Q-1

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

Read the image

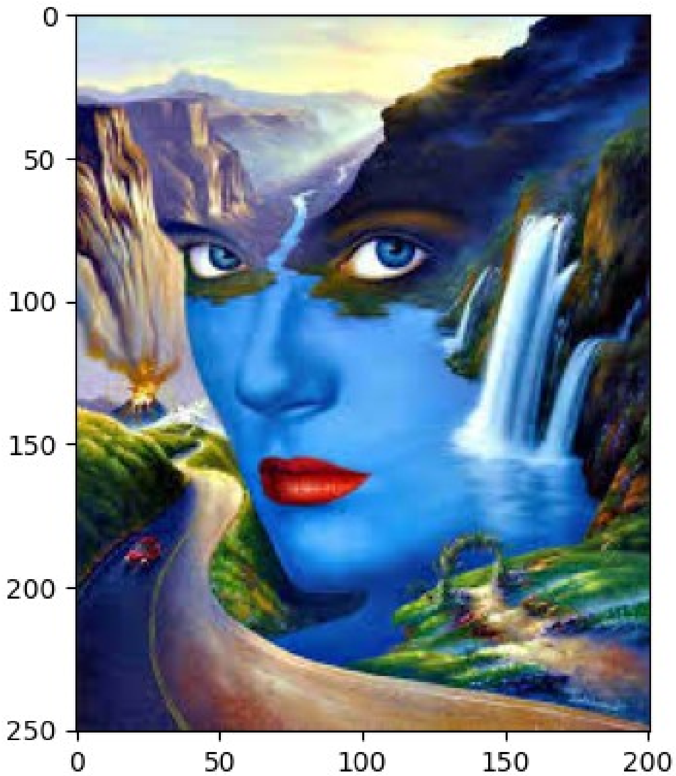
```python
image = cv2.imread(r"C:\Users\devan\OneDrive\Desktop\i1.jpeg")
```

Convert BGR to RGB

```python
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

Display the image

```python
plt.imshow(image_rgb)
# plt.axis('off')  # Turn off axis numbers
plt.show()
```



Multiplying by a constant

```
# multiplied_image = cv2.multiply(image, np.array([1.5]))
multiplied_image = image_rgb*1.5

plt.imshow(multiplied_image)
plt.title('Multiplied by Constant')
plt.axis('off')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers). Got range [0.0..382.5].
```

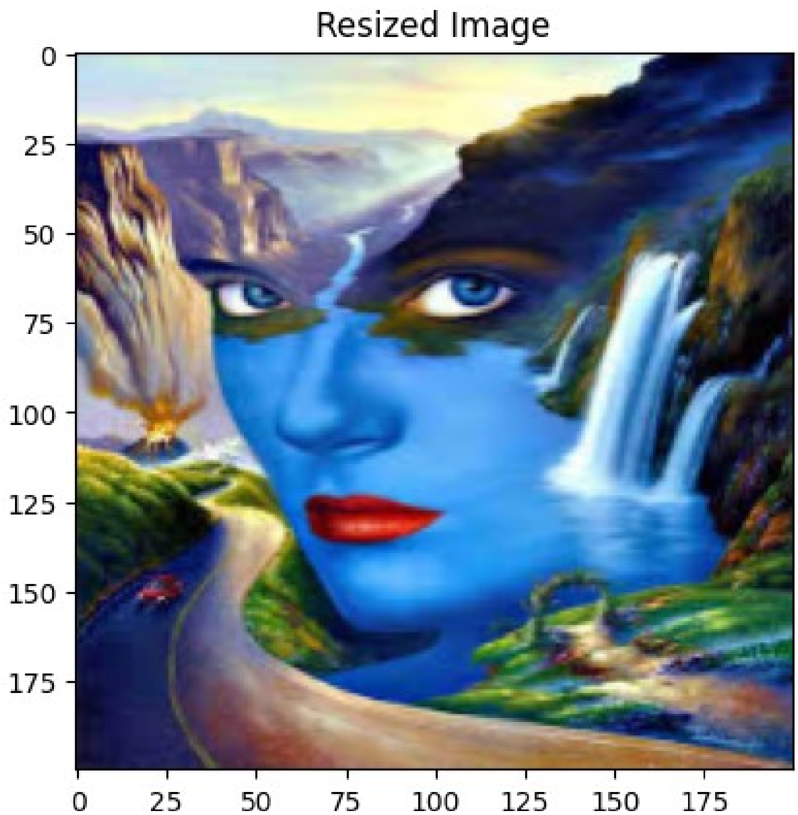
Dividing by a constant

```
divided_image = image_rgb/50

plt.imshow(divided_image)
plt.title('Divided by Constant')
plt.axis('off')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers). Got range [0.0..5.1].
```

Resize the image

```
resized_image = cv2.resize(image_rgb, (200, 200))

plt.imshow(resized_image)
plt.title('Resized Image')
# plt.axis('off')
plt.show()
```



Slicing a part of the image

```
sliced_image = image_rgb[50:200, 100:300]

plt.imshow(sliced_image)
plt.title('Sliced Part of the Image')
plt.axis('off')
plt.show()
```

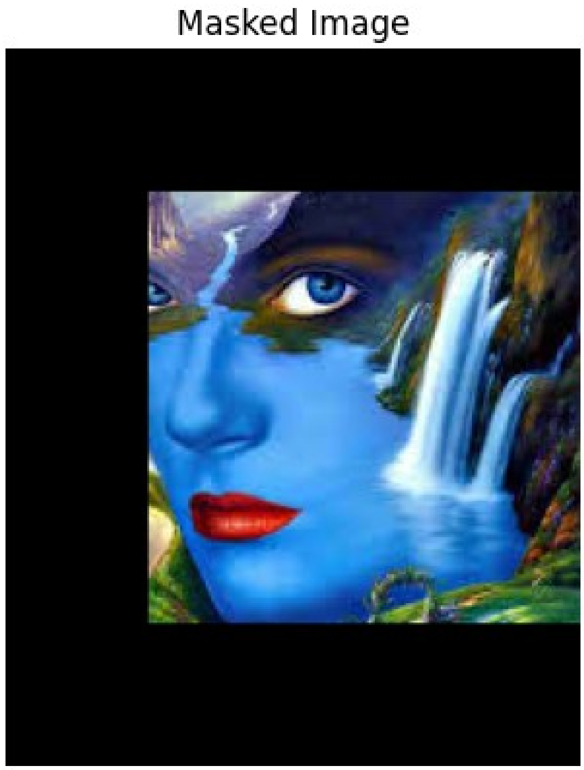
Sliced Part of the Image

Mask a part of image

```
# Create a mask to cover a rectangular area
mask = np.zeros(image_rgb.shape[:2], dtype="uint8")
cv2.rectangle(mask, (50, 50), (200, 200), 255, -1)
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

```
# Apply mask
masked_image = cv2.bitwise_and(image_rgb, image_rgb, mask=mask)

# Display
plt.imshow(masked_image)
plt.title('Masked Image')
plt.axis('off')
plt.show()
```



