Introduction to Image Processing and Computer Vision
Laboratory Project 1: Leaves Segmentation

Jakub Bodak[1]

[1]Faculty of Mathematics and and Information Science, Warsaw University of Technology

January 5, 2023

1 Introduction

Image segmentation is a technique that divides a digital image into several subgroups known as Image segments, which serves to simplify future processing or analysis of the image by decreasing the complexity of the original image. In plain English, segmentation is the process of giving pixels labels. Each pixel or piece of a picture allocated to the same category has a unique label.

In this lab, we are using a particular directory from *PlantsVillage* dataset with pictures of leaves on a simple background. In my case the directory chosen is *GrapeBlackrot*. Our goal is to segment leaves from the background of images from folder *color* (Fig. 1) and compare the results with ground truth images obtained from *segmented* folder (Fig. 2). For the sake of showing exactly what is happening step by step I randomly selected 6 separate leaves.

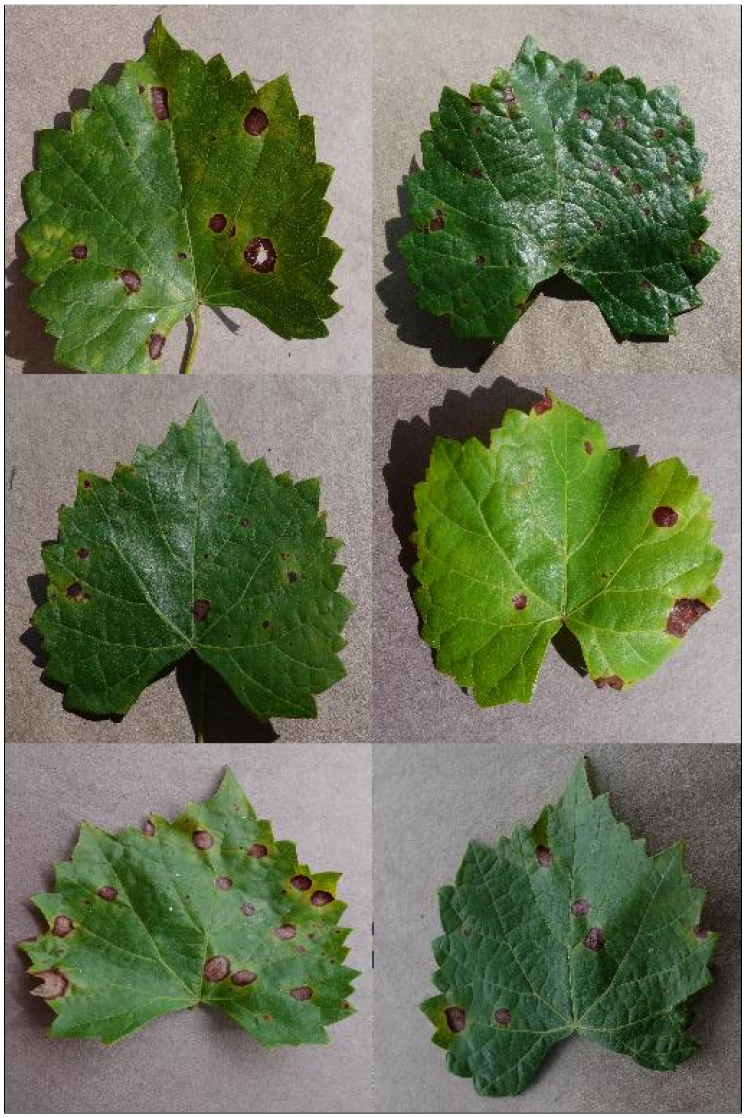

Figure 1: Images from *color* folder

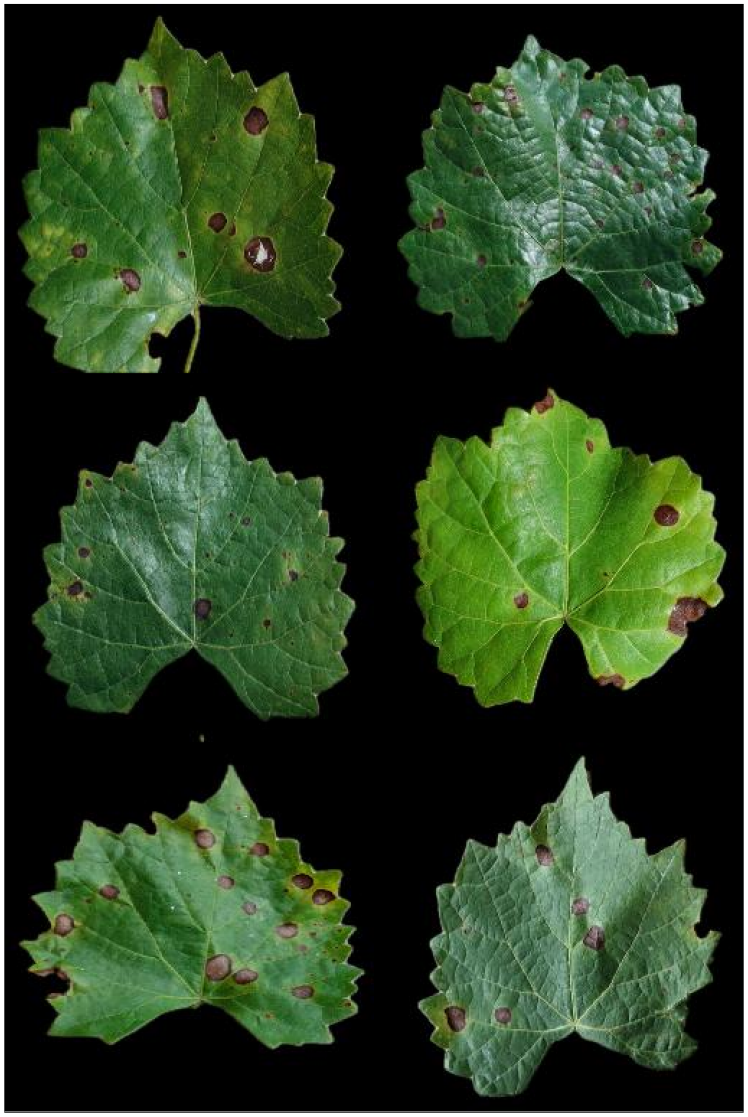

Figure 2: Images from *segmented* folder

2 Solution

The main algorithm can be separated into 2 stages: preparation and segmentation. Firstly I declared variables for directory paths as well as calculating accuracy and storing worst and best cases for results processing:

```
dir_parent = 'M:/cv/CVProj/'
dir_seg = 'segmented/'
dir_res = 'result/'
dir_col = 'color/'
dir_gt = 'ground truth/'

c = 0  # counter of files
