Start coding or generate with AI.

Image segmentaions

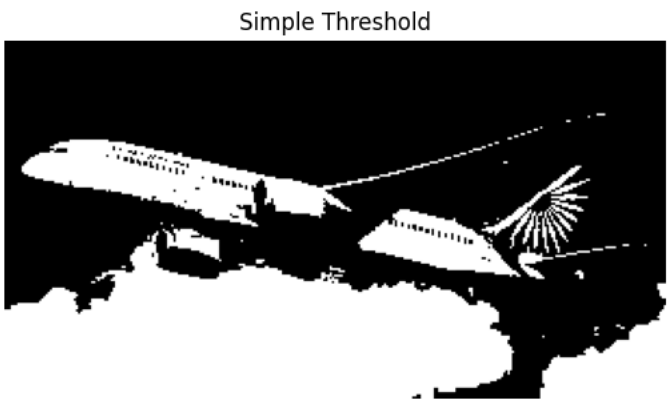
```
# Install OpenCV
!pip install opencv-python-headless

# Import Libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Helper to display images
def show(img, title="Image", cmap_type='gray'):
    plt.imshow(img, cmap=cmap_type)
    plt.title(title)
    plt.axis('off')
    plt.show()
```

```
Requirement already satisfied: opencv-python-headless in /usr/local/lib/python3.11/dist-packages (4.11.0.86)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.11/dist-packages (from opencv-python-headless) (2.0.2)
```

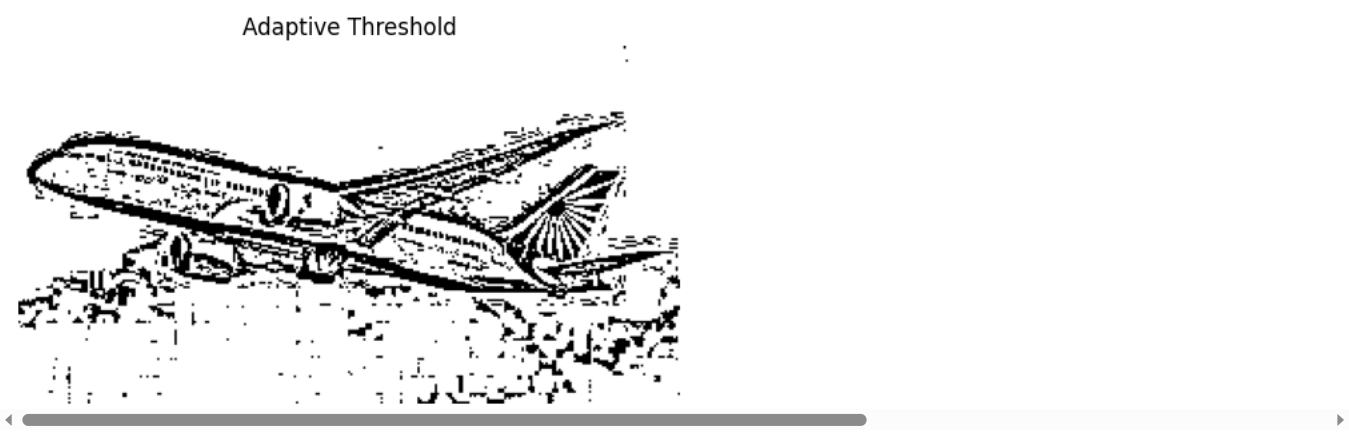
```
from google.colab import files
uploaded = files.upload()

# Read image
import io
from PIL import Image

image = cv2.imdecode(np.frombuffer(uploaded[list(uploaded.keys())[0]], np.uint8), cv2.IMREAD_COLOR)
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
show(image_rgb, "Original", cmap_type=None)
```

```
Choose Files flight.jfif
• flight.jfif(image/jpeg) - 8310 bytes, last modified: 5/15/2025 - 100% done
Saving flight.jfif to flight (1).jfif
```


```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
show(thresh, "Simple Threshold")
```



```
adaptive_thresh = cv2.adaptiveThreshold(gray, 255,
                                        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                        cv2.THRESH_BINARY, 11, 2)
show(adaptive_thresh, "Adaptive Threshold")
```




```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Define color range (e.g., red)
lower = np.array([0, 100, 100])
upper = np.array([100, 255, 255])

mask = cv2.inRange(hsv, lower, upper)
result = cv2.bitwise_and(image_rgb, image_rgb, mask=mask)
show(result, "Color Mask", cmap_type=None)
```

