Pencil Sketch Image Processing Project Documentation

Project Title

Image Dataset Conversion to Pencil Sketch using OpenCV

Course

Digital Image Processing

Tool/Language

Python (Google Colab)

Institute

Government Graduate College Burewala

Submitted By

Ayesha Bibi & Iman Zahid

Submitted To

Prof. Rimsha Kiran

Department

Computer Science

Abstract

This project presents a simple and effective approach for converting a dataset of color images into pencil sketch representations using Python and OpenCV. The main objective of the project is to apply fundamental digital image processing techniques, including grayscale conversion, Gaussian blurring, edge detection using the Canny algorithm, and image inversion.

The dataset images are first extracted and loaded automatically, after which each image is processed step by step to generate a pencil sketch effect. The results are visualized using the Matplotlib library in both individual and grid formats. This project helps in understanding basic image preprocessing, edge detection, and artistic image transformation techniques, which are widely used in computer vision and image editing applications.

Figure 1: Sample Original Image from Dataset

Introduction

Digital Image Processing (DIP) is an important field of computer science that focuses on enhancing, analyzing, and transforming digital images using computational techniques. It is widely used in various real-world applications such as medical imaging, satellite imaging, computer vision, surveillance systems, and multimedia processing. By applying image processing algorithms, meaningful information can be extracted from images in an efficient manner.

One popular application of digital image processing is the creation of artistic effects, such as pencil sketch transformations. Pencil sketch effects convert color or grayscale images into sketch-like representations by emphasizing edges and removing unnecessary details. These effects are commonly used in photo editing software, mobile applications, and creative media to give images a hand-drawn appearance.

In this project, a simple and efficient image processing pipeline is implemented using Python and OpenCV to convert real images into pencil sketches. The process involves grayscale conversion, noise reduction using Gaussian blur, edge detection using the Canny algorithm, and inversion of edges to produce the final sketch effect. The project demonstrates how basic image processing techniques can be combined to achieve visually appealing results while maintaining computational efficiency.

Figure 2: Sample of output image

Objectives

The main objectives of this project are as follows:

- To load and manage an image dataset efficiently using Python.
- To perform image preprocessing techniques such as grayscale conversion and noise reduction.
- To apply edge detection techniques for identifying important image features.
- To generate pencil sketch effects from original color images using image inversion.
- To visualize the original and processed images for better analysis and comparison.

- To understand the practical implementation of basic digital image processing concepts using OpenCV.

Tools and Libraries Used

This project is implemented using Python along with several powerful libraries that support image processing and data visualization. Each tool and library plays an important role in achieving the desired pencil sketch effect.

- **Python:** Python is used as the main programming language for this project due to its simplicity, readability, and wide support for image processing libraries.
- **OpenCV (cv2):** OpenCV is a widely used open-source computer vision library. In this project, it is used for reading images, converting color images to grayscale, applying Gaussian blur, performing edge detection, and generating the pencil sketch effect.
- **Matplotlib:** Matplotlib is used for displaying and visualizing images. It helps in presenting original images, grayscale images, edge-detected images, and final pencil sketch outputs in an organized manner.
- **OS Module:** The OS module is used for handling directories and files. It helps in recursively scanning the dataset folder and locating all image files automatically.
- **Zipfile Module:** The zipfile module is used to extract the compressed dataset. This allows the program to access all images without manual extraction.

Dataset Description

The dataset used in this project consists of multiple digital images stored in a compressed ZIP file. The images may be organized into different subfolders within the dataset directory. This structure allows better organization and management of large image collections.

The dataset includes images in commonly used formats such as **JPG**, **JPEG**, and **PNG**. After extraction, the dataset directory is scanned recursively using Python to ensure that all images are detected, regardless of their folder location. This approach makes the system flexible and capable of handling datasets with complex folder structures.

The extracted images are used as input for the image processing pipeline, where each image is read, preprocessed, and converted into a pencil sketch representation. The dataset provides a suitable foundation for demonstrating the effectiveness of basic digital image processing techniques.

Figure 3: Dataset Folder Structure

System Architecture

The system architecture of this project is designed as a sequential image processing pipeline. This structured approach ensures clarity, simplicity, and efficient processing of the image dataset.

The pipeline begins with dataset extraction, where the compressed dataset is extracted into a working directory. Next, image loading is performed to read all image files from the dataset, including those stored in subfolders. In the preprocessing stage, images are converted to grayscale and smoothed using Gaussian blur to reduce noise.