d_sum = 0  # sum of Dice coefficient accuracies
j_sum = 0  # sum of Jaccardi index accuracies

# arrays for storing best and worst cases(accuracy, filename, image)
worst = [[1, '', 0], [1, '', 0], [1, '', 0], [1, '', 0], [1, '', 0]]
best = [[0, '', 0], [0, '', 0], [0, '', 0], [0, '', 0], [0, '', 0]]
```

Due to some of the images in *segmented* folder being faulty, I removed a number of them. In order not to encounter any errors because of this action, all algorithm operations are nested in a loop iterating through files in the *segmented* folder:

```
# iterating through filenames in the folder with segmented images
for filename in os.listdir(dir_seg):
    f = os.path.join(dir_seg, filename)
    if os.path.isfile(f):
```

2.1 Ground truth preparation

The first task is to prepare ground truth binary labels of images from *segmented* folder. These pictures all have a leaf on a black background. Due to this fact, we can easily make a binary mask of the leaf. The image is read and converted to grayscale (Fig. 3). Gaussian blurring is added to eliminate some of the noise (Fig. 4) and we threshold the image with range $20 - 255$ representing all levels of grayscale excluding the darkest ones (blacks). This operation gives us ground truth binary masks (Fig. 5):

```
# reading segmented image and creating a ground truth mask
# with simple blurring and thresholding
img_seg = cv2.imread(dir_seg + filename, 1)
img_seg_gray = cv2.cvtColor(img_seg, cv2.COLOR_BGR2GRAY)
img_seg_blur = cv2.GaussianBlur(img_seg_gray, (3, 3), 0)
ret, img_gt = cv2.threshold(img_seg_blur, 20, 255, cv2.THRESH_BINARY)
```