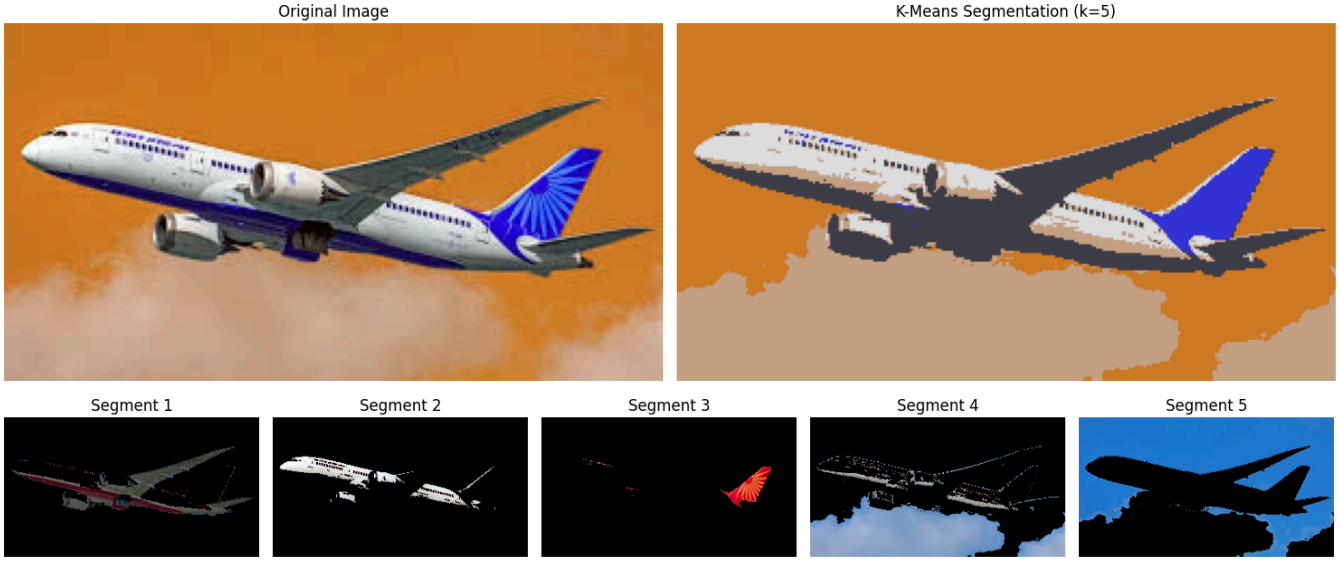

```
# Mask initialization
mask = np.zeros(image.shape[:2], np.uint8)
bgdModel = np.zeros((1,65), np.float64)
fgdModel = np.zeros((1,65), np.float64)

# Define rectangle (manually or via cv2.selectROI)
rect = (60, 60, image.shape[1]-100, image.shape[0]-100)

cv2.grabCut(image, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)
mask2 = np.where((mask==2)|(mask==0), 0, 1).astype('uint8')
grabcut_result = image_rgb * mask2[:, :, np.newaxis]
show(grabcut_result, "GrabCut", cmap_type=None)
```


```
# OpenCV Pixel-Based Segmentation Examples
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
from sklearn.cluster import KMeans, MeanShift, estimate_bandwidth
from sklearn.mixture import GaussianMixture

# Helper function to display images
def display_images(images, titles, figsize=(15, 5)):
    plt.figure(figsize=figsize)
    for i, (image, title) in enumerate(zip(images, titles)):
        plt.subplot(1, len(images), i+1)
        if len(image.shape) == 2:  # Grayscale
            plt.imshow(image, cmap='gray')
        else:  # Color
            plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        plt.title(title)
        plt.axis('off')
    plt.tight_layout()
    plt.show()



image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

print("Pure Pixel-Based Segmentation Techniques")

# 1. K-Means Pixel Clustering
print("\n1. K-Means Pixel Clustering")

# Reshape image to a 2D array of pixels for clustering
pixel_values = image_rgb.reshape((-1, 3))
pixel_values = np.float32(pixel_values)

# Define K-means parameters
k = 5  # Number of clusters
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
# OpenCV to stop the algorithm when either of the following conditions is met:
#Maximum iterations reached: 100 in this case.