Add Two Images

```
image2 = cv2.imread(r"C:\Users\devan\OneDrive\Desktop\i2.jpeg")

image2 = cv2.resize(image2, (image.shape[1], image.shape[0]))

image2_rgb = cv2.cvtColor(image2, cv2.COLOR_BGR2RGB)

plt.imshow(image2_rgb)
plt.axis('off')  # Turn off axis numbers
plt.show()
```

```
# Simple addition of two images
added_image = cv2.add(image_rgb, image2_rgb)

plt.imshow(added_image)
plt.title('Added Image')
plt.axis('off')
plt.show()
```

Weighted Add Two Images

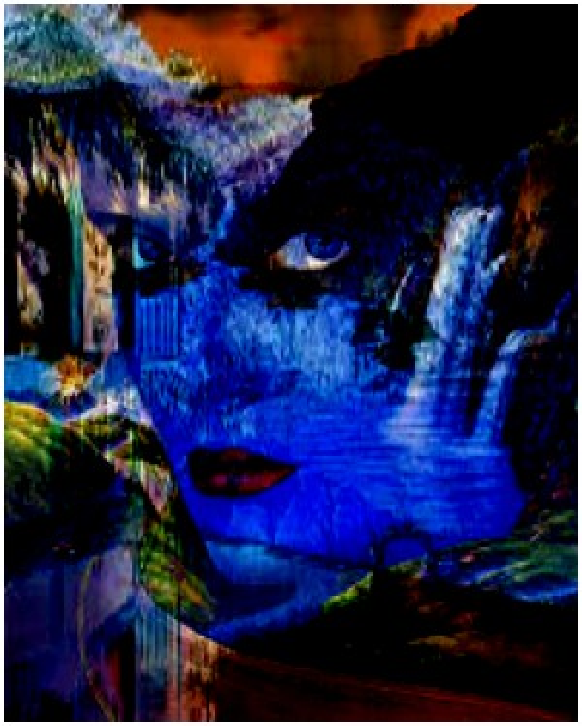
```
# Weighted addition of two images
weighted_image = cv2.addWeighted(image_rgb, 0.4, image2_rgb, 0.7, 0)

# Display
plt.imshow(weighted_image)
plt.title('Weighted Addition of Images')
plt.axis('off')
plt.show()
```

Weighted Addition of Images

Subtract Two Images

```
# Subtract one image from another
subtracted_image = cv2.subtract(image_rgb, image2_rgb)

plt.imshow(subtracted_image)
plt.title('Subtracted Image')
plt.axis('off')
plt.show()
```

Subtracted Image



Perform Logical Operations (AND, OR, NOT) on Images

```
# Logical AND
and_image = cv2.bitwise_and(image_rgb, image2_rgb)

# Display
plt.imshow(and_image)
plt.title('Logical AND')
plt.axis('off')
plt.show()
```

```
# Logical OR
or_image = cv2.bitwise_or(image_rgb, image2_rgb)

# Display
plt.imshow(or_image)
plt.title('Logical AND')
plt.axis('off')
plt.show()
```


```
# Logical NOT
not_image = cv2.bitwise_not(image_rgb, image2_rgb)

# Display
plt.imshow(not_image)
plt.title('Logical AND')
plt.axis('off')
plt.show()
```

Q-2

```
# Read the image in grayscale
image3 = cv2.imread(r"C:\Users\devan\OneDrive\Desktop\i1.jpeg",
cv2.IMREAD_GRAYSCALE)

# Display the original image
plt.imshow(image3, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.show()
```

Original Image

Mark Four and Eight Neighbors of a Pixel

```python
# Define a pixel position
x, y = 100, 100

# Create a copy of the image to draw neighbors
neighbor_image = cv2.cvtColor(image3, cv2.COLOR_GRAY2BGR)

# Four neighbors
four_neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]
for nx, ny in four_neighbors:
    cv2.circle(neighbor_image, (ny, nx), 2, (255, 0, 0), -1)  # Red for 4-neighbors

# Display
plt.imshow(cv2.cvtColor(neighbor_image, cv2.COLOR_BGR2RGB))
plt.title('Four and Eight Neighbors')
plt.axis('off')
plt.show()
```


```python
# Eight neighbors
eight_neighbors = [(x - 1, y - 1), (x - 1, y + 1), (x + 1, y - 1), (x
+ 1, y + 1),
    (x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)
]
for nx, ny in eight_neighbors:
    cv2.circle(neighbor_image, (ny, nx), 2, (0, 0, 255), -1)  # Green
for 8-neighbors

# Display
plt.imshow(cv2.cvtColor(neighbor_image, cv2.COLOR_BGR2RGB))
plt.title('Four and Eight Neighbors')
plt.axis('off')
plt.show()
```

Implement the Distance Formula

```
point1 = (100, 100)
point2 = (150, 150)

distance = np.sqrt((point2[0] - point1[0]) ** 2 + (point2[1] -
point1[1]) ** 2)
print(f"Distance between {point1} and {point2}: {distance}")

Distance between (100, 100) and (150, 150): 70.71067811865476
```

Image Negation

```
# Negate the image
negated_image = 255 - image3

# Display
plt.imshow(negated_image, cmap='gray')
plt.title('Negated Image')
plt.axis('off')
plt.show()
```

Log Transformation

```
# Log transformation
c = 255 / np.log(1 + np.max(image3))  # Scaling constant
log_transformed = c * np.log(1 + image3)
# log_transformed = np.array(log_transformed, dtype=np.uint8)
```

```
C:\Users\devan\AppData\Local\Temp\ipykernel_8564\3482941601.py:3:
RuntimeWarning: divide by zero encountered in log
  log_transformed = c * np.log(1 + image3)
```

```
# Display
plt.imshow(log_transformed, cmap='gray')
plt.title('Log Transformed Image')
plt.axis('off')
plt.show()
```

Power-Law (Gamma) Transformation

```
# Power-law (Gamma) transformation
gamma = 0.5  # Try different gamma values
c = 255 / (np.max(image3) ** gamma)
power_law_transformed = c * (image3 ** gamma)
# power_law_transformed = np.array(power_law_transformed,
dtype=np.uint8)

# Display
plt.imshow(power_law_transformed, cmap='gray')
plt.title('Power-Law Transformed Image')
plt.axis('off')
plt.show()
```

Power-Law Transformed Image

Q-3

Gray Level (Intensity) Slicing

```
# Define intensity range to highlight
min_val, max_val = 100, 200

# Slicing
sliced_image = np.where((image_rgb >= min_val) & (image_rgb <=
max_val), 255, 0)

# Display
plt.imshow(sliced_image, cmap='gray')
plt.title('Gray Level Slicing')
plt.axis('off')
plt.show()
```