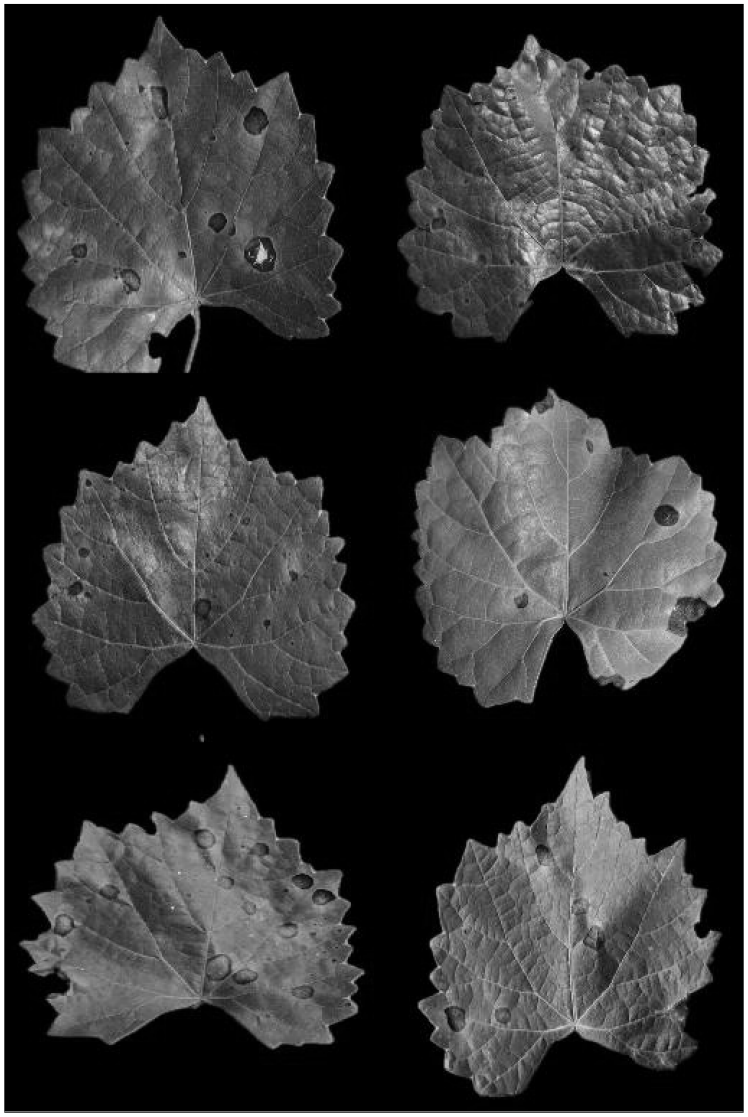

Figure 3: Images from *segmented* folder converted to grayscale

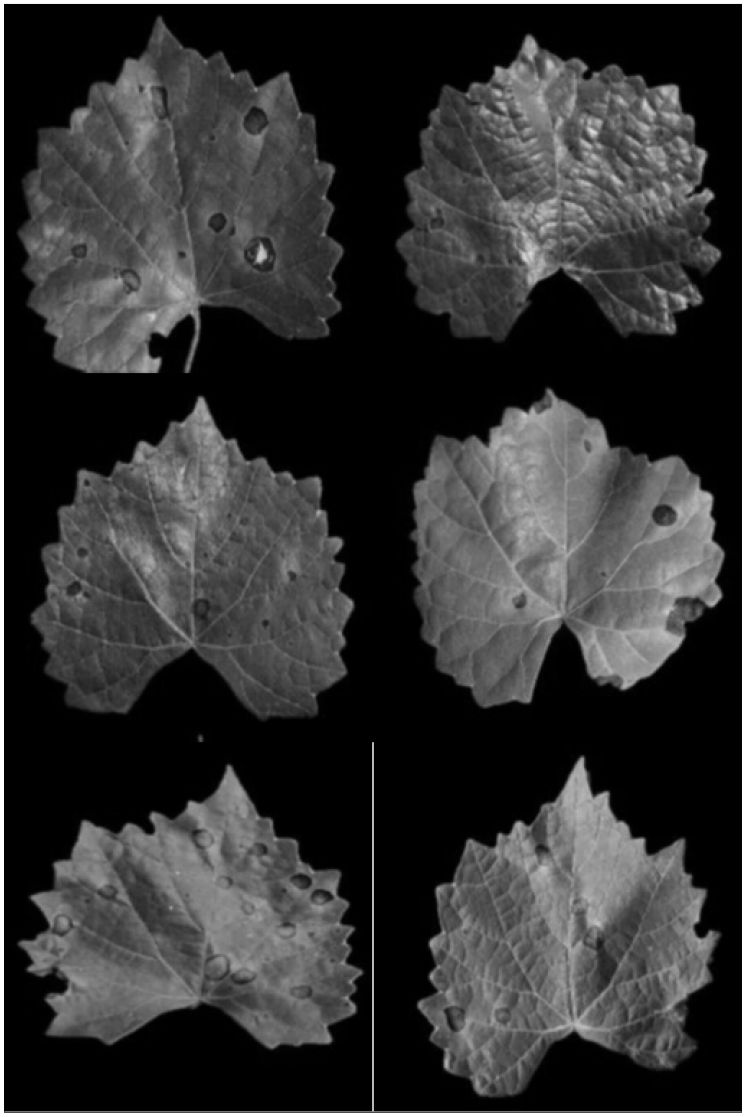

Figure 4: Grayscale images from *segmented* folder blurred

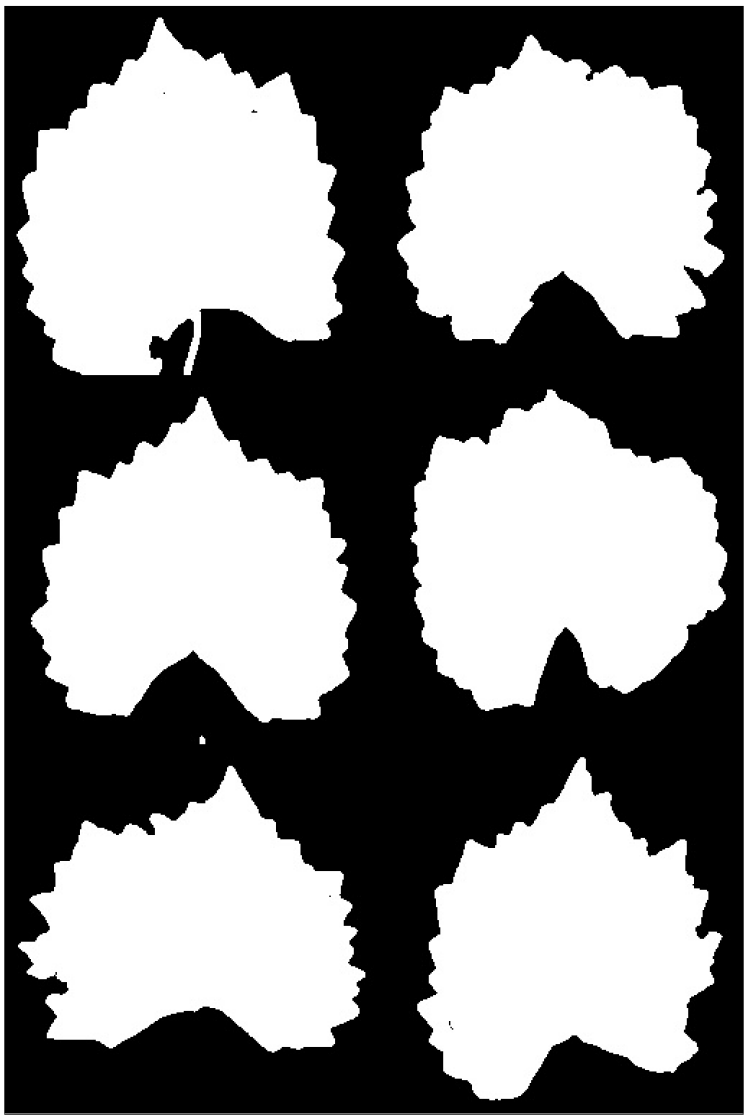

Figure 5: Binary masks of images from *segmented* folder

2.2 Segmentation and labeling

The second task is to segment the leaves from their backgrounds as binary masks. Code for this part was developed around an observation that all backgrounds in provided images are in low saturated gray-brown color. This finding proved to be helpful in determining that it is best to convert images to HSV colorspace (Fig. 6) and segment them on this basis (Fig. 7). In order to find HSV range satisfying our goal I used online color pickers, GIMP and manual testing:

```
# reading color image
img_col = cv2.imread(dir_col + filename[:len(filename) - 17] + '.JPG', 1)

# changing image to HSV colorspace
img_hsv = cv2.cvtColor(img_col, cv2.COLOR_BGR2HSV)

# finding HSV range adequate for leaves and thresholding
low_val = (18, 47, 42)
high_val = (180, 255, 255)
img_mask = cv2.inRange(img_hsv, low_val, high_val)
```

