Assignment 3

Abigail Stucki

Computer Science Undergraduate Student, Florida Polytechnic University

A report on computer vision focused on semantic segmentation methods for imagery.

Department of Computer Science
Florida Polytechnic University, Florida, USA

March 2024

ABSTRACT

The premise of this project was to apply knowledge of the mathematical and logical theory regarding image segmentation via code. Computer vision is built upon the computational capabilities of analysis and data extraction of images. Semantic segmentation is a vital component of said capabilities. Analyzing the pixels in terms of color saturation, density, and clumping can be used to separate and identify objects within the image.

Four images were fed into the program for a designated segmentation process. The two processes for implementation involve Otsu Binarization and Mean Shift segmentation methods. The combination of manual programming and pre-existing libraries allows for optimal results and demonstration of comprehension.

The following documentation will detail the application of these concepts in Python code for semantic image segmentation.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

This document entails the progression and demonstration of using the Python programming language to apply computer vision techniques focused on semantic image segmentation. Along with manual implementation, these functions accessed external libraries including the following: OpenCV-Python library (Bradski 2000), Python OS module, NumPy library (Harris et al. 2020), Sci-kit Image library (Van der Walt et al. 2014), and Matplotlib library (Hunter 2007).

The essential tasks to be executed in the program included:

1. Data importation and formatting.
2. Otsu Binarization method with two classes.
3. Otsu Binarization method with multiple classes.
4. Mean Shift segmentation method.

1.1 DEPENDENCIES

Implementation of the numerous segmentation methods as well as image reading, storage, and alteration requires multiple libraries.

```
import cv2
import os
import numpy as np
from matplotlib import pyplot as plt
from skimage.filters import threshold_multiotsu
```

Figure 1.1. This code imports the necessary libraries to execute the program. The key libraries include the OpenCV-Python library, the Python OS module, the NumPy library, the Sci-kit Image library, and the Matplotlib library.

1.2 IMAGE IMPORTATION AND FORMATTING

After dependencies are taken care of, the images may be read and stored in variables for ease of use and manipulation. Four sample images were provided with which to apply the varying specified segmentation methods. Python's OS module couples with OpenCV's library to store images in the necessary formatting for further alteration and analysis. The figure for visualization is also initiated at this point for the display of initial and resulting transformations.

```
# read input data
img1 = os.path.join(os.path.dirname(__file__), 'OTSU2class-edge_L-150x150.png')
otsu_img1 = cv2.cvtColor(cv2.imread(img1), cv2.COLOR_BGR2RGB)
otsu_img1_gray = cv2.imread(img1, cv2.IMREAD_GRAYSCALE)

img2 = os.path.join(os.path.dirname(__file__), 'OTSU2class-andreas_L-150x150.png')
otsu_img2 = cv2.cvtColor(cv2.imread(img2), cv2.COLOR_BGR2RGB)
otsu_img2_gray = cv2.imread(img2, cv2.IMREAD_GRAYSCALE)

img3 = os.path.join(os.path.dirname(__file__), 'OTSU_Multiple_Class-S01-150x150.png')
otsu_img3 = cv2.cvtColor(cv2.imread(img3), cv2.COLOR_BGR2RGB)
otsu_img3_gray = cv2.imread(img3, cv2.IMREAD_GRAYSCALE)

img4 = os.path.join(os.path.dirname(__file__), 'meanshift_S00-150x150.png')
mean_img = cv2.cvtColor(cv2.imread(img4), cv2.COLOR_BGR2RGB)
mean_img_gray = cv2.imread(img4, cv2.IMREAD_GRAYSCALE)
```

Figure 1.2. This code stores the four images as both an RGB image and a Grayscale image. The two differing color schema allow for the optimization of various segmentation methods.

```
# create figure and axes for displayal
fig1, axes1 = plt.subplots(4, 3)
fig1.tight_layout()
```

Figure 1.3. This code creates the figure and necessary axes for the visual output of all features of the program. The layout is also adjusted so all axis labels and titles are visible.

