**IMAGE CHALLENGE**

Assignment report submitted to

Graduate School at

The University of Arkansas at Little Rock

in partial fulfillment of

requirements for the graduate course of

202510 Spring: Data Science - Technologies (9U1)

March 2025

by

Naga Venkata Ramkiran Hota

Instructors

Dr. Elizabeth Pierce

and

Dr. Serhan Dagtas

ABSTRACT

This assignment consists of several challenge ideas for working with images. The three challenges presented are “Implementation of image processing operations” or “Multimodal Content Retrieval” or “Scene Cut Detection”.

For this report, first challenge is chosen which is “Implementation of image processing operations” and its inputs and outputs (results) are provided, including the original color and gray scale images are attached.

For the entire assignment to complete all the tasks, Google Colab environment is used along with *PIL* and *google.colab* and *matplotlib.pyplot* libraries. Additionally, *numpy*, *scipy.stats* libraries are also used when required for the individual tasks.

IMAGES USED IN THE PROJECT

Color Image

A rock arch in the desert with Arches National Park in the background

Description automatically generated

Gray Scale Image

A rock arch in the desert

Description automatically generated

TASKS WITH INPUTS AND OUTPUTS

**Task 1: Black and White**

Main Method:

from PIL import Image

from google.colab import files

import matplotlib.pyplot as plt

def convert\_to\_black\_and\_white(color\_image\_path):

    # Load the color image

    color\_image = Image.open(color\_image\_path)

    # Convert the image to RGB (in case it's not)

    color\_image = color\_image.convert('RGB')

    # Create a new image for the black and white output

    bw\_image = Image.new('RGB', color\_image.size)

    # Process each pixel

    for x in range(color\_image.width):

        for y in range(color\_image.height):

            # Get the RGB values

            r, g, b = color\_image.getpixel((x, y))

            # Calculate the average

            average = int((r + g + b) / 3)

            # Set the new pixel value to the average for all channels

            bw\_image.putpixel((x, y), (average, average, average))

    return bw\_image

In the above code snippet, there is a main method, *convert\_to\_black\_and\_white* which accepts are color image path from the

user’s computer. The color image will then be loaded and converted to RGB if it is not already. From this RGB a black and white image created with the same size which is a result of the processing each pixel.

Example usage:

# Example usage

# Upload an image

uploaded = files.upload()

# Load the color image

color\_image\_path = list(uploaded.keys())[0]  # Get the uploaded file name

# Convert to black and white

bw\_image = convert\_to\_black\_and\_white(color\_image\_path)

# Display the original and black and white images

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.title('Original Color Image')

plt.imshow(Image.open(color\_image\_path))

plt.axis('off')

plt.subplot(1, 2, 2)

plt.title('Black and White Image')

plt.imshow(bw\_image)

plt.axis('off')

plt.show()

# Save the black and white image

bw\_image.save('task1\_result.png')

User can execute this method like the sample above and see the converted image which is in gray scale.

Output:

A screenshot of a black and white image

Description automatically generated

**Task 2: Gaussian filter**

Main Methods:

from PIL import Image

import numpy as np

from google.colab import files

import matplotlib.pyplot as plt

import scipy.stats as st

def gaussian\_kernel(size, sigma=1):

    """Generates a Gaussian kernel."""

    interval = (2\*sigma+1.)/(size)

    x = np.linspace(-sigma-interval/2., sigma+interval/2., size+1)

    kern1d = np.diff(st.norm.cdf(x))

    kernel\_raw = np.sqrt(np.outer(kern1d, kern1d))

    kernel = kernel\_raw/kernel\_raw.sum()

    return kernel

def apply\_gaussian\_filter(image, kernel):

    """Applies a Gaussian filter to the image."""

    # Get image dimensions

    width, height = image.size

    # Create an output image

    output\_image = Image.new("L", (width, height))

    # Convert image to numpy array

    image\_array = np.array(image)

    # Get kernel dimensions

    k\_size = kernel.shape[0]

    k\_half = k\_size // 2

In the above code snippet, there are two methods, first one is *gaussian\_kernel* which generates a 7x7 gaussian kernel and second one is *apply\_gaussian\_filter* which converts the image into a numpy array and its kernel dimensions. Once the dimensions are determined, the image and kernel are convolved which is the gaussian output image.

    # Convolve the kernel with the image

    for x in range(k\_half, width - k\_half):

        for y in range(k\_half, height - k\_half):

            # Extract the region of interest

            region = image\_array[y - k\_half:y + k\_half + 1, x - k\_half:x + k\_half + 1]

            # Apply the kernel

            filtered\_value = np.sum(region \* kernel)

            # Set the pixel value in the output image

            output\_image.putpixel((x, y), int(filtered\_value))

    return output\_image

Example usage:

# Example usage

# Upload an image

uploaded = files.upload()

# Load the color image

color\_image\_path = list(uploaded.keys())[0]  # Get the uploaded file name

image = Image.open(color\_image\_path).convert("L")  # Convert to grayscale

# Create a 7x7 Gaussian kernel

kernel\_size = 7

sigma = 1.0

gaussian\_k = gaussian\_kernel(kernel\_size, sigma)

# Apply the Gaussian filter

filtered\_image = apply\_gaussian\_filter(image, gaussian\_k)

User can execute this method like the sample above and see the converted gaussian filtered image.

