Python Library To Transform Image Datasets to Query-able SQL Tables

Class: *CSCI 6917/6918*
Supervisors: *Dr. Kaisler & Dr. Hasanov*
Student: *Elshan Naghizade*
Date: *10 July*

Project Statement

Problem: Conventional Pytorch/Tensorflow dataloaders use unindexed/sequential CRUD operations.

Solution: Delegating data manipulation of Image Datasets to Relational Databases (SQL)

Rationalizing the objective:

- Search/Fetch/Update operations get blazingly fast thanks to the internal optimizations SQL databases have. (indexing, hashing, etc.)
- Simpler SQL queries to preprocess datasets as opposed to Python coding.

Users' Perspective

- Computer Vision engineers spend a significant amount of time to cleanse and transform their image datasets into a state suitable for the model to be developed. Having simple SQL queries to snap through the datasets simplifies the automation of data-preprocessing requiring less manual interventions into the dataset itself, while interacting with the databases itself.

RGB BLOB files and their features from image datasets can be automatically extracted and transformed into a structured form with this library. The use of conventional SQL queries on these database tables is most precious utility, dramatically reducing the time previously spent on developing case-specific dataloaders. Above all, the flexibility and speed of SQL have been effectively harnessed.

Library Workflow

Image Dataset (Zip/Rar/Folder) Path Detection

↓

Subforlder Traversal

↓

Image to Numpy transformation

↓

Feature Extraction

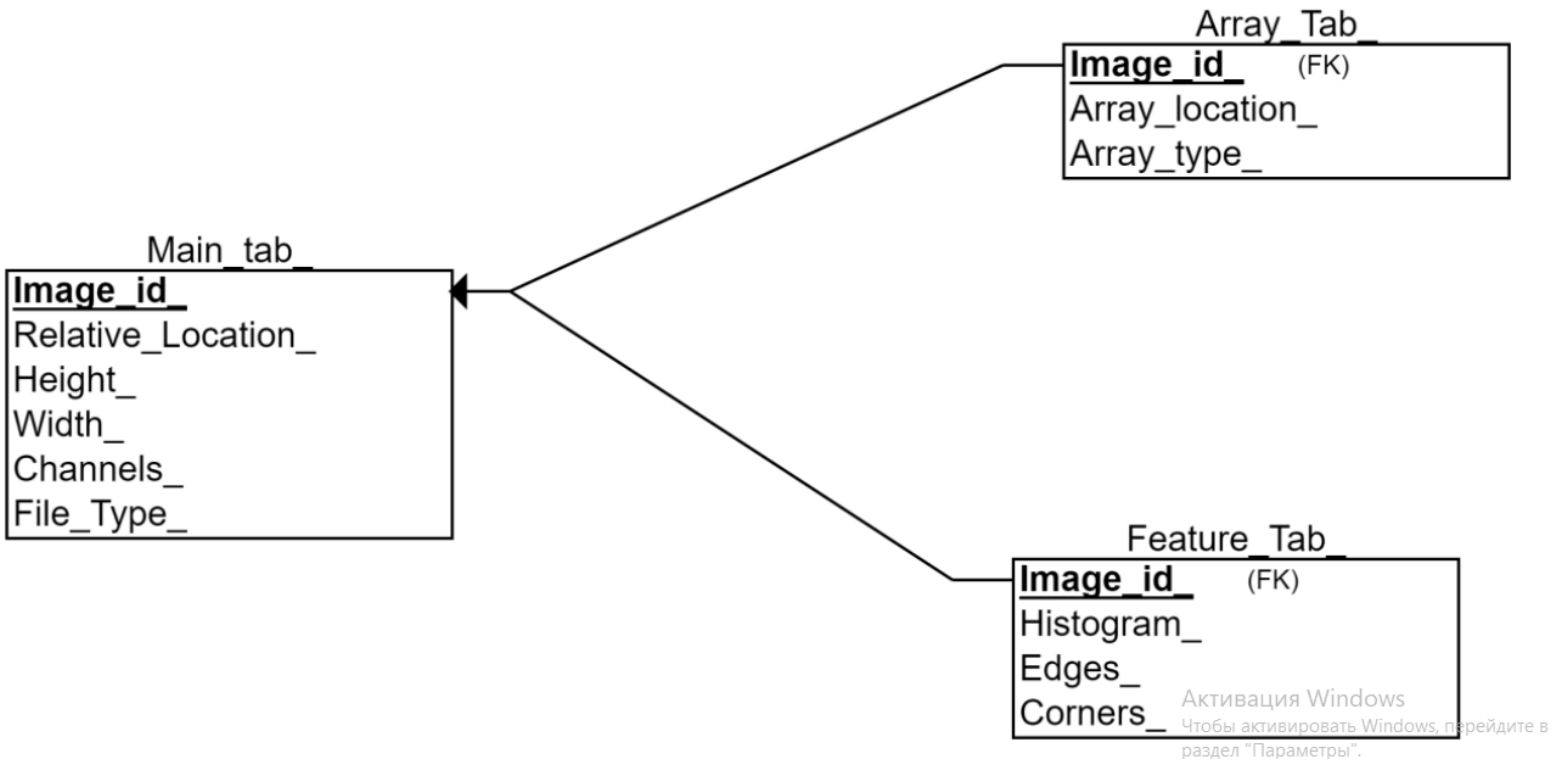
↓

Numpy to BLOB transformation

↓

Database Entry Stage

Database Schema



- 4NF Structure
- Images, Features Stand fro BLOBS
- Location stands for BLOB storage
- Generated keys were deemed more suitable since they are to be used as Foreign Keys

SQL Transformation

Lite version - *Generation of SQL statements as texts and storing them as .sql files.*

Full version – *Using psycopg2 and sqlite3 to directly operate on database systems.*

Advantages of "BLOB"s over Image File Paths:

- ✓ Direct directly running queries on images
- ✓ By having the image stored in the database, synchronicity between the image and its related data is ensured. This implies that when a record is deleted, the associated image is also deleted.