After preprocessing, edge detection is applied using the Canny edge detection algorithm to highlight important boundaries and features within the images. The detected edges are then processed in the pencil sketch generation stage by inverting the edge image to create a sketch-like appearance. Finally, the visualization stage displays the original and processed images using Matplotlib for analysis and comparison.

Figure 4: System Architecture Steps:

Dataset ZIP

↓

Extraction

↓

Image Loading

↓

Preprocessing

↓

Edge Detection

↓

Pencil Sketch

↓

Visualization

Dataset Extraction

Step 1 – Dataset Extraction

The first step of the system involves extracting the image dataset from a compressed ZIP file. This is performed using Python's built-in **zipfile** module. Dataset extraction is an important step because it makes all image files accessible for further processing.

Once the ZIP file is extracted, all images stored inside the archive, including those in subfolders, become available in the specified directory. This automated extraction process removes the need for manual file handling and ensures that the dataset is ready for image loading and processing.

```
# Step 1: Unzip the dataset
zip_path = "/content/dataset.zip"
extract_path = "/content/dataset"

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)

print("Dataset extracted to:", extract_path)
```

Figure 5: Dataset Extraction Using Zipfile Module

Image File Collection

Step 2 – Image File Collection

After extracting the dataset, the next step is to locate and collect all image files from the dataset directory. For this purpose, Python's **OS walk** function is used to recursively traverse through the main dataset folder and its subfolders.

During this process, the program checks the file extensions of each file and selects only valid image formats such as **JPG**, **JPEG**, and **PNG**. All detected image file paths are stored in a list, which allows efficient access to images during further processing. This recursive approach ensures that no image is missed, even if the dataset contains multiple nested folders.

```
# Step 2: Recursively get all image files
image_files = []
for root, dirs, files in os.walk(extract_path):
    for f in files:
        if f.lower().endswith(('.jpg', '.jpeg', '.png')):  # lowercase check
            image_files.append(os.path.join(root, f))

print(f"Found {len(image_files)} images.")
```

Figure 6: Recursive Image File Collection Using OS Walk

Image Reading

Step 3 – Image Reading

In this step, the collected image files are read using the **imread()** function provided by the OpenCV library. This function loads each image into memory so that further image processing operations can be applied.

OpenCV reads images in **BGR (Blue, Green, Red)** color format by default. However, for correct visualization using Matplotlib, the images are converted from BGR format to **RGB (Red, Green, Blue)** format. This conversion ensures that the displayed images appear with accurate colors during visualization.

Figure 7: Original Color Image Read Using OpenCV

Grayscale Conversion

Grayscale conversion is an important preprocessing step in digital image processing. In this step, each color image is converted into a grayscale image using OpenCV. A grayscale image contains only intensity information and removes color details, which simplifies further image processing operations.

Converting images to grayscale reduces computational complexity and helps in focusing on important structural features such as edges and shapes. This step is especially useful before applying edge detection algorithms, as it improves accuracy and processing efficiency.

Figure 8: Grayscale Image After Conversion

Gaussian Blur

Gaussian Blur is a commonly used image smoothing technique in digital image processing. In this step, Gaussian Blur is applied to the grayscale image to reduce noise and small unwanted details. This smoothing process helps in minimizing sudden intensity variations in the image.

Applying Gaussian Blur improves the performance of edge detection algorithms by preventing false edges caused by noise. As a result, the edges detected in the next step become more accurate and clear, which contributes to a better pencil sketch effect.

Figure 9: Image After Applying Gaussian Blur

Edge Detection (Canny Algorithm)

Edge detection is a key step in digital image processing that helps identify important boundaries and structural features within an image. In this project, the **Canny edge detection algorithm** is used to detect strong and significant edges from the preprocessed image.

The Canny algorithm works by identifying areas with rapid intensity changes and highlighting them as edges. It is widely used because it provides good edge detection results while reducing noise and false detections. The output of this step is a binary image that clearly shows the outlines of objects present in the image, which forms the basis for generating the pencil sketch effect.