Intensity Level Slicing with Background Retention

```
# Intensity slicing with background
sliced_with_background = np.where((image_rgb >= min_val) & (image_rgb
<= max_val), 255, image_rgb)

# Display
plt.imshow(sliced_with_background, cmap='gray')
plt.title('Intensity Slicing with Background')
plt.axis('off')
plt.show()
```

Boxplot Slicing (Using Intensity Quartiles)

```
# Calculate quartiles
q1, q3 = np.percentile(image_rgb, [25, 75])
interquartile_range = (image_rgb >= q1) & (image_rgb <= q3)

# Slicing based on quartiles
boxplot_sliced_image = np.where(interquartile_range, 255, 0)

# Display
plt.imshow(boxplot_sliced_image, cmap='gray')
plt.title('Boxplot Slicing')
plt.axis('off')
plt.show()
```

Boxplot Slicing

Bit Plane Slicing

```
# Bit plane slicing (e.g., extract the 7th and 6th bit planes)
bit_planes = []
for i in range(8):
    bit_plane = (image_rgb & (1 << i)) >> i
    bit_planes.append(bit_plane * 255)  # Scale for visibility

# Display a specific bit plane (e.g., the 7th bit plane)
plt.imshow(bit_planes[7], cmap='gray')
plt.title('7th Bit Plane')
plt.axis('off')
plt.show()
```

Histogram Equalization

```
# Histogram equalization
equalized_image = cv2.equalizeHist(image3)

# Display
plt.imshow(equalized_image, cmap='gray')
plt.title('Histogram Equalized Image')
plt.axis('off')
plt.show()
```

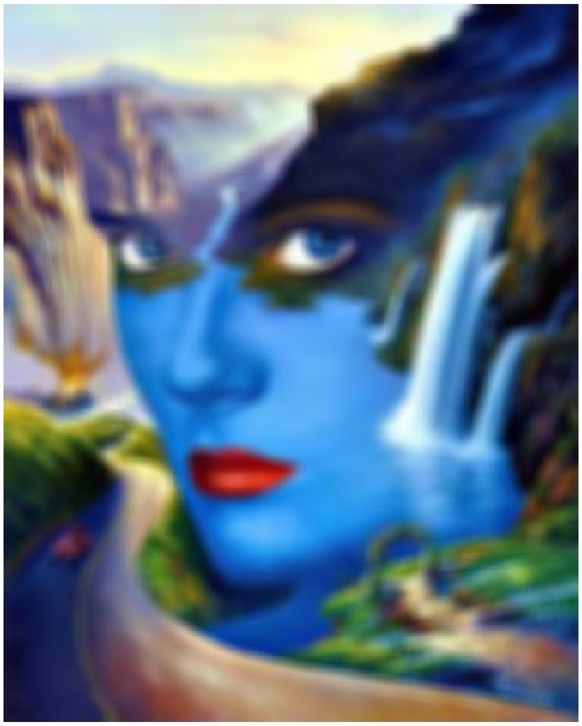
Histogram Equalized Image

Q-4

Box Filter

```
box_filtered = cv2.boxFilter(image_rgb, -1, (5, 5))

# Display result
plt.imshow(box_filtered)
plt.title('Box Filter')
plt.axis('off')
plt.show()
```

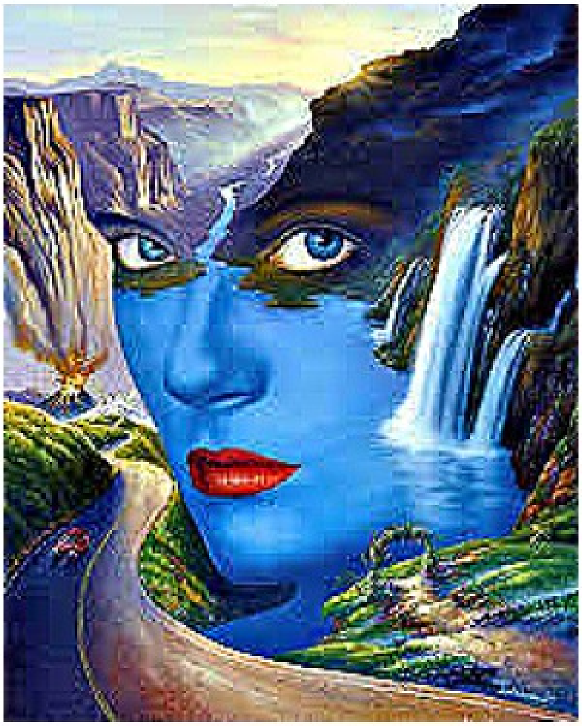
Box Filter



2D Filter (Arbitrary Kernel)

```
# Define a sharpening kernel
kernel = np.array([[0, -1, 0],
                   [-1, 5, -1],
                   [0, -1, 0]])

# Apply the filter using filter2D
filtered_image = cv2.filter2D(image_rgb, -1, kernel)

# Display result
plt.imshow(filtered_image)
plt.title('2D Filter (Sharpening)')
plt.axis('off')
plt.show()
```

2D Filter (Sharpening)



Gaussian Blur

```
# Apply Gaussian Blur
gaussian_blur = cv2.GaussianBlur(image_rgb, (5, 5), sigmaX=1)

# Display result
plt.imshow(gaussian_blur)
plt.title('Gaussian Blur')
plt.axis('off')
plt.show()
```

Gaussian Blur

Median Blur

```
# Apply Median Blur
median_blur = cv2.medianBlur(image_rgb, 5)

# Display result
plt.imshow(median_blur)
plt.title('Median Blur')
plt.axis('off')
plt.show()
```

Median Blur

Custom Convolution Using a Kernel

```
def apply_custom_convolution(image, kernel):
    # Convert to grayscale for simplicity
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Get the dimensions of the image and kernel
    image_h, image_w = gray_image.shape
    kernel_h, kernel_w = kernel.shape

    # Compute padding width
    pad_h, pad_w = kernel_h // 2, kernel_w // 2

    # Pad the image with zeros on all sides
    padded_image = np.pad(gray_image, ((pad_h, pad_h), (pad_w, 
pad_w)), mode='constant', constant_values=0)

    # Initialize output image
    output_image = np.zeros_like(gray_image)

    # Perform convolution
    for i in range(image_h):
        for j in range(image_w):
            # Extract the region of interest
```

```
            region = padded_image[i:i + kernel_h, j:j + kernel_w]
            # Apply the kernel (element-wise multiplication and
summing)
            output_image[i, j] = np.sum(region * kernel)

    # Clip values to be in valid grayscale range
    output_image = np.clip(output_image, 0, 255)

    return output_image

# Define a sample kernel (e.g., edge detection kernel)
sample_kernel = np.array([[1, 0, -1],
                          [0, 0, 0],
                          [-1, 0, 1]])

# Apply custom convolution
convolution_result = apply_custom_convolution(image_rgb,
sample_kernel)

# Display result
plt.imshow(convolution_result, cmap='gray')
plt.title('Custom Convolution Result')
plt.axis('off')
plt.show()
```