- ✓ There is no need to worry about the management of a separate file storage system, and all related data to an entity is kept in one place.
- ✓ Transferring the database between systems or environments is made easier when all data is contained within the database. If images are stored as separate files, both the database and image files would need to be transferred, and consistency of the paths would need to be maintained.

Lite Database Entry:
Automatically Generated Insert Statements

```
INSERT INTO Main_tab_
(Relative_Location_, Height_, Width_, Channels_, File_Type_, Image_id_)
VALUES
('BLOB_Storage', 720, 1280, 3, 'jpg', 1);

INSERT INTO Array_Tab_
(Array_location_, Array_type_, Image_id_)
VALUES
('BLOB_Storage', 'TypeofArray', 1);

INSERT INTO Feature_Tab_
(Histogram_, Edges_, Corners_, Image_id_)
VALUES
('Histogram/Value', 'Edges/Value', 'Corners/Value', 1);
```

Full Database Entry:
Database Context Manager

```
@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()
```

Technical Overview (1)

Data extraction: The initial step of the process, data extraction, is performed by a loader capable of reading a multitude of image formats. These include JPEG, PNG, TIFF, BMP, and more. This feat was achieved by integrating the capabilities of two existing libraries, PIL (Pillow), and OpenCV. Both individual images and directories comprising subdirectories of images can be navigated by the loader, which systematically traverses the file tree.

```
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()
```

Technical Overview (2)

Feature extraction: After successful data extraction, the next stage involves the extraction of meaningful features. This task is performed by feature extractors equipped to handle basic and complex features. Simple features include color histograms, texture, and shape, while complex ones encompass edges and corners. Moreover, deep learning features derived from pre-trained models are incorporated. OpenCV is the primary tool utilized for simple feature extraction, but when paired with scikit-image, advanced features are effectively extracted.

```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))
```

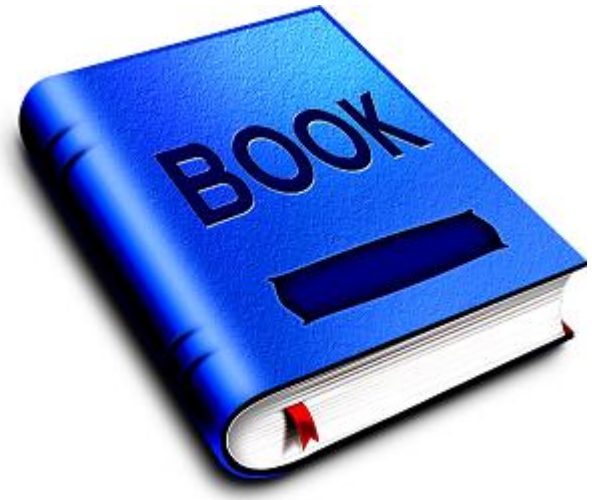
Technical Overview (3)