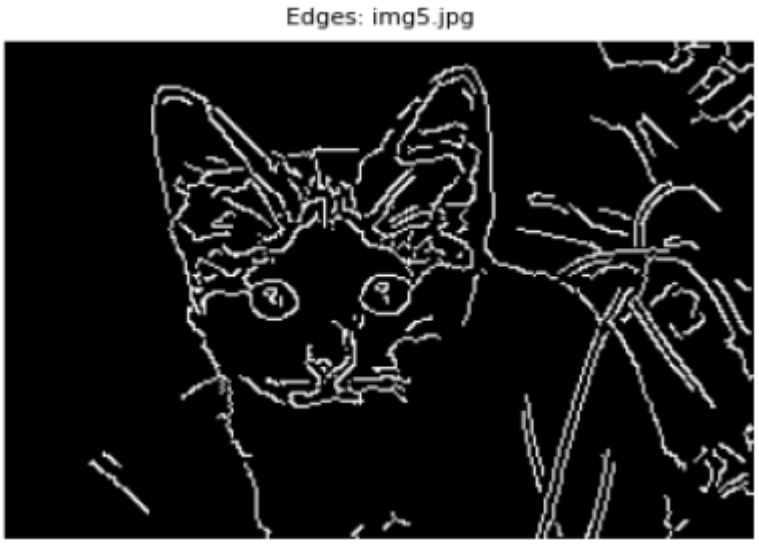


Figure 10: Edge Detection Output Using Canny Algorithm

Pencil Sketch Effect

The pencil sketch effect is generated by inverting the edge-detected image obtained from the Canny algorithm. Image inversion changes the pixel values in such a way that the detected edges appear darker on a light background, similar to hand-drawn pencil sketches.

This step enhances the visual appearance of the edges and gives the image an artistic sketch-like look. By combining edge detection and inversion, the project successfully transforms real images into visually appealing pencil sketches using simple image processing techniques.

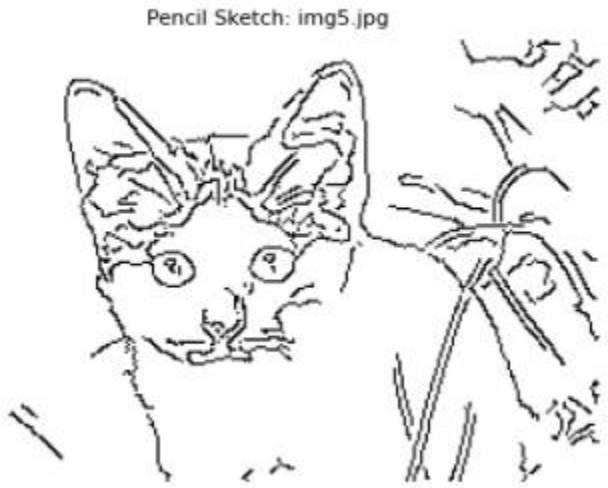


Figure 11: Final Pencil Sketch Output Image

Weight Calculation

In this step, the weight of each pencil sketch image is calculated by counting the number of non-zero pixels present in the sketch. This is done using OpenCV's pixel counting functions. Non-zero pixels represent the visible sketch lines in the image.

The calculated weight provides a simple numerical measure of sketch density. A higher weight indicates that the sketch contains more edge information, while a lower weight shows fewer detected edges. This analysis helps in comparing different sketch outputs and understanding the effect of image processing parameters on the final result.

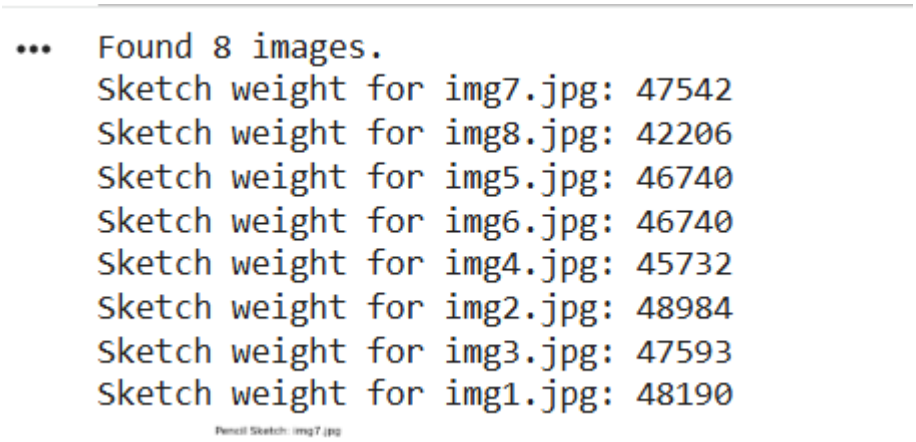


Figure 12: Console Output Showing Sketch Weight Calculation

Code Explanation

The code used in this project is written in a modular and well-structured manner to ensure readability and ease of understanding. Each major step of the image processing pipeline is clearly separated, including dataset extraction, image loading, preprocessing, edge detection, pencil sketch generation, and visualization.