1.3 OTSU BINARIZATION (TWO CLASSES)

The next feature to be implemented is the Otsu Binarization segmentation method with a focus on two classes. OpenCV's library has multiple pre-existing functions and variables provided to support the coding of this segmentation using two classes. The pixel classes are obtained by converting the image to grayscale, thus the options are condensed to either black or white through the segmentation.

```
# otsu binarization with two pixel classes
_, thresh1 = cv2.threshold(otsu_img1_gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
hist1 = cv2.calcHist([otsu_img1_gray], [0], None, [256], [0, 256])
```

Figure 1.4. This code shows the implementation of OpenCV's pre-existing Otsu Binarization capabilities. The calculation of a histogram is also visible in this code.

```
# display first set of images and matching histogram
axes1[0][0].set_title('Original Image')
axes1[0][0].axis('off')
axes1[0][0].imshow(otsu_img1, cmap='gray')
axes1[0][1].set_title('Histogram')
axes1[0][1].plot(hist1)
axes1[0][2].set_title('Otsu\'s Binarization')
axes1[0][2].axis('off')
axes1[0][2].imshow(thresh1, cmap='gray')
```

Figure 1.5. This code shows the programming required to return the original image, histogram, and resulting image of the Otsu Binarization segmentation.

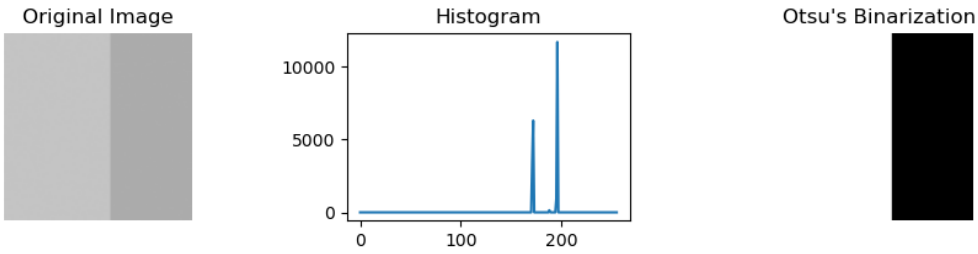

Figure 1.6. This image displays the resulting images and histogram from the prior code.

This process was also applied to a secondary, more complex image to further demonstrate the capabilities and segmentation. Where the first image was a seemingly clean split between two tones for simple segmentation, the second image is visually complex with more than two tones present.

```
# otsu binarization with two pixel classes
_, thresh2 = cv2.threshold(otsu_img2_gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
hist2 = cv2.calcHist([otsu_img2_gray], [0], None, [256], [0, 256])
```

Figure 1.7. This code shows the implementation of OpenCV's pre-existing Otsu Binarization capabilities. The calculation of a histogram is also visible in this code.

```
# display second set of images and matching histogram
axes1[1][0].set_title('Original Image')
axes1[1][0].axis('off')
axes1[1][0].imshow(otsu_img2, cmap='gray')
axes1[1][1].set_title('Histogram')
axes1[1][1].plot(hist2)
axes1[1][2].set_title('Otsu\'s Binarization')
axes1[1][2].axis('off')
axes1[1][2].imshow(thresh2, cmap='gray')
```

Figure 1.8. This code shows the programming required to return the original image, histogram, and resulting image of the Otsu Binarization segmentation.