Custom Convolution Result

Q-6

```
image4 = cv2.imread(r"C:\Users\devan\OneDrive\Desktop\i1.jpeg",
cv2.IMREAD_GRAYSCALE)

plt.imshow(image4, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.show()
```

Original Image

1. Roberts Edge Detection

Manual Implementation (2D Filter)

```
# Define Roberts kernels
roberts_kernel_x = np.array([[1, 0], [0, -1]], dtype=np.float32)
roberts_kernel_y = np.array([[0, 1], [-1, 0]], dtype=np.float32)

# Apply kernels using filter2D
roberts_x = cv2.filter2D(image4, -1, roberts_kernel_x)
roberts_y = cv2.filter2D(image4, -1, roberts_kernel_y)
roberts_edge = cv2.addWeighted(roberts_x, 0.5, roberts_y, 0.5, 0)
```

```
# Display results
plt.imshow(roberts_edge, cmap='gray')
plt.title('Roberts Edge Detection')
plt.axis('off')
plt.show()
```

Sobel Edge Detection

OpenCV Implementation

```
# Apply Sobel using OpenCV functions
sobel_x = cv2.Sobel(image4, cv2.CV_64F, 1, 0, ksize=3)  # x direction
sobel_y = cv2.Sobel(image4, cv2.CV_64F, 0, 1, ksize=3)  # y direction
sobel_edge = cv2.magnitude(sobel_x, sobel_y)

# Display results
plt.imshow(sobel_edge, cmap='gray')
plt.title('Sobel Edge Detection')
plt.axis('off')
plt.show()
```

Sobel Edge Detection

Manual Implementation (2D Filter)

```
# Define Sobel kernels
sobel_kernel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]],
dtype=np.float32)
sobel_kernel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]],
dtype=np.float32)

# Apply kernels using filter2D
sobel_x = cv2.filter2D(image4, -1, sobel_kernel_x)
sobel_y = cv2.filter2D(image4, -1, sobel_kernel_y)
sobel_edge_manual = cv2.addWeighted(sobel_x, 0.5, sobel_y, 0.5, 0)

# Display results
plt.imshow(sobel_edge_manual, cmap='gray')
plt.title('Sobel Edge Detection (Manual)')
plt.axis('off')
plt.show()
```

Prewitt Edge Detection

Manual Implementation (2D Filter)

```
# Define Prewitt kernels
prewitt_kernel_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]],
dtype=np.float32)
prewitt_kernel_y = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]],
dtype=np.float32)

# Apply kernels using filter2D
prewitt_x = cv2.filter2D(image4, -1, prewitt_kernel_x)
prewitt_y = cv2.filter2D(image4, -1, prewitt_kernel_y)
prewitt_edge = cv2.addWeighted(prewitt_x, 0.5, prewitt_y, 0.5, 0)

# Display results
plt.imshow(prewitt_edge, cmap='gray')
plt.title('Prewitt Edge Detection')
plt.axis('off')
plt.show()
```

Canny Edge Detection

OpenCV Implementation

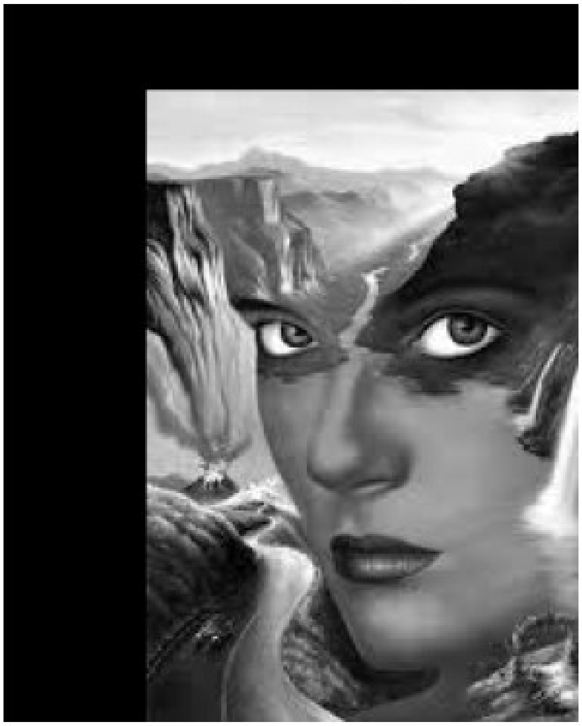
```
# Apply Canny edge detection
canny_edge = cv2.Canny(image4, 100, 200)

# Display results
plt.imshow(canny_edge, cmap='gray')
plt.title('Canny Edge Detection')
plt.axis('off')
plt.show()
```

Q-7

Translation

```
def translate_image(image, tx, ty):
    rows, cols = image.shape
    translation_matrix = np.array([[1, 0, tx],
                                   [0, 1, ty],
                                   [0, 0, 1]], dtype=np.float32)
    translated_image = cv2.warpPerspective(image, translation_matrix,
(cols, rows))
    return translated_image

# Apply translation
translated_image = translate_image(image4, tx=50, ty=30)
plt.imshow(translated_image, cmap='gray')
plt.title('Translated Image')
plt.axis('off')
plt.show()
```

Translated Image



Scaling

```
def scale_image(image, sx, sy):
    rows, cols = image.shape
    scaling_matrix = np.array([[sx, 0, 0],
                               [0, sy, 0],
                               [0, 0, 1]], dtype=np.float32)
    scaled_image = cv2.warpPerspective(image, scaling_matrix,
(int(cols * sx), int(rows * sy)))
    return scaled_image

# Apply scaling
scaled_image = scale_image(image4, sx=1.5, sy=1.5)
plt.imshow(scaled_image, cmap='gray')
plt.title('Scaled Image')
plt.axis('off')
plt.show()
```