Proper comments are included in the code to explain the purpose of each section and function. This makes the code easy to follow, modify, and extend for future improvements. The use of standard Python libraries such as OpenCV, Matplotlib, and OS ensures that the implementation remains efficient and compatible with different datasets.

Overall, the modular structure of the code helps in debugging, testing, and maintaining the project, while also making it suitable for educational and practical use.

1.Import important libraries and un zip dataset of images

```
import zipfile
import os
import cv2
import matplotlib.pyplot as plt

# Step 1: Unzip the dataset
zip_path = "/content/dataset.zip"
extract_path = "/content/dataset"

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)

print("Dataset extracted to:", extract_path)
```

2.Read and display all images

```
# Step 2: Recursively get all image files
image_files = []
for root, dirs, files in os.walk(extract_path):
    for f in files:
        if f.lower().endswith(('.jpg', '.jpeg', '.png')):  # lowercase check
            image_files.append(os.path.join(root, f))
print(f"Found {len(image_files)} images.")

# Step 3: Loop through and display each image
for img_path in image_files:
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(img)
    plt.title(f"Original Image: {os.path.basename(img_path)}")
    plt.axis('off')
    plt.show()
```

3.Convert original images into gray scale

```
image_files = []
for root, dirs, files in os.walk(extract_path):
    for f in files:
        if f.lower().endswith(('.jpg', '.jpeg', '.png')):
            image_files.append(os.path.join(root, f))

print(f"Found {len(image_files)} images.")

# Step 3: Loop through and display each image in grayscale
for img_path in image_files:
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # convert to grayscale

    plt.imshow(gray, cmap='gray')
    plt.title(f"Gray Image: {os.path.basename(img_path)}")
    plt.axis('off')
```

4.Applying Gaussian Blur to Grayscale Image for a Softened Effect

```
plt.figure(figsize=(15, 10))  # adjust figure size
for i, img_path in enumerate(image_files):
    img = cv2.imread(img_path)
# Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
 # Apply Gaussian blur
    blur = cv2.GaussianBlur(gray, (5, 5), 0)
 # Plot in a grid
    plt.subplot(2, 4, i+1)  # change 2,4 if more images
    plt.imshow(blur, cmap='gray')
    plt.title(f"{os.path.basename(img_path)}", fontsize=8)
    plt.axis('off')

plt.tight_layout()
plt.show()
```

5.Detecting Edges in the Blurred Image using Canny Edge Detection

```
num_images = len(image_files)
cols = 4  # number of columns in grid
rows = (num_images + cols - 1) // cols  # compute rows needed
plt.figure(figsize=(15, 5 * rows))  # adjust figure size
for i, img_path in enumerate(image_files):
    img = cv2.imread(img_path)
# Apply Canny edge detection
    edges = cv2.Canny(blur, 60, 100)
 # Plot in grid
    plt.subplot(rows, cols, i+1)
    plt.imshow(edges, cmap='gray')
    plt.title(f"Edges: {os.path.basename(img_path)}", fontsize=8)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

6.Creating a Pencil Sketch Effect by Inverting Edges in the Image

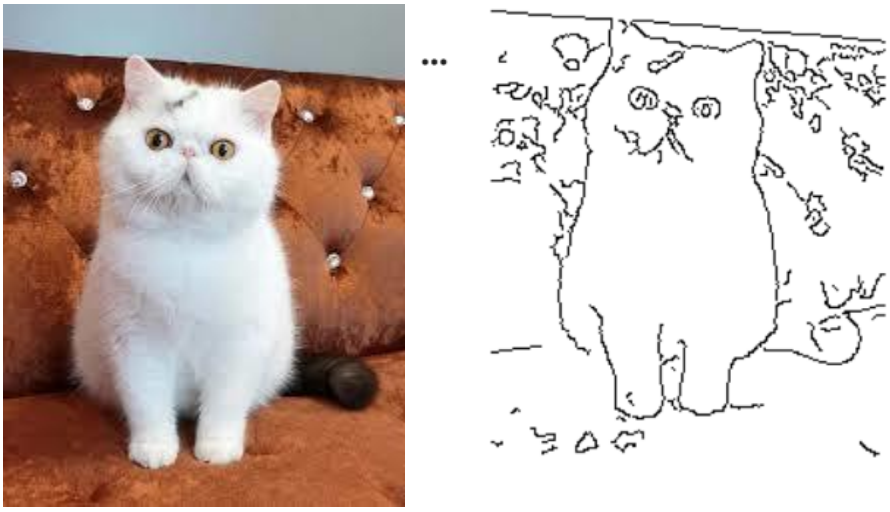
```
image_files = []
for root, dirs, files in os.walk(extract_path):
    for f in files:
        if f.lower().endswith(('.jpg', '.jpeg', '.png')):
            image_files.append(os.path.join(root, f))