#Centroids move less than epsilon (0.2)
# Perform K-means clustering
_, labels, centers = cv2.kmeans(pixel_values, k, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)

# Convert back to uint8 values
centers = np.uint8(centers)

# Map each pixel to its corresponding center
segmented_image = centers[labels.flatten()]

# Reshape back to the original image dimensions
kmeans_result = segmented_image.reshape(image_rgb.shape)

# Create individual segment masks
segment_masks = []
segment_images = []
for i in range(k):
    mask = np.zeros(labels.shape, dtype=np.uint8)
    mask[labels.flatten() == i] = 255
    mask = mask.reshape(gray.shape)
    segment_masks.append(mask)
```

```
    # Apply mask to original image
    segment_img = cv2.bitwise_and(image, image, mask=mask)
    segment_images.append(segment_img)

# Display the results
display_images([image_rgb, kmeans_result],
               ['Original Image', 'K-Means Segmentation (k=5)'])

# Display individual segments
display_images(segment_images[:5],
               [f'Segment {i+1}' for i in range(5)])
```

Pure Pixel-Based Segmentation Techniques

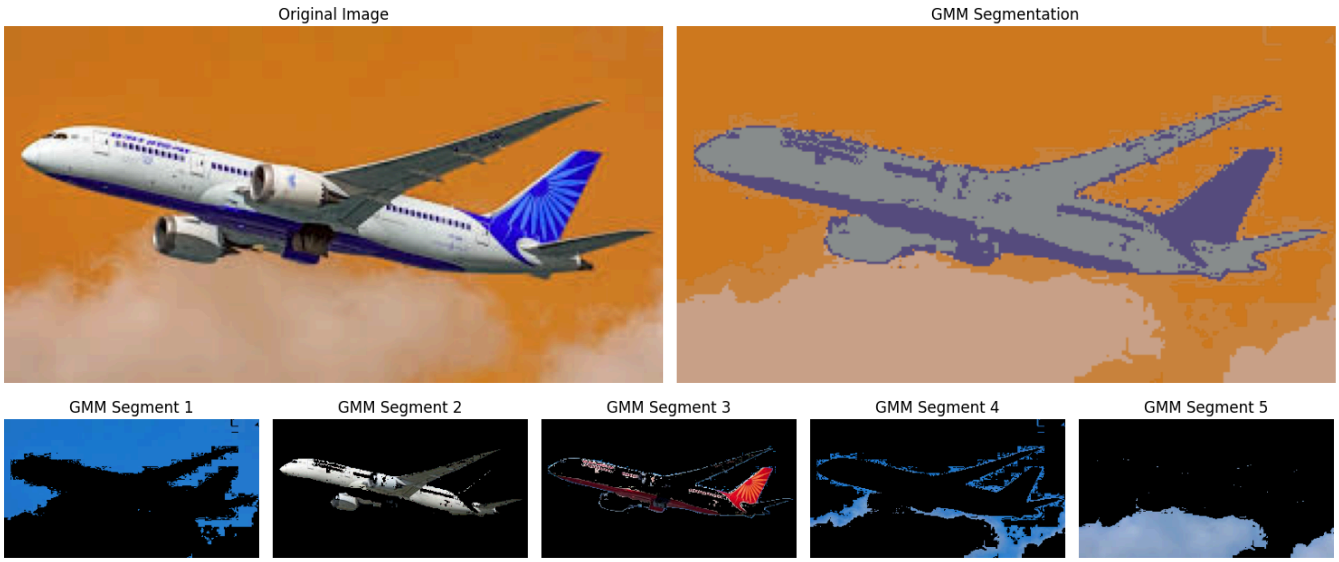
1. K-Means Pixel Clustering




```
# 2. Gaussian Mixture Model (GMM) Segmentation
print("\n2. Gaussian Mixture Model (GMM) Segmentation")

# Flatten the image
img_flat = image_rgb.reshape((-1, 3))

# Apply GMM
n_components = 5
gmm = GaussianMixture(n_components=n_components, covariance_type='full')
gmm.fit(img_flat)
gmm_labels = gmm.predict(img_flat)

# Create segmented image
gmm_result = np.zeros_like(img_flat)
for i in range(n_components):
    gmm_result[gmm_labels == i] = np.mean(img_flat[gmm_labels == i], axis=0)

# Reshape back to original image shape
gmm_result = gmm_result.reshape(image_rgb.shape).astype(np.uint8)

# Create individual segment masks
gmm_segments = []
for i in range(n_components):
    mask = np.zeros(gmm_labels.shape, dtype=np.uint8)
    mask[gmm_labels == i] = 255
    mask = mask.reshape(gray.shape)

    # Apply mask to original image
    segment_img = cv2.bitwise_and(image, image, mask=mask)
    gmm_segments.append(segment_img)