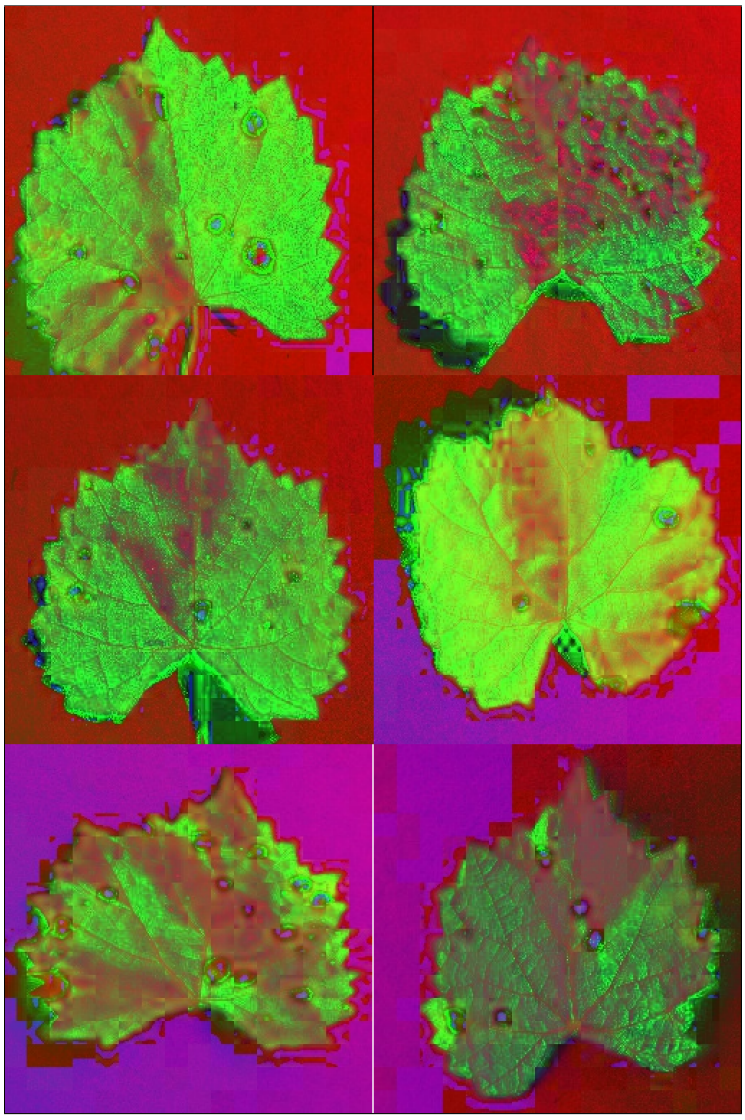

Figure 6: Images from *color* folder in HSV colorspace

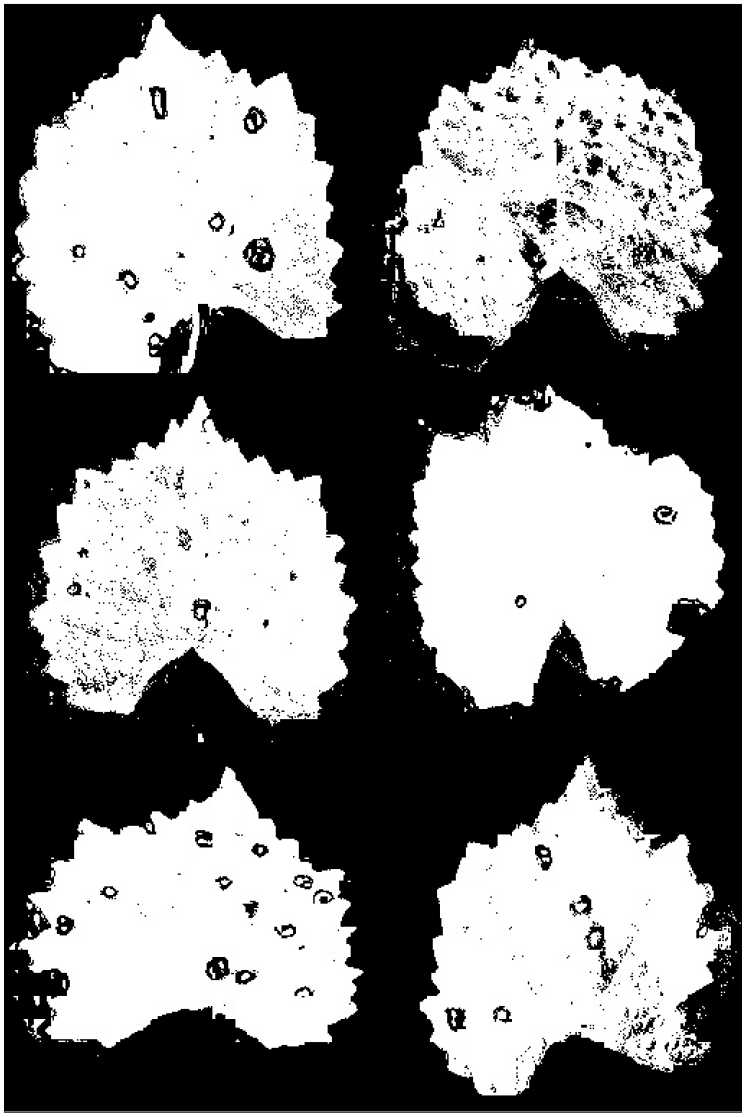

Figure 7: Binary masks of images from *color* folder after segmentation

We can see a lot of noise outside and inside the mask. It is caused by our range being not completely accurate to the colors of images. In order to clean them up and get our final mask (Fig. 8) noise reducing functions are used:

```
# applying noise removal
img_morph = cv2.morphologyEx(img_mask, cv2.MORPH_CLOSE, kernel=np.ones((5, 5), dtype=np.uint8))
img_res = cv2.morphologyEx(img_morph, cv2.MORPH_OPEN, kernel=np.ones((5, 5), dtype=np.uint8))
```

