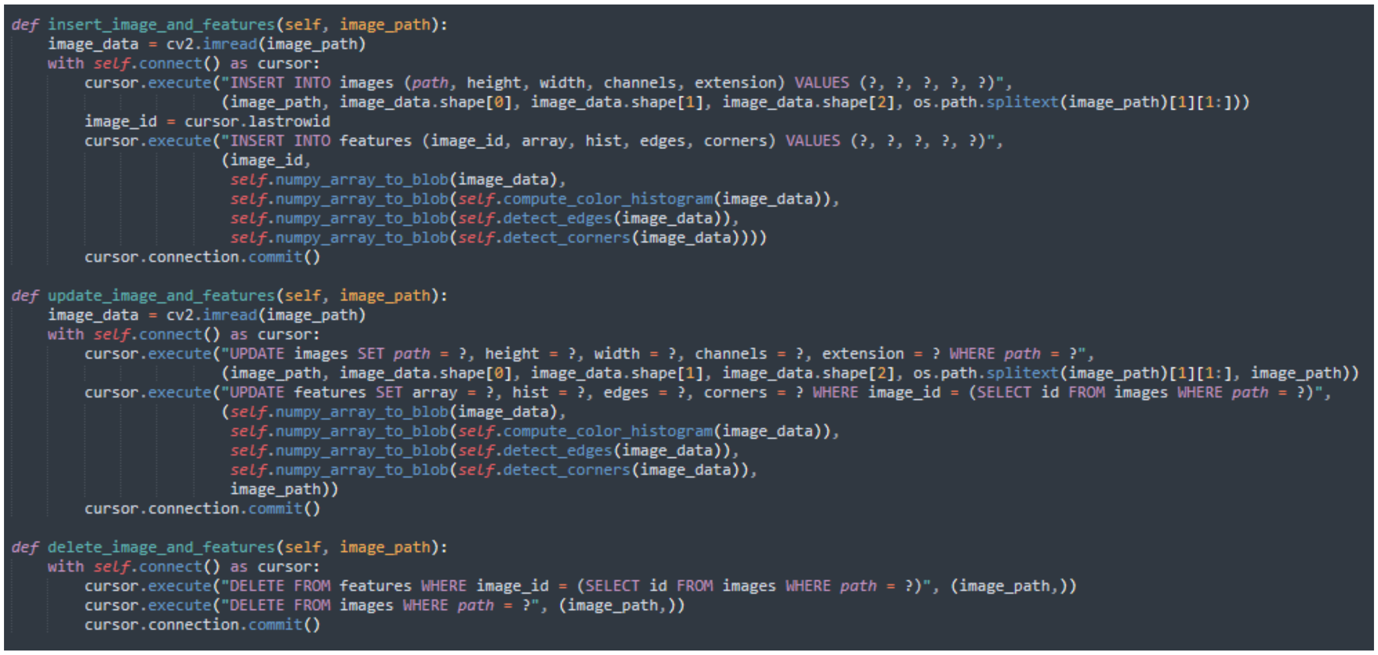
DATABASE & IMAGE FEATURE SUPPORT

Another critical differentiation between the old multimodule codebase and the current codebase is the support for multiple database systems. The current codebase has the capability to connect to and operate with both SQLite and PostgreSQL databases. In the old codebase, only SQLite was supported. The introduction of PostgreSQL support in the current codebase offers a broader array of possibilities. With its ability to handle more extensive datasets, increased robustness, and excellent support for high concurrency, PostgreSQL offers numerous advantages in scenarios where SQLite might not be the most efficient choice. The addition of PostgreSQL support therefore ensures that the current codebase is more adaptable and better suited to a variety of use cases. The current codebase also introduces improvements in the way image records are handled. Operations such as the insertion, updating, and deletion of image records and their associated features are managed in a comprehensive manner in the current codebase. This integrated handling of image and feature data maintains data consistency and integrity. It enhances the reliability of the codebase and ensures that the data it handles is always accurate and up to date.



```python
def create_tables(self):
    with self.connect() as cursor:
        cursor.execute("CREATE TABLE IF NOT EXISTS images (id INTEGER PRIMARY KEY, path TEXT, height INTEGER, width INTEGER, channels INTEGER, extension TEXT)")
        cursor.execute("CREATE TABLE IF NOT EXISTS features (id INTEGER PRIMARY KEY, image_id INTEGER, array BLOB, hist BLOB, edges BLOB, corners BLOB)")
        cursor.connection.commit()
```