print(f"Found {len(image_files)} images.")

# Step 3: Apply Pencil Sketch effect to all images and display
num_images = len(image_files)
cols = 4  # number of columns in grid
rows = (num_images + cols - 1) // cols  # compute number of rows

plt.figure(figsize=(15, 5 * rows))

for i, img_path in enumerate(image_files):
    # convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (5, 5), 0)
    edges = cv2.Canny(blur, 60, 100)
    sketch = cv2.bitwise_not(edges)

    weight = cv2.countNonZero(sketch)
    print(f"Sketch weight for {os.path.basename(img_path)}: {weight}")

    # Display sketch
    plt.subplot(rows, cols, i+1)
    plt.imshow(sketch, cmap='gray')
    plt.title(f"Pencil Sketch: {os.path.basename(img_path)}", fontsize=8)
    plt.axis('off')

plt.tight_layout()
plt.show()
```

Results and Discussion

The results obtained from this project show that the applied image processing techniques successfully convert color images into pencil sketch representations. The generated sketches clearly highlight the main edges and structural features of the original images, giving them a hand-drawn appearance.

The quality of the pencil sketch output depends on several factors, such as the parameters used for Gaussian blur and the threshold values selected for the Canny edge detection algorithm. Proper selection of these parameters results in clear and visually appealing sketches, while inappropriate values may produce weak or noisy edges. Overall, the results demonstrate that basic digital image processing techniques can effectively be used to create artistic image transformations.

Figure 13: Comparison of Original Image and Pencil Sketch Output



Applications

The pencil sketch image processing technique developed in this project has several practical applications in different fields. Some of the major applications are listed below:

- **Photo Editing Applications:** Pencil sketch effects are widely used in photo editing software and mobile applications to enhance images and provide artistic visual effects.
- **Artistic Image Filters:** This technique can be used to create artistic filters that transform normal photographs into hand-drawn style sketches.
- **Feature Extraction:** Edge-based sketch images are useful for extracting important features such as object boundaries and shapes in computer vision tasks.
- **Computer Vision Preprocessing:** Pencil sketch and edge-detected images can be used as preprocessing steps in various computer vision and pattern recognition systems.

Advantages

The proposed pencil sketch image processing approach offers several advantages, making it suitable for educational and practical applications:

- **Simple Implementation:** The project uses basic image processing techniques that are easy to understand and implement using Python and OpenCV.
- **Fast Processing:** The algorithm is computationally efficient and processes images quickly without requiring heavy system resources.
- **Works on Multiple Images:** The system can handle and process multiple images automatically from a dataset, making it scalable and flexible.
- **No Need for Deep Learning:** The approach does not rely on complex deep learning models, reducing training time, data requirements, and computational cost.

Limitations

Despite producing visually appealing pencil sketch effects, the proposed system has certain limitations:

• The method is **sensitive to noise**, and poor-quality images may produce unwanted edges.

• **Fixed threshold values** in the Canny edge detector may not work optimally for all types of images.

• The sketch effect offers **limited artistic variation** compared to advanced or deep learning–based approaches.

• The technique mainly focuses on **edge information** and may lose texture details.

Conclusion

This project successfully demonstrates how basic digital image processing techniques can be combined to convert color images into visually appealing pencil sketch representations. By applying grayscale conversion, Gaussian blur, edge detection, and image inversion, the system produces sketch-like images that closely resemble hand-drawn pencil sketches.

The implementation using Python and OpenCV proves to be simple, efficient, and effective for processing multiple images from a dataset. The results show that even without using complex deep learning models, meaningful and artistic image transformations can be achieved. Overall, this project provides a strong foundation for understanding core image processing concepts and can be further extended to develop advanced image processing and computer vision applications.

.