# Display the results
display_images([image_rgb, gmm_result],
               ['Original Image', 'GMM Segmentation'])

# Display individual segments
```

```
display_images(gmm_segments[:5],
               [f'GMM Segment {i+1}' for i in range(5)])
```

```
2. Gaussian Mixture Model (GMM) Segmentation
```



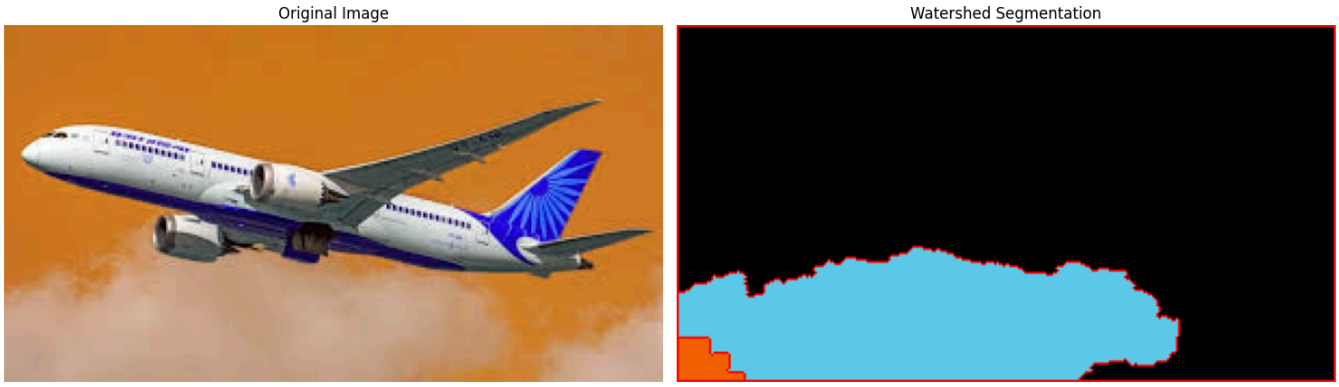

```
# 3. Mean Shift Pixel Clustering
print("\n3. Mean Shift Pixel Clustering")

# Resize for faster processing (Mean Shift is computationally intensive)
small_img = cv2.resize(image_rgb, (100, 100))
flat_small_img = small_img.reshape((-1, 3))

# Estimate bandwidth for mean shift
bandwidth = estimate_bandwidth(flat_small_img, quantile=0.1, n_samples=500)

# Apply Mean Shift clustering
ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(flat_small_img)
ms_labels = ms.labels_

# Get number of clusters
n_clusters = len(np.unique(ms_labels))
print(f"Number of estimated clusters: {n_clusters}")

# Create segmented image
ms_result = np.zeros_like(flat_small_img)
for i in range(n_clusters):
    ms_result[ms_labels == i] = np.mean(flat_small_img[ms_labels == i], axis=0)

# Reshape back to original image shape
ms_result = ms_result.reshape(small_img.shape).astype(np.uint8)

# Resize back to original size for display
ms_result_resized = cv2.resize(ms_result, (image_rgb.shape[1], image_rgb.shape[0]))

# Display the results
display_images([image_rgb, ms_result_resized],
               ['Original Image', f'Mean Shift Segmentation ({n_clusters} clusters)'])
```

```
3. Mean Shift Pixel Clustering
Number of estimated clusters: 16
```


```
# 5. Watershed Algorithm (Marker-based segmentation)
print("\n5. Watershed Algorithm (Marker-based)")

# Create markers for watershed
def create_markers(image):
    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Otsu's thresholding
    _, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

    # Noise removal with morphological operations
    kernel = np.ones((3, 3), np.uint8)
    opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=2)

    # Sure background area
    sure_bg = cv2.dilate(opening, kernel, iterations=3)