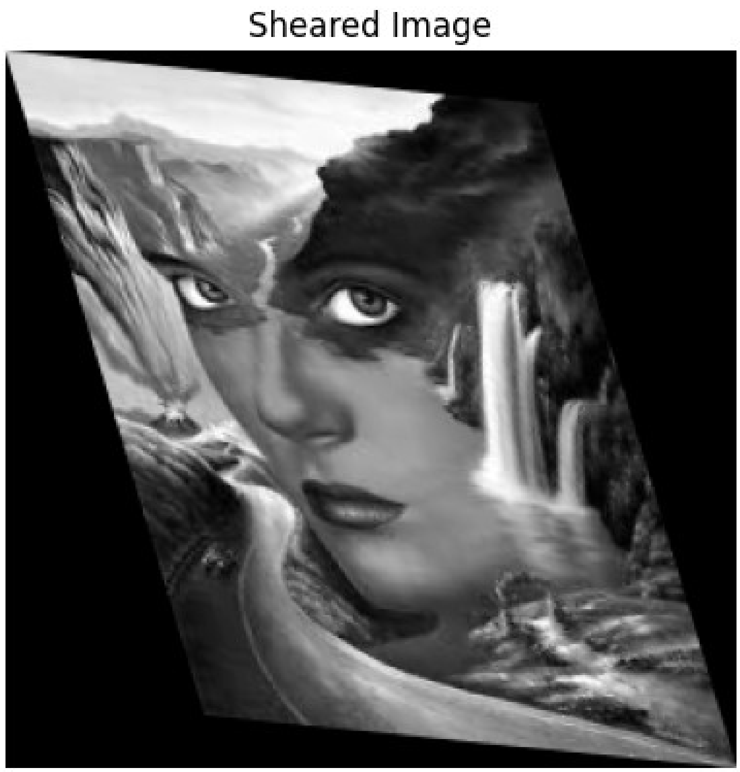
Rotation

```
def rotate_image(image, angle):
    rows, cols = image.shape
    angle_rad = np.deg2rad(angle)
    rotation_matrix = np.array([[np.cos(angle_rad), -
np.sin(angle_rad), 0],
                                [np.sin(angle_rad), np.cos(angle_rad),
0],
                                [0, 0, 1]], dtype=np.float32)
    rotated_image = cv2.warpPerspective(image, rotation_matrix, (cols,
rows))
    return rotated_image

# Apply rotation
rotated_image = rotate_image(image4, angle=30)
plt.imshow(rotated_image, cmap='gray')
plt.title('Rotated Image')
plt.axis('off')
plt.show()
```

Rotated Image

Shearing

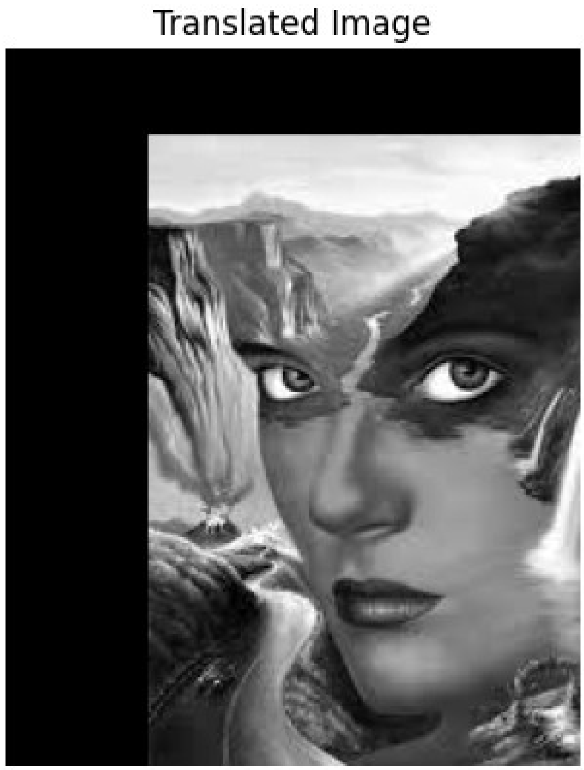
```
def shear_image(image, shx, shy):
    rows, cols = image.shape
    shearing_matrix = np.array([[1, shx, 0],
                                [shy, 1, 0],
                                [0, 0, 1]], dtype=np.float32)
    sheared_image = cv2.warpPerspective(image, shearing_matrix, (cols
+ int(abs(shx * rows)), rows + int(abs(shy * cols))))
    return sheared_image

# Apply shearing
sheared_image = shear_image(image4, shx=0.3, shy=0.1)
plt.imshow(sheared_image, cmap='gray')
plt.title('Sheared Image')
plt.axis('off')
plt.show()
```



Q-8

Translation

```
def translate_image(image, tx, ty):
    rows, cols = image.shape
    translation_matrix = np.float32([[1, 0, tx], [0, 1, ty]])
    translated_image = cv2.warpAffine(image, translation_matrix,
(cols, rows))
    return translated_image

# Apply translation
translated_image = translate_image(image4, tx=50, ty=30)
plt.imshow(translated_image, cmap='gray')
plt.title('Translated Image')
plt.axis('off')
plt.show()
```



Scaling

```
def scale_image(image, sx, sy):
    scaled_image = cv2.resize(image, None, fx=sx, fy=sy, 
interpolation=cv2.INTER_LINEAR)
    return scaled_image

# Apply scaling
scaled_image = scale_image(image4, sx=1.5, sy=1.5)
plt.imshow(scaled_image, cmap='gray')
plt.title('Scaled Image')
plt.axis('off')
plt.show()
```

Scaled Image

Rotation

```
def rotate_image(image, angle):
    rows, cols = image.shape
    center = (cols / 2, rows / 2)
    rotation_matrix = cv2.getRotationMatrix2D(center, angle, 1)
    rotated_image = cv2.warpAffine(image, rotation_matrix, (cols,
rows))
    return rotated_image

# Apply rotation
rotated_image = rotate_image(image4, angle=45)
plt.imshow(rotated_image, cmap='gray')
plt.title('Rotated Image')
plt.axis('off')
plt.show()
```

Rotated Image

Shearing

```
def shear_image(image, shx, shy):
    rows, cols = image.shape
    # Define the shearing matrix
    shearing_matrix = np.float32([[1, shx, 0], [shy, 1, 0]])
    sheared_image = cv2.warpAffine(image, shearing_matrix, (cols + 
int(abs(shx * rows)), rows + int(abs(shy * cols))))
    return sheared_image

# Apply shearing
sheared_image = shear_image(image4, shx=0.3, shy=0.1)
plt.imshow(sheared_image, cmap='gray')
plt.title('Sheared Image')
plt.axis('off')
plt.show()
```

Sheared Image

Q-9

Simple Thresholding

```
def simple_threshold(image, threshold_value=127, max_value=255):
    _, thresholded_image = cv2.threshold(image, threshold_value, 
max_value, cv2.THRESH_BINARY)
    return thresholded_image

# Apply simple thresholding
simple_thresh_image = simple_threshold(image4, threshold_value=127)
plt.imshow(simple_thresh_image, cmap='gray')
plt.title('Simple Thresholding')
plt.axis('off')
plt.show()
```

Simple Thresholding

Otsu’s Binarization

```
def otsu_threshold(image):
    _, otsu_thresh_image = cv2.threshold(image, 0, 255,
cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    return otsu_thresh_image

# Apply Otsu's thresholding
otsu_thresh_image = otsu_threshold(image4)
plt.imshow(otsu_thresh_image, cmap='gray')
plt.title("Otsu's Binarization")
plt.axis('off')
plt.show()
```