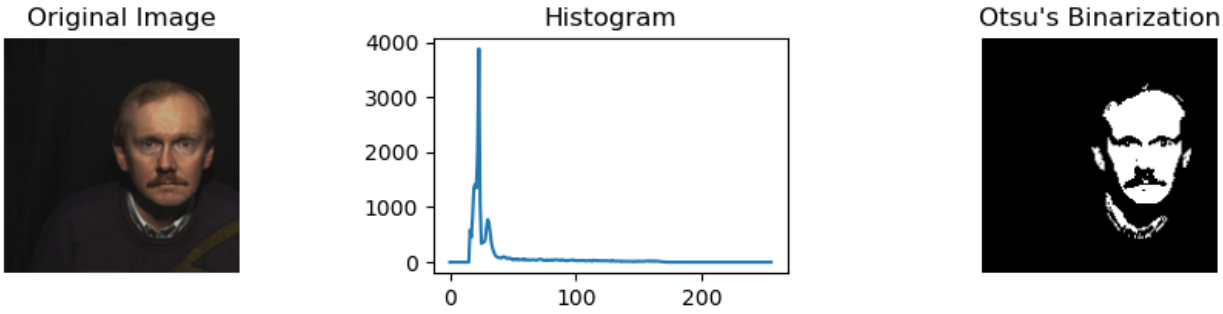

Figure 1.9. This image displays the resulting images and histogram from the prior code.

1.4 OTSU BINARIZATION (MULTI-CLASS)

The next feature to be implemented is the Otsu Binarization segmentation method with a focus on multiple classes. OpenCV's library has multiple pre-existing functions and variables provided to support the coding of this segmentation using two classes, however that alone is not enough to accomplish a multi-class approach. OpenCV's capabilities need to be combined with that of Numpy in order to produce the designated results. The pixel classes are obtained by converting the image to grayscale and pulling the threshold from said grayscale image. Numpy is then used to manipulate the image in the assigned thresholds.

```
# otsu binarization with multiple classes
thresh = threshold_multiotsu(otsu_img3_gray)
reg = np.digitize(otsu_img3_gray, bins=thresh)
```

Figure 1.10. This code shows the implementation of both OpenCV's Otsu Binarization capabilities and Numpy's digitization capabilities. The combination of efforts produces an image with more than two pixel classes.

```
# display third set of images and matching histogram
axes1[2][0].set_title('Original Image')
axes1[2][0].axis('off')
axes1[2][0].imshow(otsu_img3, cmap='gray')
axes1[2][1].set_title('Histogram')
axes1[2][1].hist(otsu_img3_gray.ravel(), 255)
for t in thresh:
    axes1[2][1].axvline(t, color='r')
axes1[2][2].set_title('Otsu\'s Binarization (Multi)')
axes1[2][2].axis('off')
axes1[2][2].imshow(reg, cmap='gray')
```

Figure 1.11. This code shows the programming required to return the original image, histogram, and resulting image of the Otsu Binarization segmentation. Included is the calculation of the histogram.

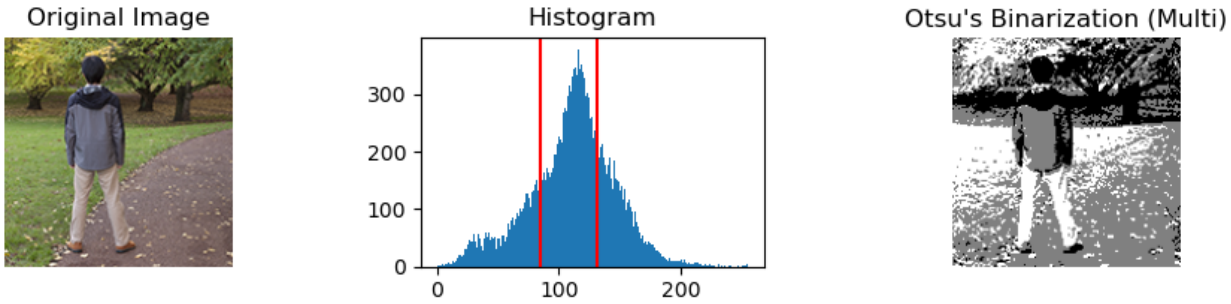

Figure 1.12. This image displays the resulting images and histogram from the prior code.