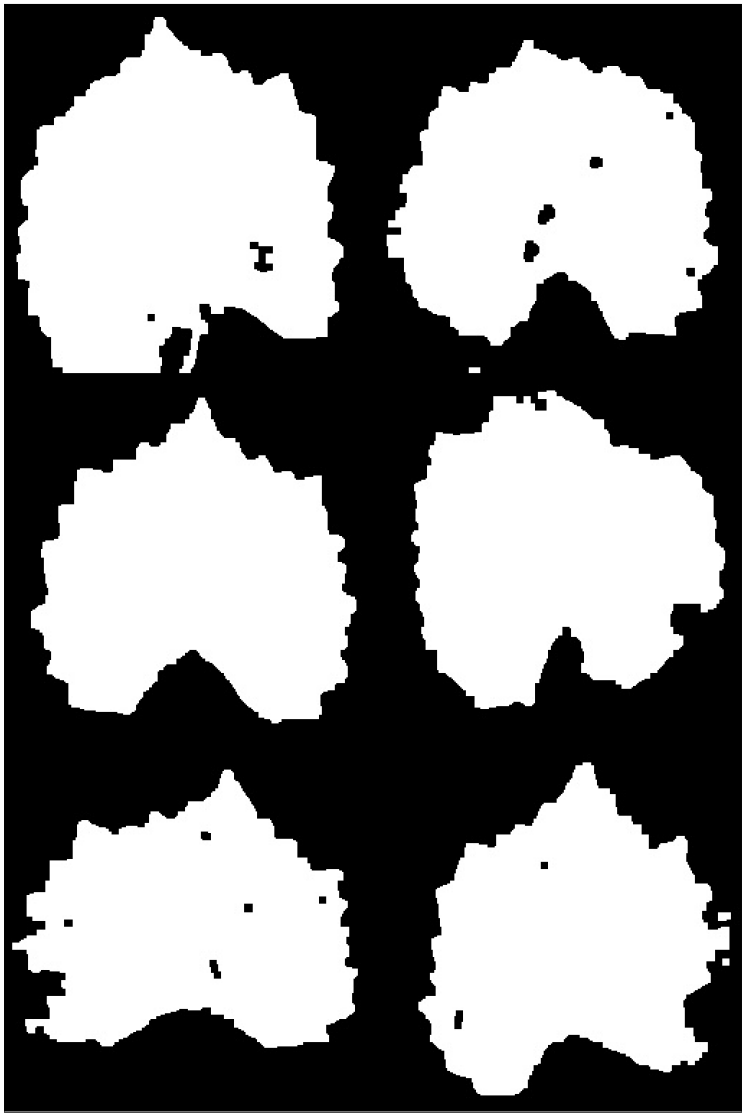

Figure 8: Binary masks of images from *color* after noise reduction

3 Results

The algorithm also compares the resulting segmentation from Section 2.2 masks with ground truth images prepared in Section 2.1. The accuracy evaluation is done by calculating Jaccardi index:

$$J = \frac{A \cap B}{A \cup B}$$

and Dice coefficient:

$$D = \frac{2|A \cap B|}{|A| + |B|}$$

for each image:

```
# finding Dice coefficient and Jaccardi index
d = 2 * np.sum(cv2.bitwise_and(img_gt, img_res)) / (np.sum(img_gt) + np.sum(img_res))
j = np.sum(cv2.bitwise_and(img_gt, img_res)) / np.sum(cv2.bitwise_or(img_gt, img_res))
```

With these values we can find mean accuracy, best and worst outcomes of our segmentation.

3.1 Mean results

Values of the calculation and counter of number of images are summed up iteratively in the main loop:

```
# incrementing the counter and adding to sums
c = c + 1
d_sum = d_sum + d
j_sum = j_sum + j
```

The final results are shown when the loop ends:

```
print("Mean Dice coefficient:")
print(d_sum / c)
print("Mean Jaccardi index:")
print(j_sum / c)
```

The accuracy obtained in this project across all images was 95,6526% for Dice coefficients and 91,8120% for Jaccardi indexes.