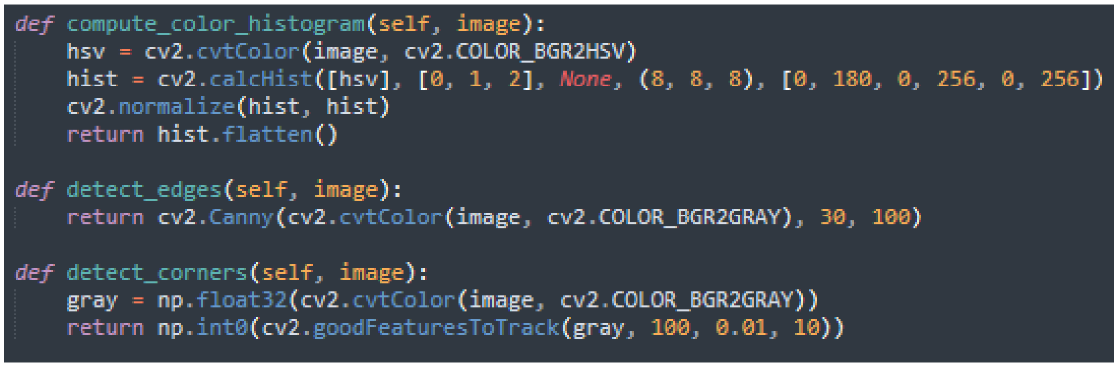
Figure 2. Inline Database Creation.



```python
def insert_image_and_features(self, image_path):
    image_data = cv2.imread(image_path)
    with self.connect() as cursor:
        cursor.execute("INSERT INTO images (path, height, width, channels, extension) VALUES (?, ?, ?, ?, ?)",
                       (image_path, image_data.shape[0], image_data.shape[1], image_data.shape[2], os.path.splitext(image_path)[1][1:]))
        image_id = cursor.lastrowid
        cursor.execute("INSERT INTO features (image_id, array, hist, edges, corners) VALUES (?, ?, ?, ?, ?)",
                       (image_id,
                        self.numpy_array_to_blob(image_data),
                        self.numpy_array_to_blob(self.compute_color_histogram(image_data)),
                        self.numpy_array_to_blob(self.detect_edges(image_data)),
                        self.numpy_array_to_blob(self.detect_corners(image_data))))
        cursor.connection.commit()

def update_image_and_features(self, image_path):
    image_data = cv2.imread(image_path)
    with self.connect() as cursor:
        cursor.execute("UPDATE images SET path = ?, height = ?, width = ?, channels = ?, extension = ? WHERE path = ?",
                       (image_path, image_data.shape[0], image_data.shape[1], image_data.shape[2], os.path.splitext(image_path)[1][1:], image_path))
        cursor.execute("UPDATE features SET array = ?, hist = ?, edges = ?, corners = ? WHERE image_id = (SELECT id FROM images WHERE path = ?)",
                       (self.numpy_array_to_blob(image_data),
                        self.numpy_array_to_blob(self.compute_color_histogram(image_data)),
                        self.numpy_array_to_blob(self.detect_edges(image_data)),
                        self.numpy_array_to_blob(self.detect_corners(image_data)),
                        image_path))
        cursor.connection.commit()

def delete_image_and_features(self, image_path):
    with self.connect() as cursor:
        cursor.execute("DELETE FROM features WHERE image_id = (SELECT id FROM images WHERE path = ?)", (image_path,))
        cursor.execute("DELETE FROM images WHERE path = ?", (image_path,))
        cursor.connection.commit()
```

Figure 3. Inline Data Msanagement.

In the realm of image processing functions, the current codebase surpasses the old multimodule codebase in many aspects. The old codebase was devoid of any sophisticated image analysis features. In contrast, the current codebase incorporates functions to compute the color histogram of images, detect edges, and identify corners. These functions enhance the capabilities of the ImageFeatureDataset class and pave the way for a myriad of potential applications ranging from machine learning to advanced computer vision tasks. Thus, the current codebase is far more versatile and potent in handling image processing tasks than the old multimodule codebase.

