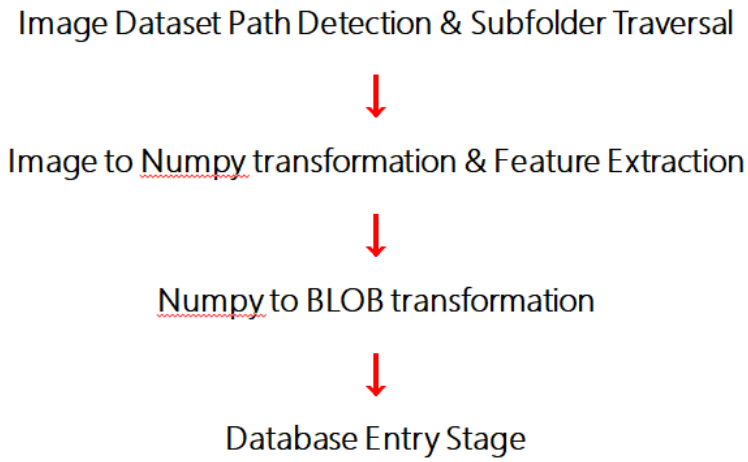
Student: Elshan Naghizade

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

Date: 6 August, 2023

INTRODUCTION

This report summarizes the development of the Python library to transform image datasets into query-able SQL tables and encompasses the decisions made at different stages that make up the project. Those stages are shown below:



Each of these stages will be scrutinized in their respective section with the breakdown of its current implementation. Additionally, methods to demonstrate the use cases for the library will be inspected.

IMAGE DATASET PATH DETECTION & SUBFOLDER TRAVERSAL

Initial Ideas	Final Version
Combining all the image subfolders into a single root folder and treating any folder tree as a large list of images was the simplest solution. However, the conventional machine learning practice, namely, dividing the datasets into train/test/dev sets should have been taken into account.	The current solution is to be able to divide folder tree into subfolders at the given level which is achieved by means of the built-in "os" module in Python. Consequently, the subfolder paths are then fed into the library's main class.

This stage consists of two functions that manage directories and files together. In the separate_folder_tree function, the directory tree of a specified path is traversed. First, the path is normalized and the depth of the root is determined. As each sub-directory is walked through using os.walk(), the depth of the current directory is compared to the desired level. If they match, the directory path is added to the subfolders list, which is returned at the end. In the get_file_paths function, all file paths in a specified directory are retrieved. During the traversal of the directory, every file path is constructed and appended to the file_paths list. This list is then returned.

```
import os

def separate_folder_tree(path, level):
    subfolders = []

    path = os.path.normpath(path)
    root_depth = path.count(os.sep)

    for dirpath, dirnames, filenames in os.walk(path):
        depth = dirpath.count(os.sep) - root_depth
        if depth == level:
            subfolders.append(dirpath)

    return subfolders

def get_file_paths(directory):
    file_paths = []
    for root, dirs, files in os.walk(directory):
        for file_name in files:
            file_path = os.path.join(root, file_name)
            file_paths.append(file_path)
    return file_paths
```

IMAGE TO NUMPY TRANSFORMATION & FEATURE EXTRACTION

Initial Ideas	Final Version
The early concept can be summarized as simply taking the image file and transforming it into a numpy array and having its features derived in numpy as well. Later to are saved as text files and their file paths are entered into the database as strings. However, this approach would not allow directly running queries on the image datasets, moreover, introducing integrity risks as a file could have been simply unavailable.	The final version solves both of those problems by employing database-level supported Binary Large OBjects (BLOB) allowing direct queries and inherent integrity constraints overwatched by the RDBMS.

```
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))

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()
```


NOTE: ***Please, take into account that these features are just examples to fill the Features table, thus, it is up to the Computer Vision engineer to extract the needed features which then by means of this library can be turned from a numpy array into a BLOB record in the database.***

In the provided `update_image_and_features` method, the database records related to an image, identified by its path, are updated. Firstly, the image is read from the given path using the OpenCV library, resulting in a matrix representation of the image. Using a connection to a database, two update operations are executed. In the first operation, attributes of the image, including its path, height, width, number of channels, and file extension, are updated in the `images` table. For this, the dimensions of the image matrix are extracted, and the file extension is derived from the image path.