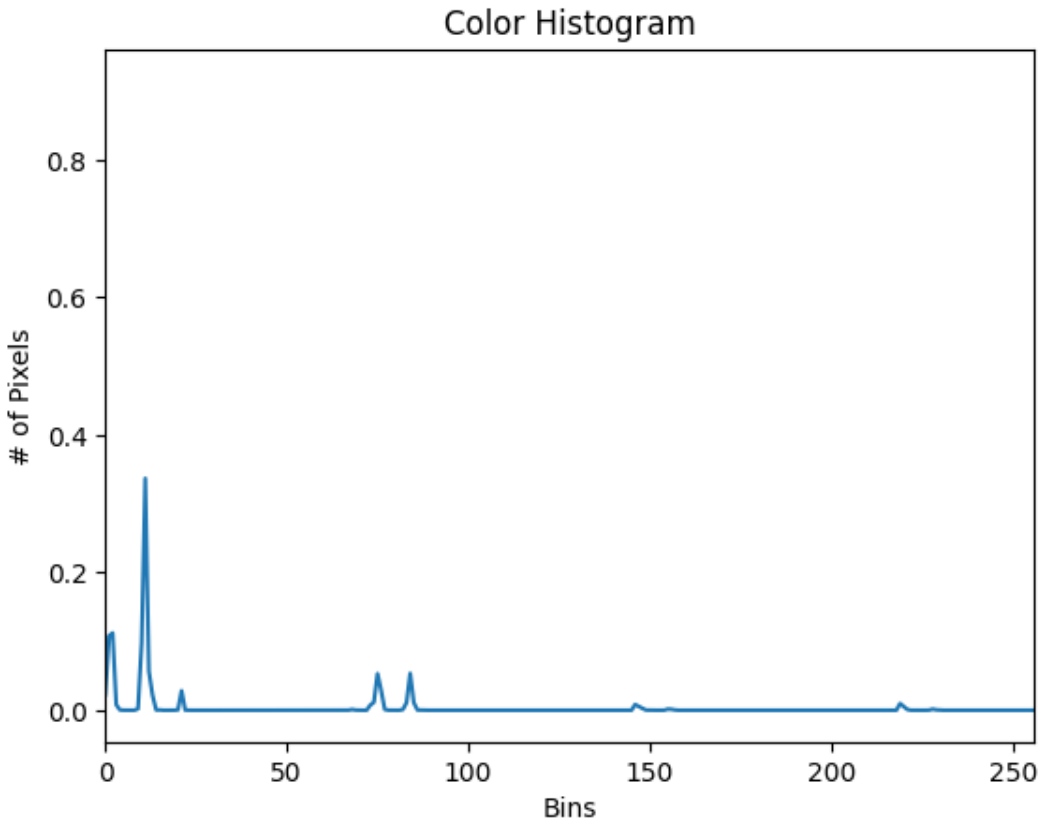
Data transformation: Subsequent to the extraction of features, they are converted into an SQL format, which enables query-ability. Libraries such as SQL3Lite and psycopg2 are utilized to create SQL tables and to insert data into them. This transformation allows the initially unstructured image data to be stored and queried in a structured manner, facilitating a more efficient data handling process.

```
def get_image_and_features(self, image_path):
    with self.connect() as cursor:
        cursor.execute("SELECT images.id, path, height, width, channels, extension, array, hist, edges, corners FROM images "
                       "JOIN features ON images.id = features.image_id WHERE path=?", (image_path,))
        row = cursor.fetchone()
        if row:
            image_id, _, height, width, channels, extension, array, hist, edges, corners = row
            image_data = self.blob_to_numpy_array(array, (height, width, channels))
            hist_data = self.blob_to_numpy_array(hist)
            edges_data = self.blob_to_numpy_array(edges)
            corners_data = self.blob_to_numpy_array(corners)
            return image_data, hist_data, edges_data, corners_data, extension
        return None, None, None, None, None
```

Results (Sample Image Output)

```
File Extension: png
Image Dimension    : (206, 245, 3)
Image Height       : 206
Image Width        : 245
Number of Channels : 3
Total Pixels       : 151410
Image Datatype     : uint8
```