Otsu's Binarization

Adaptive Thresholding

```
def adaptive_threshold(image, method='mean', block_size=11, C=2):
    if method == 'mean':
        adaptive_thresh_image = cv2.adaptiveThreshold(image, 255,
cv2.ADAPTIVE_THRESH_MEAN_C,

cv2.THRESH_BINARY, block_size, C)
    elif method == 'gaussian':
        adaptive_thresh_image = cv2.adaptiveThreshold(image, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C,

cv2.THRESH_BINARY, block_size, C)
    return adaptive_thresh_image

# Apply adaptive mean thresholding
adaptive_mean_image = adaptive_threshold(image4, method='mean')
plt.imshow(adaptive_mean_image, cmap='gray')
plt.title('Adaptive Mean Thresholding')
plt.axis('off')
plt.show()
```

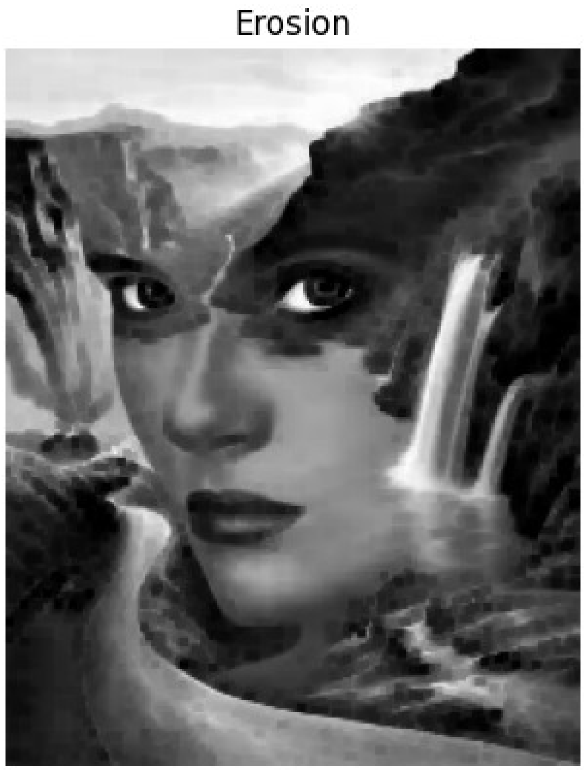
Adaptive Mean Thresholding

```
# Apply adaptive Gaussian thresholding
adaptive_gaussian_image = adaptive_threshold(image4,
method='gaussian')
plt.imshow(adaptive_gaussian_image, cmap='gray')
plt.title('Adaptive Gaussian Thresholding')
plt.axis('off')
plt.show()
```

Q-10

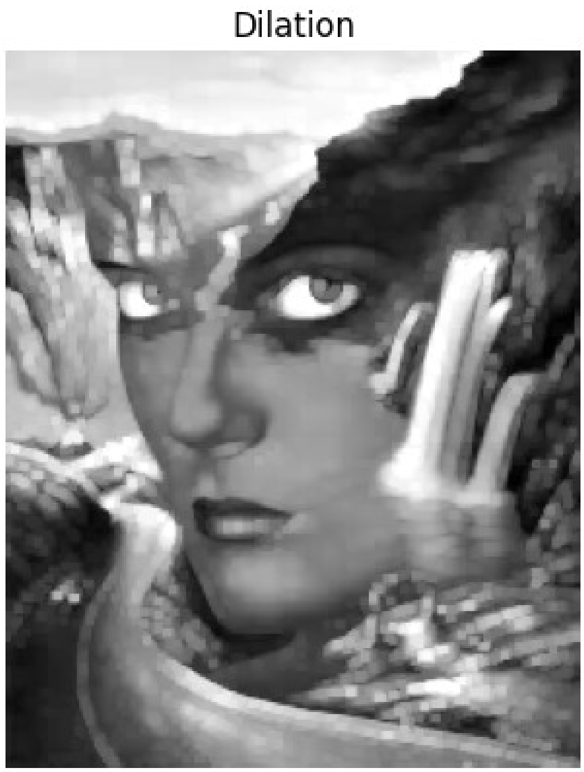
Erosion

```
def erosion(image, kernel_size=3):
    kernel = np.ones((kernel_size, kernel_size), np.uint8)  # Define kernel size
    eroded_image = cv2.erode(image, kernel, iterations=1)  # Perform erosion
    return eroded_image

# Apply erosion
eroded_image = erosion(image4)
plt.imshow(eroded_image, cmap='gray')
plt.title('Erosion')
plt.axis('off')
plt.show()
```



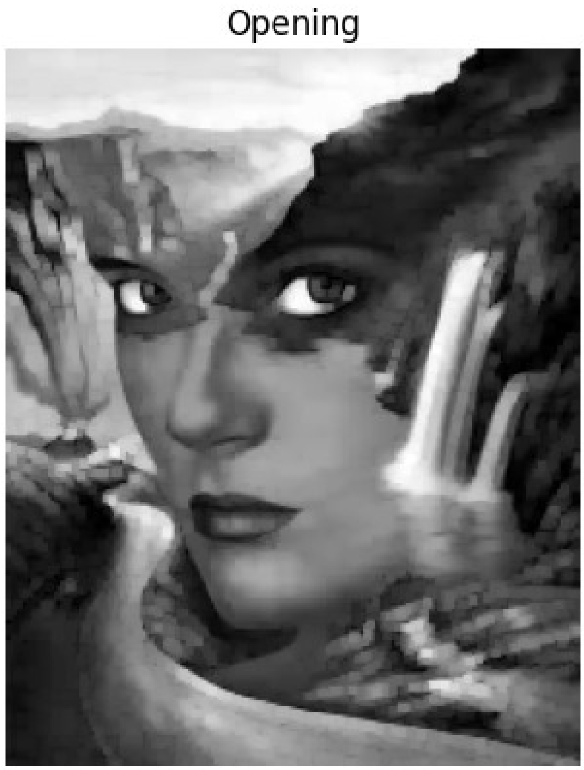
Dilation

```
def dilation(image, kernel_size=3):
    kernel = np.ones((kernel_size, kernel_size), np.uint8)  # Define kernel size
    dilated_image = cv2.dilate(image, kernel, iterations=1)  # Perform dilation
    return dilated_image

# Apply dilation
dilated_image = dilation(image4)
plt.imshow(dilated_image, cmap='gray')
plt.title('Dilation')
plt.axis('off')
plt.show()
```



Opening

```
def opening(image, kernel_size=3):
    kernel = np.ones((kernel_size, kernel_size), np.uint8)  # Define 
kernel size
    opened_image = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)  # 
Perform opening
    return opened_image

# Apply opening
opened_image = opening(image4)
plt.imshow(opened_image, cmap='gray')
plt.title('Opening')
plt.axis('off')
plt.show()
```



Closing

```
def closing(image, kernel_size=3):
    kernel = np.ones((kernel_size, kernel_size), np.uint8)  # Define kernel size
    closed_image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)  # Perform closing
    return closed_image

# Apply closing
closed_image = closing(image4)
plt.imshow(closed_image, cmap='gray')
plt.title('Closing')
plt.axis('off')
plt.show()
```