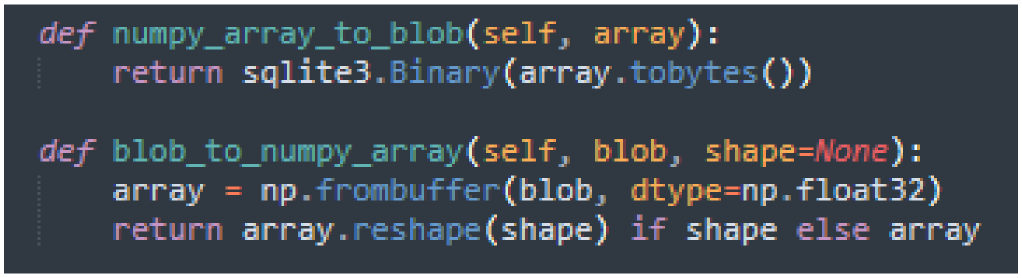


```python
def compute_color_histogram(self, image):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    hist = cv2.calcHist([hsv], [0, 1, 2], None, (8, 8, 8), [0, 180, 0, 256, 0, 256])
    cv2.normalize(hist, hist)
    return hist.flatten()

def detect_edges(self, image):
    return cv2.Canny(cv2.cvtColor(image, cv2.COLOR_BGR2GRAY), 30, 100)

def detect_corners(self, image):
    gray = np.float32(cv2.cvtColor(image, cv2.COLOR_BGR2GRAY))
    return np.int0(cv2.goodFeaturesToTrack(gray, 100, 0.01, 10))
```

Figure 4. Feature Extraction.

STRUCTURAL DIFFERENCES

Lastly, the current codebase includes functions to convert numpy arrays into blobs and vice versa. This is an essential feature for the storage and retrieval of image data and features. This capability to seamlessly convert between numpy arrays and blobs enables the efficient storage of large quantities of image data and features in the database. The absence of such functions in the old multimodule codebase limited its capacity to handle image data effectively.



```python
def numpy_array_to_blob(self, array):
    return sqlite3.Binary(array.tobytes())

def blob_to_numpy_array(self, blob, shape=None):
    array = np.frombuffer(blob, dtype=np.float32)
    return array.reshape(shape) if shape else array
```

Figure 5. BLOB Conversion.

In summary, the analysis of the old multimodule codebase and the current codebase reveals that the latter presents numerous advancements over the former. These advancements include structural changes, enhanced database connection management, support for an additional database system, improvements in image processing and data handling functions, and the introduction of conversion functions between numpy arrays and blobs. These developments render the current codebase more versatile, efficient, robust, and capable. As a result, the choice of the current codebase as the final codebase is highly justified. The combination of its increased capabilities and advancements in image data management and processing indicate that it is the optimal choice for a more effective ImageFeatureDataset.

IMPLEMENTATION BREAKDOWN

At the heart of the codebase is the ImageFeatureDataset class, initialized with two optional parameters: db_name and db_connection. The initialization process is designed to initiate a SQLite or PostgreSQL database connection using the provided parameters. The decision to use either database management system depends on the presence of a database connection string, a process that enhances the flexibility of the codebase to cater to different usage scenarios. It is worth noting that if no connection string is provided, the program defaults to using SQLite with a database name as provided or the default 'image_features.db'.

The ImageFeatureDataset class is noted to contain a context manager called connect(). This context manager is fundamental to the class as it manages the database connection process, ensuring connections are correctly initiated and properly closed to avoid connection leaks. The incorporation of a context manager into the connection handling process marks an evolution towards automated resource management and significantly reduces potential errors associated with manual resource handling.

The 'create_tables' method is observed to be responsible for initializing the necessary tables in the database, including 'images' and 'features'. Each table serves a unique purpose with the 'images' table storing basic image properties and the 'features' table storing advanced image features. The 'images' table is found to store the path, height, width, channels, and the extension of each image, while the 'features' table is responsible for storing image features, including the color histogram, detected edges, and corners. The proper segregation of basic image data and advanced features in different tables has been employed, which greatly aids in the organization of data and potentially speeds up database queries.

A range of image processing methods are included in the ImageFeatureDataset class. These methods, such as 'compute_color_histogram', 'detect_edges', and 'detect_corners', perform specific operations on an image, enabling the class to handle various image processing tasks. These functions are crucial in extracting valuable information from image data, thus increasing the scope of applications of the class.

The 'insert_image_and_features', 'update_image_and_features', and 'delete_image_and_features' methods are observed in the class. These methods manage the insertion, updating, and deletion of image data and associated features from the database. The presence of these functions indicates a comprehensive data management strategy in the ImageFeatureDataset class, and it is ensured that the data remains consistent and accurate.

The 'get_all_images' and 'get_image_and_features' methods have been incorporated to facilitate the retrieval of data from the database. The 'get_all_images' method returns all the image paths stored in the database, while the 'get_image_and_features' method retrieves a specific image's data and features using the image's path as an identifier. These methods ensure that the required data can be retrieved effortlessly, contributing to the usability of the class.

Finally, the 'numpy_array_to_blob' and 'blob_to_numpy_array' methods are included in the class. These methods handle the conversion of numpy arrays into blobs for storage in the database and vice versa. They allow for efficient storage and retrieval of image data and features from the database, thus enhancing the overall performance and flexibility of the class.