In the second operation, the computed features of the image, namely its color histogram, edges, and corners, are updated in the `features` table. Before updating, these features are processed by respective methods, and their outputs are converted into a blob format suitable for database storage. Lastly, after both update operations are executed, the changes are committed to the database.

NUMPY TO BLOB TRANSFORMATION

Initial Ideas	Final Version
The early concept was to transform text files containing the values from numpy arrays. This intermediate was dropped when the database-level Numpy-blob conversion was discovered.	The current codebase utilizes numpy's internal methods to perform two-way transformations which is done with minimal preprocessing since the numpy arrays are saved in the blobs in the "as-is" fashion, unlike the text files. Therefore, the processing overhead of text-to-numerical conversion is completely avoided.

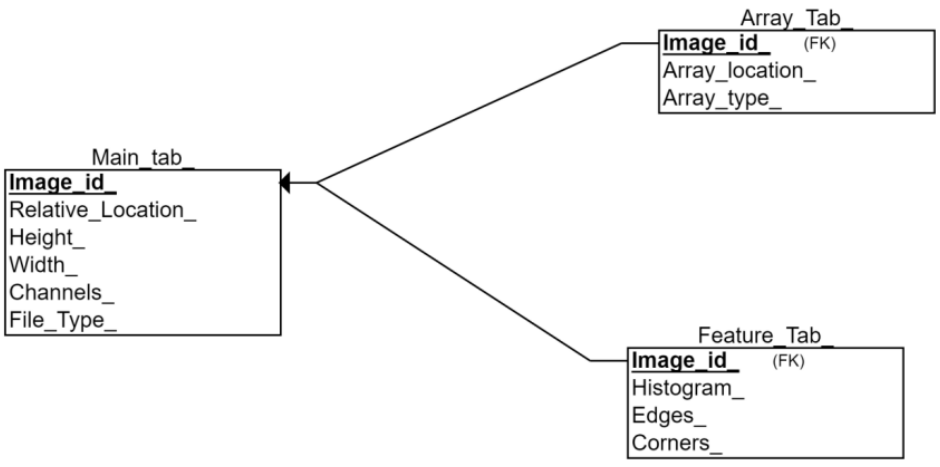
```
def get_image_and_features(self, image_path):
    with self.connect() as cursor:
        cursor.execute("SELECT images.id, path, height, width, channels, extension, array, hist, edges, corners FROM images "
                       "JOIN features ON images.id = features.image_id WHERE path=?", (image_path,))
        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

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
```

The essence of the current implementation is when an image's path is taken as an input argument to the get_image_and_features method. Within this method, a connection to the database is established and an SQL query is executed to fetch the associated image and feature details. If a match is found in the database, the relevant data is extracted and transformed from binary blobs into numpy arrays using the helper methods blob_to_numpy_array and numpy_array_to_blob. In blob_to_numpy_array, a binary blob is converted into a numpy array. If a shape is provided, the array is reshaped accordingly. Conversely, in numpy_array_to_blob, a numpy array is converted into a binary blob format suitable for storage in the database. If no image path match is found in the database, null values are returned for all attributes.

DATABASE ENTRY STAGE

Initial Ideas	Final Version
The simplistic database schema where all the attributes were supposed to be stored inside a single table would result in a highly unnormalized database. Thus, the actual schema was designed with the main idea of having a separate table for each entity.	The current database schema consists of 3 tables: • **Main_Tab** – general image file information such as width, height, file_path, color channels, and array data type. • **Array_tab** – Image BLOB itself • **Features_Tab –** which should be used to store the features that the Computer Vision engineer would extract and store as query-able BLOBs.

Database Schema Diagram:



USAGE EXAMPLE

NOTE: Some universal database manipulation methods were implemented to showcase the usage of the library:

```
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()

def get_all_images(self):
    with self.connect() as cursor:
        cursor.execute("SELECT path FROM images")
        return [row[0] for row in cursor.fetchall()]
```


The operations for updating, deleting, and retrieving images and their features from an SQLite database were developed.

update_image_and_features - an image is read from a given path using OpenCV. A connection to the database is then established. SQL statements are executed to update the image and its associated features based on the given image path. Binary blobs of the image and its features, such as histograms, edges, and corners, are generated using helper methods and then stored in the database.

delete_image_and_features - records associated with the provided image path are removed from both the features and images tables in the database.

get_all_images - all image paths stored in the images table are retrieved. A connection to the database is established, and an SQL query is executed to fetch all image paths. A list of these paths is then returned.