    # Finding sure foreground area
    dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
    _, sure_fg = cv2.threshold(dist_transform, 0.7*dist_transform.max(), 255, 0)
    sure_fg = np.uint8(sure_fg)

    # Finding unknown region
    unknown = cv2.subtract(sure_bg, sure_fg)

    # Marker labelling
    _, markers = cv2.connectedComponents(sure_fg)

    # Add 1 to all labels so that background is not 0, but 1
    markers = markers + 1

    # Mark the unknown region with 0
    markers[unknown == 255] = 0

    return markers

# Create markers for watershed
markers = create_markers(image)

# Apply watershed algorithm
watershed_markers = cv2.watershed(image, markers.copy())

# Create a colored image to visualize the watershed segments
height, width = gray.shape
watershed_colored = np.zeros((height, width, 3), dtype=np.uint8)

# Assign random colors to each segment
np.random.seed(42)  # for reproducibility
colors = np.random.randint(0, 255, size=(len(np.unique(watershed_markers)), 3), dtype=np.uint8)

# Fill segments with colors
for i in range(2, len(colors)):  # Skip background (0) and first segment (1)
    watershed_colored[watershed_markers == i] = colors[i]

# Mark watershed boundaries in red
watershed_colored[watershed_markers == -1] = [0, 0, 255]
```

```
# Display the results
display_images([image_rgb, watershed_colored],
               ['Original Image', 'Watershed Segmentation'])
```

```
5. Watershed Algorithm (Marker-based)
```



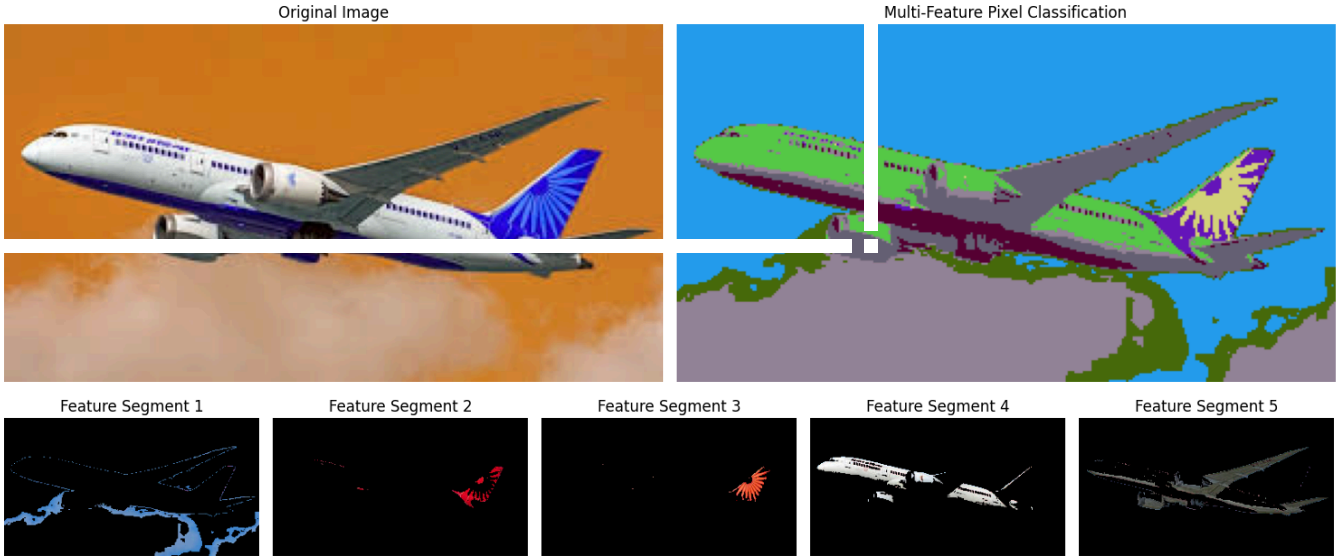

```
#6. SuperPixel Segmentation
print("\n6. SLIC SuperPixel Segmentation")

# Install scikit-image
!pip install -q scikit-image

from skimage.segmentation import slic, mark_boundaries

# Apply SLIC superpixel segmentation
segments = slic(image_rgb, n_segments=100, compactness=10, sigma=1)

# Create a visualization of the segments
segment_boundaries = mark_boundaries(image_rgb.copy() / 255.0, segments)
segment_boundaries = (segment_boundaries * 255).astype(np.uint8)

# Create a color-averaged version
superpixel_result = np.zeros_like(image_rgb)
for segment_id in np.unique(segments):
    mask = segments == segment_id