1.5 MEAN SHIFT SEGMENTATION

The Mean Shift segmentation method requires following a radius within the given bandwidth to group pixels together in classes. As none of the libraries had pre-existing functions to execute a mean shift, multiple varying methods were combined to get the desired result. The Numpy library was useful for segmenting the image based on the chosen radius and bandwidth. The shifted image was obtained from executing a series of moves to obtain the mean in clusters and transform the new image in vectors. The histogram clearly shows the separation between clusters.

```
# mean shift method
img_rows, img_cols = mean_img_gray.shape
mean_shift = np.zeros_like(mean_img_gray)
radius = 3
band = 1
for r in range(img_rows):
    for c in range(img_cols):
        cluster = mean_img_gray[max(0, r-radius):min(img_rows, r+1+radius),
                                max(0, c-radius):min(img_cols, c+1+radius)]
        vector = np.mean(cluster, axis=(0,1)) - mean_img_gray[r,c]
        mean_shift[r,c] = mean_img_gray[r,c] + vector * band
```

Figure 1.13. This code shows the implementation of the mathematical calculations necessary to execute Mean Shift segmentation on the image.

```
#display fourth set of images and matching histogram
axes1[3][0].set_title('Original Image')
axes1[3][0].axis('off')
axes1[3][0].imshow(mean_img)
axes1[3][1].set_title('Histogram')
axes1[3][1].hist(mean_shift.ravel(), 255)
axes1[3][2].set_title('Mean Shift')
axes1[3][2].axis('off')
axes1[3][2].imshow(mean_shift, cmap='gray')

plt.show()
```

Figure 1.14. This code shows the programming required to return the original image, histogram, and resulting image of the Mean Shift segmentation. Included is the calculation of the histogram.

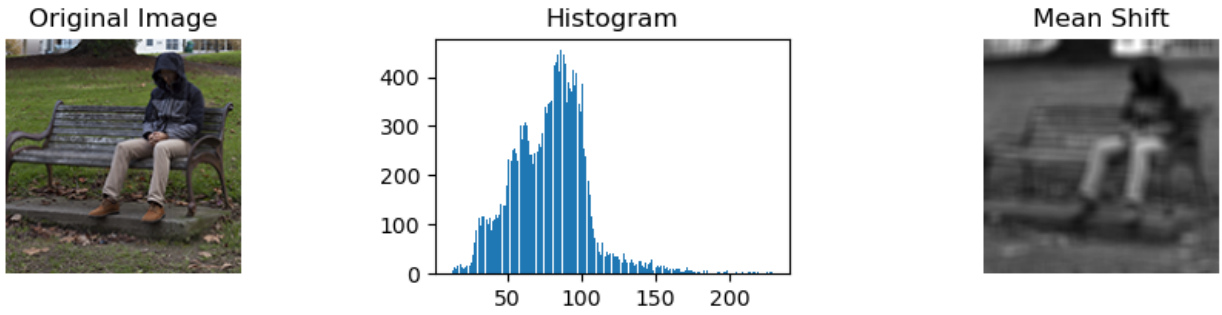

Figure 1.15. This image displays the resulting images and histogram from the prior code. Again, the clustering of pixels is visible in the outputted image as well as the histogram.

1.6 FINAL VISUALIZATION

When fully executed, the program will return the following imagery for user visualization and comprehension.