3.2 Best and worst cases

My algorithm checks for five best (Fig. 9) and five worst (Fig. 10) accuracies encountered during the loop. It saves each such image, their Dice coefficient and name in an array with predefined template:

```
# checking worst and best cases
worst.sort(key=lambda x: x[0], reverse=True)
best.sort(key=lambda x: x[0])
for i in range(5):
    if d < worst[i][0]:
        worst[i][0] = d
        worst[i][1] = filename
        worst[i][2] = img_res
        break
    if d > best[i][0]:
        best[i][0] = d
        best[i][1] = filename
        best[i][2] = img_res
        break
```

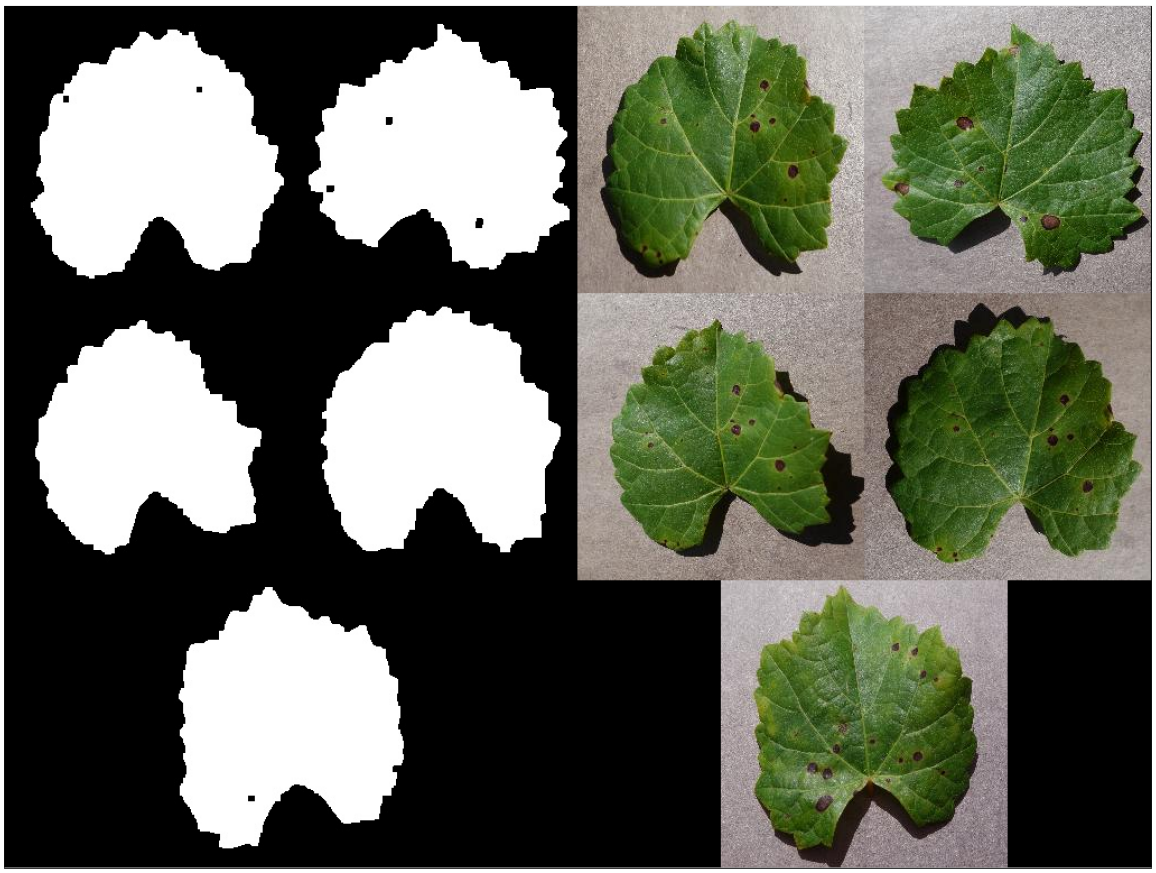

Figure 9: Cases with best accuracies: 99,0511%; 98,7689%; 98,9512%; 98,9079%; 98,8955%

We can see even the best cases have some unmasked parts with the highest accuracy being 99,0511%. Possible improvement for this problem could be adding another HSV threshold with range adequate for these inconveniences - parts of the leaves illnesses. Then adding this mask with the current one would possibly give higher results.