    superpixel_result[mask] = np.mean(image_rgb[mask], axis=0).astype(np.uint8)

# Display the results
display_images([image_rgb, segment_boundaries, superpixel_result],
               ['Original Image', 'SLIC Segment Boundaries', 'SLIC Segmented Image'])
```

```
6. SLIC SuperPixel Segmentation
```


```
# 7. Pixel Classification using Color Features
print("\n7. Pixel Classification using Color Features")

# Convert to different color spaces for better feature representation
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)

# Create feature vector for each pixel
features = np.zeros((image.shape[0] * image.shape[1], 9), dtype=np.float32)

# Flatten and combine color spaces
idx = 0
```

```
for i in range(image.shape[0]):
    for j in range(image.shape[1]):
        features[idx, 0:3] = image[i, j, :]       # BGR
        features[idx, 3:6] = hsv[i, j, :]         # HSV
        features[idx, 6:9] = lab[i, j, :]         # LAB
        idx += 1

# Perform K-means clustering on the feature vectors
k = 8  # Number of clusters
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
_, labels, centers = cv2.kmeans(features, k, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS

# Map back to original image shape
result_features = np.zeros((image.shape[0], image.shape[1]), dtype=np.uint8)
result_features = labels.reshape(image.shape[0], image.shape[1])

# Create a colored visualization
result_colored = np.zeros_like(image)
for i in range(k):
    result_colored[result_features == i] = np.random.randint(0, 255, 3)

# Create individual segments
feature_segments = []
for i in range(k):
    mask = np.zeros_like(gray)
    mask[result_features == i] = 255
    segment = cv2.bitwise_and(image, image, mask=mask)
    feature_segments.append(segment)

# Display the results
display_images([image_rgb, result_colored],
               ['Original Image', 'Multi-Feature Pixel Classification'])

# Display individual segments
display_images(feature_segments[:5],
               [f'Feature Segment {i+1}' for i in range(5)])
```

7. Pixel Classification using Color Features



Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.