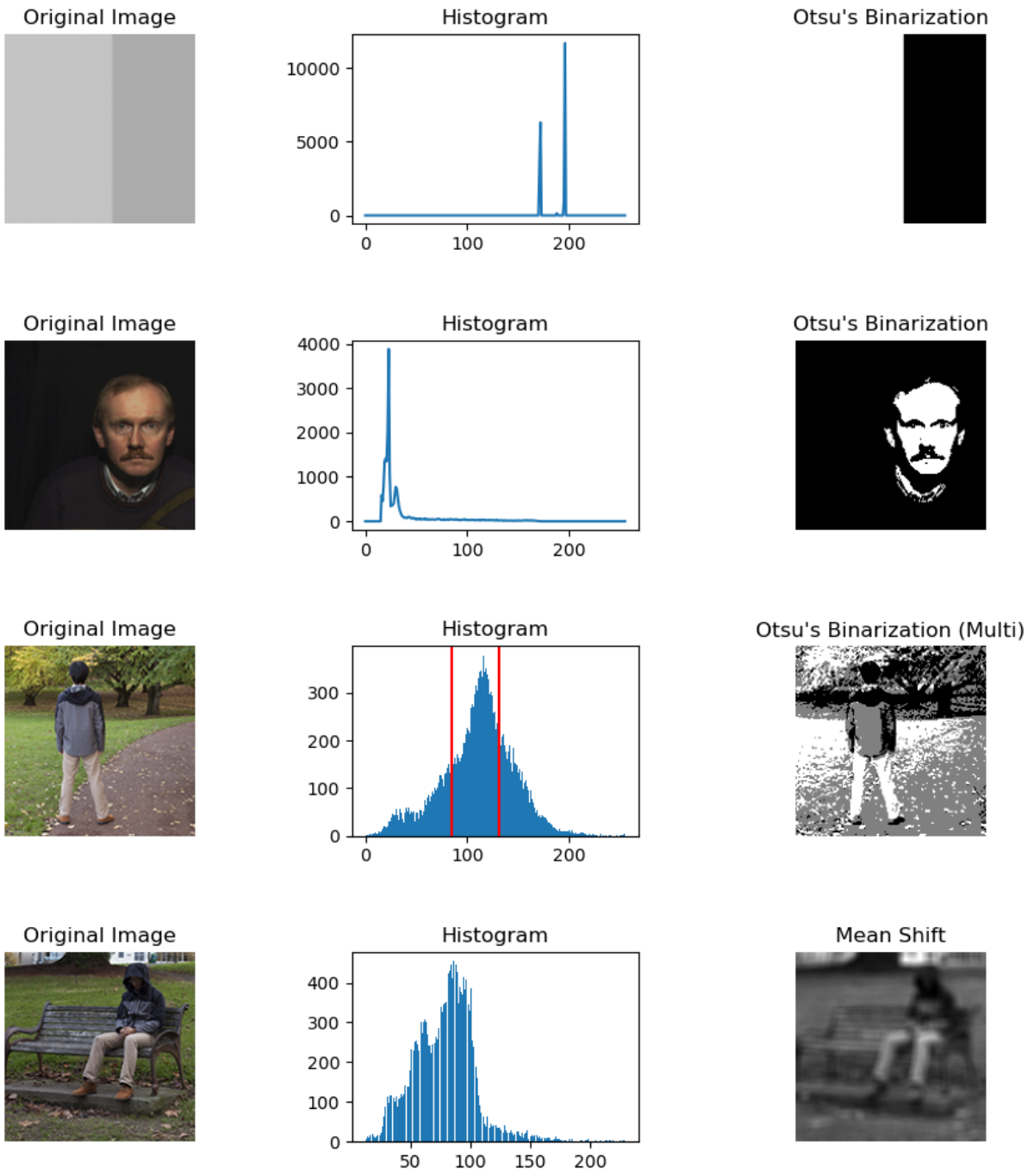

Figure 1.16. This image displays the resulting images and histograms from the prior code. Here, the differing segmentation methods are presented all together.

REFERENCES

Bradski, G. (2000). "The OpenCV Library." *Dr. Dobb's Journal of Software Tools*.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). "Array programming with NumPy." *Nature*, 585(7825), 357–362.

Hunter, J. D. (2007). "Matplotlib: A 2d graphics environment." *Computing in Science & Engineering*, 9(3), 90–95.

Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., and Yu, T. (2014). "scikit-image: image processing in python." *PeerJ*, 2, e453.