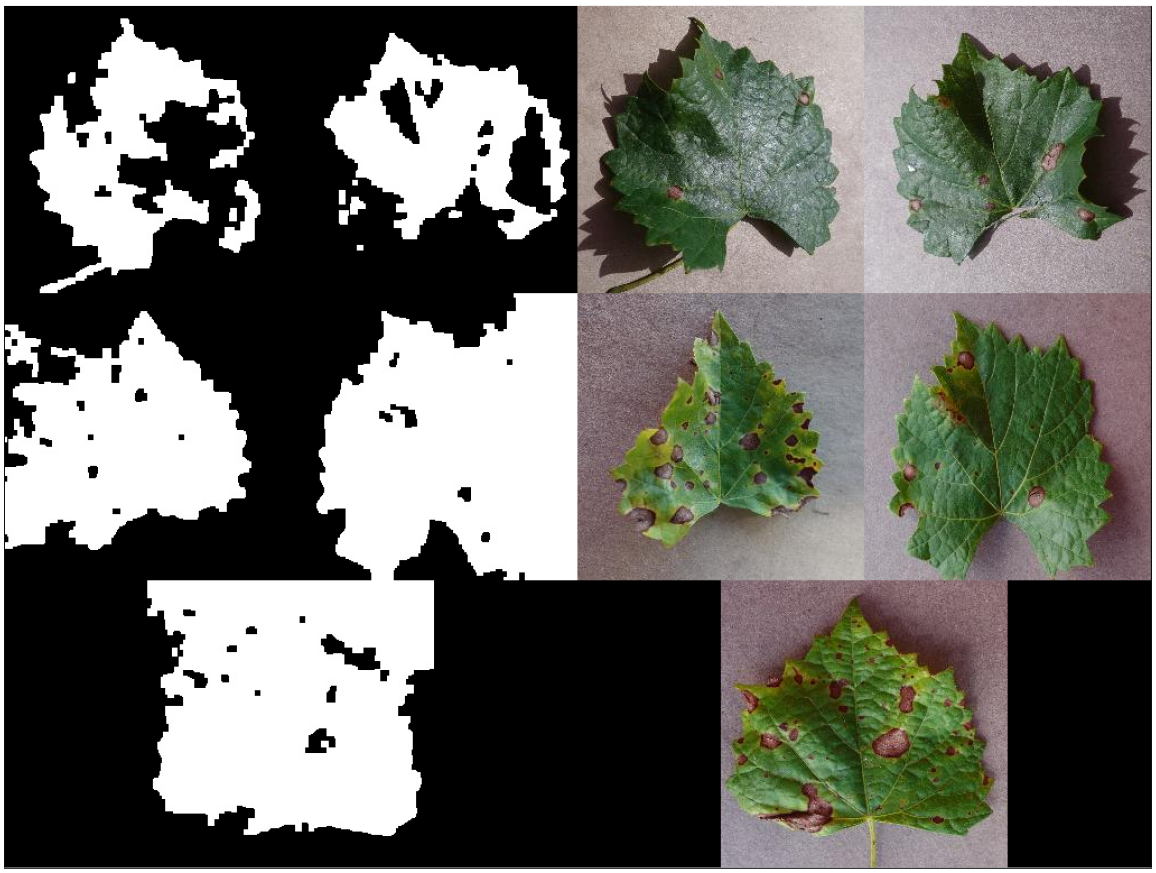

Figure 10: Cases with worst accuracies: 78,6791%; 78,8697%; 78,7453%; 77,3018%; 78,1085%

There were also no utterly wrong results with the lowest accuracy being 77,3018%. This leads to the conclusion that the algorithm of finding leaves worked correctly. As we can see two of these failures were due to strong light pointing at the leaves in the moment of taking a photograph, which turned the leaves almost white in some places. Other three failures were caused by slightly different color of the background. It is more saturated and fits in our HSV range for thresholding. We could improve these results by splitting the range into two or more separate ranges (i.e. for greens on leaf, for browns on leaf, for yellows on leaf...) and then combining masks resulting from these thresholdings.

3.3 Saving the results

Each iteration of the loop, a ground truth and my segmented masks are being saved on the drive:

```
# saving all the resulting ground truths and segmentations
os.chdir(dir_gt)
cv2.imwrite(filename, img_gt)
os.chdir(dir_parent + dir_res)
cv2.imwrite(filename, img_res)
os.chdir(dir_parent)
```

After the loop ends, the algorithm sorts, prints and saves duplicates of best and worst cases in different folders:

```
# printing and saving best and worst cases
print("Best cases:")
best.sort(key=lambda x: x[0], reverse=True)
os.chdir(dir_parent + dir_res + '/best')
for i in range(5):
    cv2.imwrite(best[i][1], best[i][2])
    print(str(i+1) + '. ' + best[i][1] + ' accuracy:' + str(best[i][0]))

print("Worst cases:")
worst.sort(key=lambda x: x[0])
os.chdir(dir_parent + dir_res + '/worst')
for i in range(5):
    cv2.imwrite(worst[i][1], worst[i][2])
    print(str(i+1) + '. ' + worst[i][1] + ' accuracy: ' + str(worst[i][0]))
```