# Display the original and filtered images

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.title('Original Grayscale Image')

plt.imshow(image, cmap='gray')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.title('Gaussian Filtered Image')

plt.imshow(filtered\_image, cmap='gray')

plt.axis('off')

plt.show()

# Save the filtered image

filtered\_image.save('task2\_gaussian\_result.png')

Output:

A screenshot of a picture of a stone arch

Description automatically generated

**Task 3: Thresholding**

Main Methods:

from PIL import Image

import numpy as np

from google.colab import files

import matplotlib.pyplot as plt

def generate\_histogram(image):

    """Generates a histogram of the image."""

    histogram = image.histogram()

    return histogram

def apply\_threshold(image, threshold):

    """Applies a binary threshold to the image."""

    # Create a new binary image

    binary\_image = image.point(lambda p: 255 if p > threshold else 0)

    return binary\_image

In the above code snippet, there are two methods, first one is *generate\_histogram* which generates a histogram and second one is *apply\_threshold* which applies a binary threshold to the newly created binary image. Threshold values can be changed based on the histogram analysis which is depicted in the example usage below.

Example usage:

# Example usage

# Upload an image

uploaded = files.upload()

# Load the color image

color\_image\_path = list(uploaded.keys())[0]  # Get the uploaded file name

User can execute this method like the sample above and see the histogram and the converted threshold image

image = Image.open(color\_image\_path).convert("L")  # Convert to grayscale

# Generate histogram

histogram = generate\_histogram(image)

# Plot the histogram

plt.figure(figsize=(10, 5))

plt.title('Histogram')

plt.xlabel('Pixel Intensity')

plt.ylabel('Frequency')

plt.bar(range(256), histogram, color='gray')

plt.show()

# Select a threshold value (you can change this based on your histogram analysis)

threshold\_value = 128  # Example threshold value

# Apply the threshold

binary\_image = apply\_threshold(image, threshold\_value)

# Display the original and binary images

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.title('Original Grayscale Image')

plt.imshow(image, cmap='gray')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.title('Binary Image after Thresholding')

plt.imshow(binary\_image, cmap='gray')

plt.axis('off')

plt.show()

# Save the binary image

binary\_image.save('task3\_threshold\_result.png')

Output:

A screenshot of a computer

Description automatically generated

**Task 4: Prewitt derivative estimate**

Main Methods:

from PIL import Image

import numpy as np

from google.colab import files

import matplotlib.pyplot as plt

def apply\_prewitt(image):

    """Applies Prewitt operator to the image."""

    # Define Prewitt kernels

    prewitt\_horizontal = np.array([[-1, 0, 1],

                                   [-1, 0, 1],

                                   [-1, 0, 1]], dtype=np.float32)

    prewitt\_vertical = np.array([[-1, -1, -1],

                                 [0, 0, 0],

                                 [1, 1, 1]], dtype=np.float32)

    # Convert image to numpy array

    image\_array = np.array(image)

    # Get image dimensions

    width, height = image.size

    k\_size = 3

    k\_half = k\_size // 2

    # Create output images

    horizontal\_edges = np.zeros((height, width), dtype=np.float32)

    vertical\_edges = np.zeros((height, width), dtype=np.float32)

    # Convolve the Prewitt kernels with the image

    for x in range(k\_half, width - k\_half):

        for y in range(k\_half, height - k\_half):

            # Extract the region of interest

            region = image\_array[y - k\_half:y + k\_half + 1, x - k\_half:x + k\_half + 1]

            # Apply the horizontal kernel

            horizontal\_edges[y, x] = np.sum(region \* prewitt\_horizontal)

            # Apply the vertical kernel

            vertical\_edges[y, x] = np.sum(region \* prewitt\_vertical)

    # Combine the results

    edges = np.abs(horizontal\_edges) + np.abs(vertical\_edges)

    # Clip the values to be in the range [0, 255]

    edges = np.clip(edges, 0, 255).astype(np.uint8)

    return Image.fromarray(edges)

In the above code snippet, the main method is *apply\_prewitt* in which the Prewitt kernels defined and subsequently the image is converted into numpy array. Prewitt image is generated by convolving the Prewitt horizontal and vertical kernels and the numpy image array.

Example usage:

# Example usage

# Upload an image

uploaded = files.upload()

# Load the color image

color\_image\_path = list(uploaded.keys())[0]  # Get the uploaded file name

image = Image.open(color\_image\_path).convert("L")  # Convert to grayscale

# Apply the Prewitt operator

prewitt\_edges = apply\_prewitt(image)

# Display the original and edge-detected images

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.title('Original Grayscale Image')

plt.imshow(image, cmap='gray')

plt.axis('off')

plt.subplot(1, 2, 2)

plt.title('Prewitt Edge Detected Image')

plt.imshow(prewitt\_edges, cmap='gray')

plt.axis('off')

plt.show()

# Save the edge-detected image

prewitt\_edges.save('task4\_prewitt\_result.png')

User can execute this method like the sample above and see the Prewitt derivative images.

Output:

A screenshot of a computer

Description automatically generated