Q-11

```
from sklearn.cluster import KMeans

def compress_image_kmeans(image, k=8):
    # Reshape the image to a 2D array (pixels, 3 for RGB values)
    image_reshaped = image.reshape((-1, 3))

    # Apply KMeans clustering
    kmeans = KMeans(n_clusters=k, random_state=0)
    kmeans.fit(image_reshaped)

    # Get the cluster centers (dominant colors)
    centers = kmeans.cluster_centers_.astype(int)

    # Assign each pixel to the nearest centroid
    labels = kmeans.labels_

    # Reconstruct the image by replacing each pixel's value with its
corresponding centroid
    compressed_image = centers[labels].reshape(image.shape)

    return compressed_image, centers, labels
```

```python
# Compress the image
k = 8 # Number of clusters (reduce to 8 colors)
compressed_image, centers, labels = compress_image_kmeans(image_rgb,
k)

# Display the compressed image
plt.imshow(compressed_image)
plt.title(f'Compressed Image with K={k}')
plt.axis('off')
plt.show()
```

Compressed Image with K=8

Display the original image vs compressed image

```python
plt.figure(figsize=(12, 6))

# Original Image
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title('Original Image')
plt.axis('off')

# Compressed Image
plt.subplot(1, 2, 2)
```

```python
plt.imshow(compressed_image)
plt.title(f'Compressed Image with K={k}')
plt.axis('off')

plt.show()
```

Original Image

Compressed Image with K=8

Q-12

```python
def watershed_segmentation(image):
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply GaussianBlur to smooth the image
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)

    # Apply edge detection (Canny or Sobel could also be used)
    edges = cv2.Canny(blurred, 100, 200)

    # Apply dilation to the edges
    kernel = np.ones((3, 3), np.uint8)
    dilated_edges = cv2.dilate(edges, kernel, iterations=3)

    # Apply distance transform to compute the foreground markers
    dist_transform = cv2.distanceTransform(dilated_edges, cv2.DIST_L2,
5)

    # Normalize the distance image
```

```
    _, fg_markers = cv2.threshold(dist_transform, 0.7 *
dist_transform.max(), 255, 0)

    # Convert the background to markers (0 for background)
    bg_markers = cv2.subtract(np.ones_like(fg_markers, dtype=np.uint8)
* 255, fg_markers.astype(np.uint8))

    # Combine background and foreground markers
    markers = np.int32(fg_markers) + np.int32(bg_markers)
    markers = markers * 1  # Convert markers to 8-bit

    # Apply watershed algorithm
    cv2.watershed(image, markers)

    # Mark the boundaries with -1 (from watershed)
    image[markers == -1] = [255, 0, 0]

    return image, markers

# Apply Watershed algorithm
segmented_image, markers = watershed_segmentation(image_rgb)

# Display the segmented image
plt.imshow(segmented_image)
plt.title('Watershed Segmentation Result')
plt.axis('off')
plt.show()
```

Watershed Segmentation Result

Q-13

WAP for image classification using traditional ML algorithms.

```
import numpy as np
import cv2
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from skimage.feature import hog
from sklearn.preprocessing import StandardScaler

# Load the MNIST dataset
digits = datasets.load_digits()
X, y = digits.images, digits.target

# Reshape images and extract HOG features
hog_features = []
for image in X:
    # Resize image to a fixed size if necessary (e.g., 28x28)
    image_resized = cv2.resize(image, (8, 8))

    # Extract HOG features
    features = hog(
        image_resized,
        orientations=9,
        pixels_per_cell=(4, 4),
        cells_per_block=(2, 2),
        block_norm='L2-Hys'
    )
    hog_features.append(features)

# Convert HOG features and labels to numpy arrays
hog_features = np.array(hog_features)
y = np.array(y)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(hog_features, y,
test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train an SVM classifier
clf = SVC(kernel='linear', random_state=42)
```

```
clf.fit(X_train, y_train)

# Predict on the test set
y_pred = clf.predict(X_test)

# Evaluate the classifier
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test,
y_pred))
```

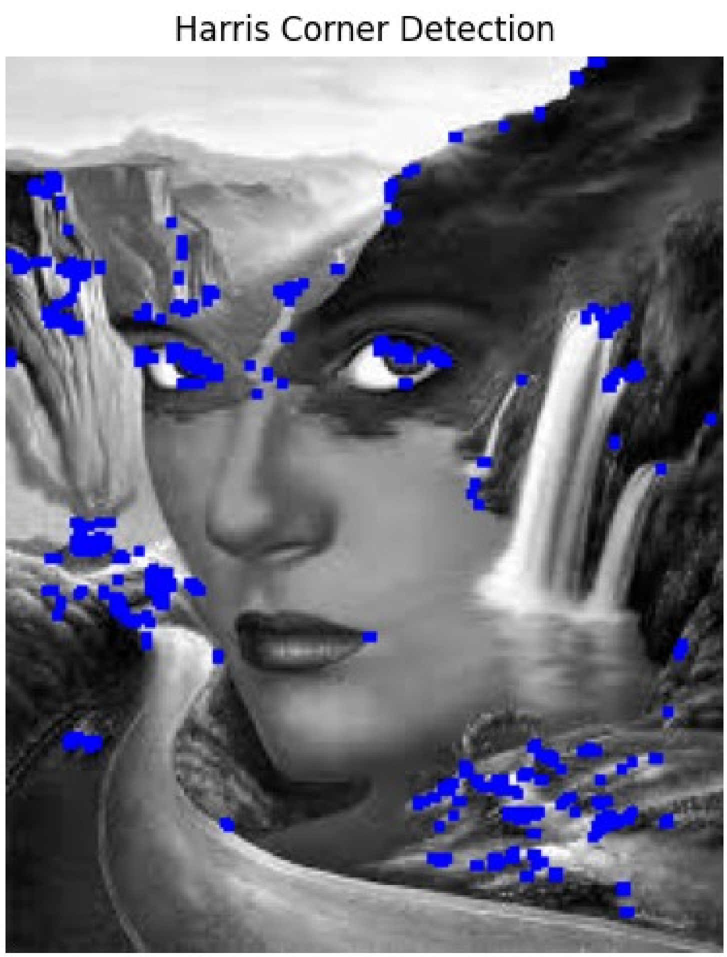
```
Accuracy: 0.8666666666666667
Classification Report:
               precision    recall  f1-score   support

           0       0.86      0.94      0.90        33
           1       0.93      0.93      0.93        28
           2       0.88      0.91      0.90        33
           3       0.80      0.82      0.81        34
           4       0.87      0.89      0.88        46
           5       0.87      0.87      0.87        47
           6       1.00      0.91      0.96        35
           7       1.00      0.97      0.99        34
           8       0.70      0.70      0.70        30
           9       0.76      0.72      0.74        40

    accuracy                           0.87       360
   macro avg       0.87      0.87      0.87       360
weighted avg       0.87      0.87      0.87       360
```