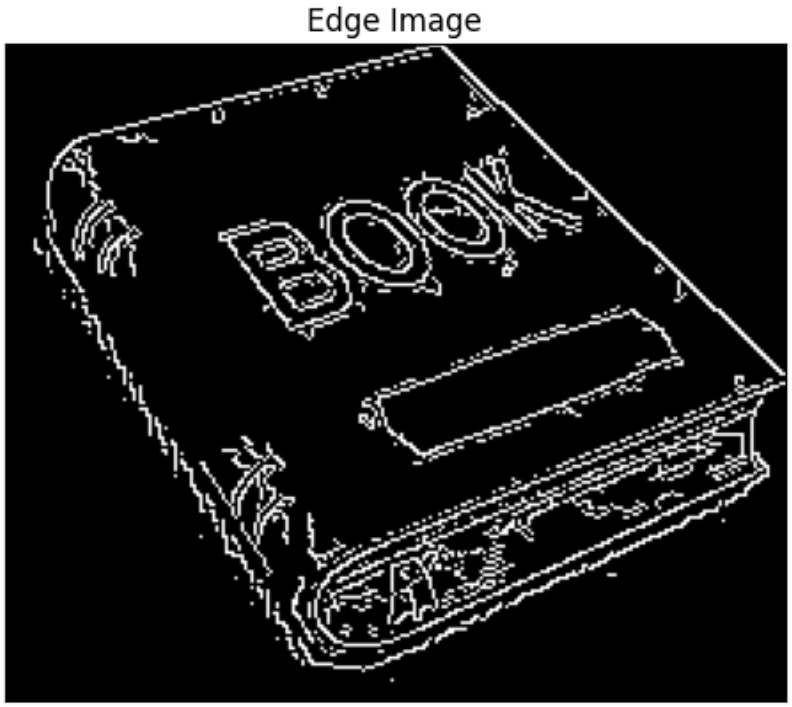




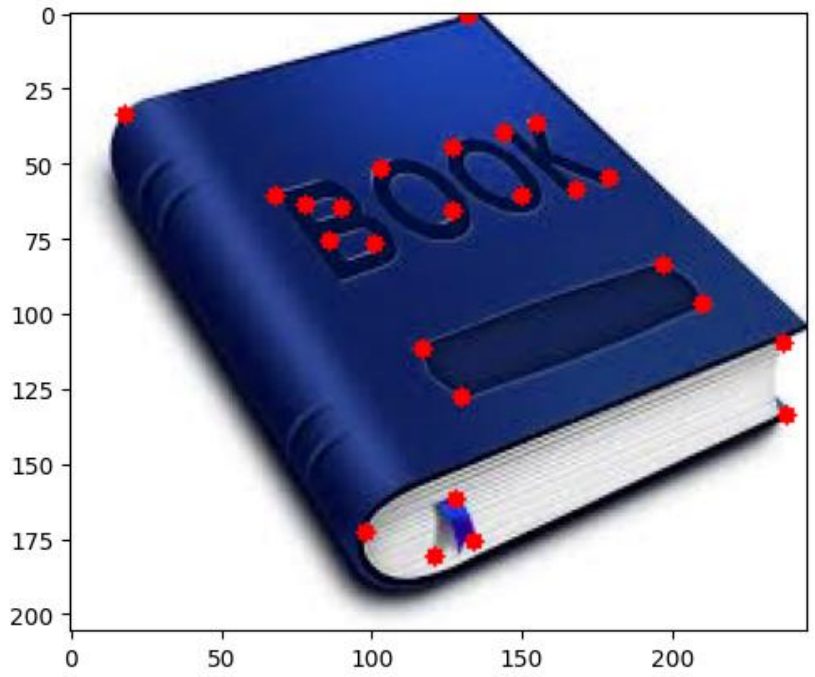
Results (Sample Image Output)

BLOB storage is converted into Numpy arrays to facilitate visualzation with MatPlotLib

Edge Array Visualized:



Corner Vector Visualized:



Future Work

- ✓ *Comprehensive comparative analysis with PyTorch DataLoader Class*
- ✓ *Support for more dataset types (text, video, etc.)*
- ✓ *Scope-Specific Image Format Support (medical imaging, satellite capture, etc.)*
- ✓ *Metadata Extraction*
- ✓ *NoSQL Implementation – Primarily, MongoDB*