Q-14

WAP to extract Haris corner detection feature.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load the image in grayscale
image = cv2.imread('/i1.jpeg', cv2.IMREAD_GRAYSCALE)

# Convert to float32 for better precision in Harris corner detection
gray = np.float32(image)

# Apply the Harris Corner Detection
dst = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)

# Dilate the result to enhance corner points
```

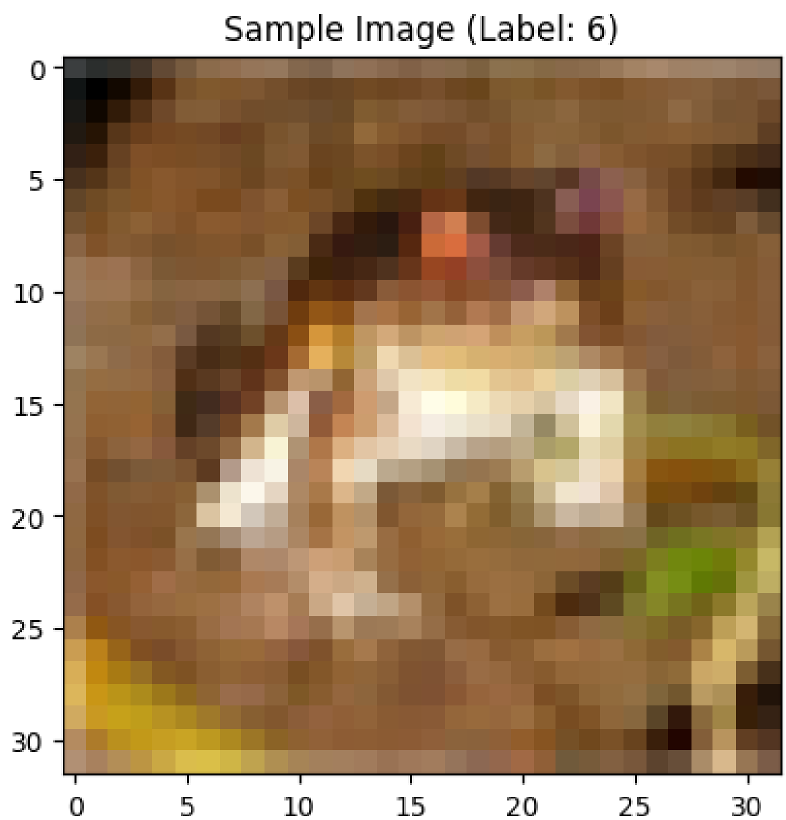
```python
dst = cv2.dilate(dst, None)

# Threshold to mark the corners on the original image
threshold = 0.01 * dst.max()
image_with_corners = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
image_with_corners[dst > threshold] = [0, 0, 255]  # Marking corners in red

# Display the result
plt.figure(figsize=(10, 6))
plt.imshow(image_with_corners)
plt.title("Harris Corner Detection")
plt.axis('off')
plt.show()
```



Q-15

WAP to extract HOG features.

```
import numpy as np
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split
from skimage.feature import hog
from skimage import color
from tensorflow.keras.datasets import cifar10
import matplotlib.pyplot as plt

# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Flatten y_train and y_test to make them 1D arrays
y_train = y_train.flatten()
y_test = y_test.flatten()

# Display a sample image
plt.imshow(X_train[0])
plt.title(f'Sample Image (Label: {y_train[0]})')
plt.show()
```




```
# Function to extract HOG features from the dataset
def extract_hog_features(images):
    hog_features = []
```

```python
    for img in images:
        # Convert the image to grayscale
        gray_img = color.rgb2gray(img)

        # Extract HOG features
        features = hog(gray_img,
                       pixels_per_cell=(8, 8),
                       cells_per_block=(2, 2),
                       block_norm='L2-Hys',
                       visualize=False)
        hog_features.append(features)

    return np.array(hog_features)

# Extract HOG features for train and test sets
X_train_hog = extract_hog_features(X_train)
X_test_hog = extract_hog_features(X_test)

print("Feature extraction complete.")
print(f"Feature vector shape for a single image:
{X_train_hog[0].shape}")
```

```
Feature extraction complete.
Feature vector shape for a single image: (324,)
```

```python
# Split the training data further into training and validation sets
X_train_split, X_val_split, y_train_split, y_val_split =
train_test_split(X_train_hog, y_train, test_size=0.2, random_state=42)

# Train an SVM classifier
svm_clf = SVC(kernel='linear', C=1.0, random_state=42)
svm_clf.fit(X_train_split, y_train_split)

# Validate the model
y_val_pred = svm_clf.predict(X_val_split)
val_accuracy = accuracy_score(y_val_split, y_val_pred)
print(f"Validation Accuracy: {val_accuracy:.4f}")
```

```
Validation Accuracy: 0.5243
```

```python
# Predict on the test set
y_test_pred = svm_clf.predict(X_test_hog)

# Evaluate the model
test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")

# Classification report
print("\nClassification Report:\n")
print(classification_report(y_test, y_test_pred, target_names=[
    'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
```

```
'horse', 'ship', 'truck'
]))
```

```
Test Accuracy: 0.5266

Classification Report:

              precision    recall  f1-score   support

    airplane       0.57      0.61      0.59      1000
  automobile       0.58      0.64      0.61      1000
        bird       0.45      0.41      0.43      1000
         cat       0.40      0.32      0.35      1000
        deer       0.43      0.49      0.46      1000
         dog       0.46      0.42      0.44      1000
        frog       0.53      0.62      0.57      1000
       horse       0.58      0.56      0.57      1000
        ship       0.58      0.57      0.58      1000
       truck       0.66      0.63      0.64      1000

    accuracy                           0.53     10000
   macro avg       0.52      0.53      0.52     10000
weighted avg       0.52      0.53      0.52     10000
```

Q-16

WAP to extract SIFT features.

```
import cv2
import matplotlib.pyplot as plt

# Load the image in grayscale
image = cv2.imread('/i1.jpeg', cv2.IMREAD_GRAYSCALE)

# Initialize the SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints and descriptors
keypoints, descriptors = sift.detectAndCompute(image, None)

# Draw keypoints on the image
output_image = cv2.drawKeypoints(image, keypoints, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Display the output
plt.figure(figsize=(10, 6))
plt.imshow(output_image, cmap='gray')
```

```
plt.title("SIFT Keypoints")
plt.axis('off')
plt.show()

# Optional: Print number of keypoints and descriptor shape
print("Number of keypoints detected:", len(keypoints))
print("Descriptor shape:", descriptors.shape)
```

SIFT Keypoints

```
Number of keypoints detected: 400
Descriptor shape: (400, 128)
```