INTRODUCTION TO IMAGE PROCESSING AND COMPUTER VISION

Project 1: Plant segmentation and labeling

Szymon Majorek

s.majorek@student.mini.pw.edu.pl

Table of contents

1. Introduction

1) General Introduction

In computer vision, image segmentation is the process of partitioning a digital image into multiple segments. The goal of segmentation is to simplify (and/or) change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects (our case) and boundaries. More precisely, it's the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics. Result of image segmentation is a set of segments, which combined, covers the entire image. It can also be set of contours in case we are detecting edges.

In our task we are given set of 900 images of plants, from multiple cameras, at various stages of it's growth. Our task is to first find the plant in an image and then to find individual leaves for each plant. Our segmentation is basically plant and not plant.

Figure 2 - Example 1 of input image

Figure 1 - Example 2

2) Part 1

This part, we could call the initial one, since we are supposed to separate everything that is plant from everything that is not a plant. Plant's most prominent characteristics when compared to it's background is it's color. We will discuss that more in depth in chapter 2. Above this point we can see some examples of images that we need to process.

3) Part 2

Here however the task is more complicated, since we not only have to have our plants separated from the background, we also need to separate each individual leaf. For this we can use beforementioned object location and recognition of separate objects even though they do not appear to be separated on the initial image. In part 2 of this project, our input will be the output of part 1, since it's pointless to perform the same task twice.

2. Part 1 – plant segmentation

1) Short introduction

As mentioned before, the goal of this part of the project is to, for each input image in our set of 900 images, produce output image which will be a binary mask for the plant present on input image, so in short the output in this part is binary mask containing plant for an input image. We also add some output used for comparison (Dice coefficient as well as Jaccard index more about which a little later).

2) Ground truth for part 1

In addition to plant images (Figures 1 and 2), we are given our ground truth set containing images of plants with leaves labeled from youngest to oldest according to this color table:

From oldest	R	G	B
1	0	0	255
2	0	255	0
3	0	255	255
4	255	0	0
5	255	0	255
6	255	255	0
7	128	128	128
8	0	0	128

Figure 3 - Ground truth color scheme

Below we can see some examples of said ground truth images we are given.

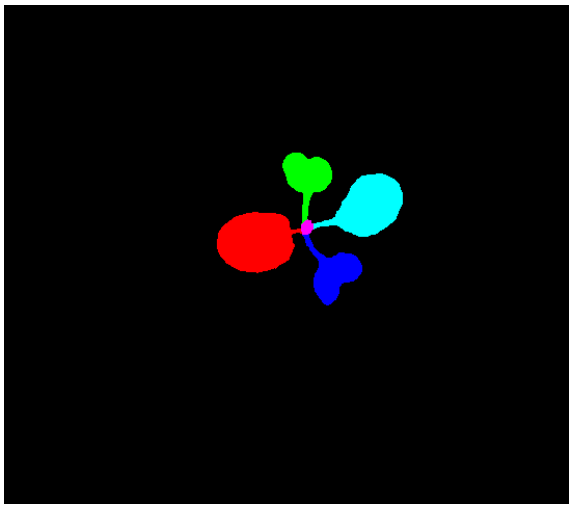

Figure 4 - Ground truth example 1

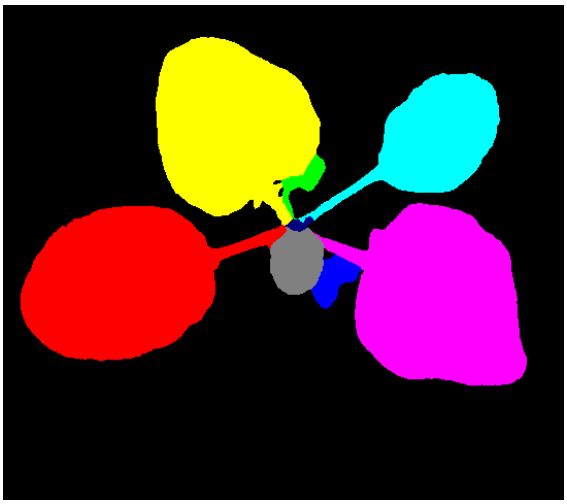

Figure 5 - Ground truth example 2

We can see that on the left picture we theoretically have 5 different leaves whereas on the picture on the right we have 8 different leaves, which is the maximum number of them in our set.

Now those ground truth images are virtually useless to us at this stage since to make a comparison and to calculate both Dice coefficient as well as Jaccard index we need to turn them into binary images. Since plants on them are separated flawlessly, we can simply separate what is colorful from what Is not colorful and it'll produce us binary masks which we can use later on for comparison purposes.

The algorithm used for this is rather simple, we only convert the image to HSV scale and then apply thresholding:

```
mask = cv.inRange(hsv, (0, 0, 0), (255, 255, 255))
imask = mask > 0
green = np.zeros_like(img, np.uint8)
green[imask] = img[imask]
grey = cv.cvtColor(green, cv.COLOR_BGR2GRAY)
ret, thresh = cv.threshold(grey, 0, 255, cv.THRESH_BINARY)
```

Figure 6 - Code snipper for ground truth to binary mask transformation (full code in ground_truth_init.py)

After those operation we get binary masks which we later on use to make comparisons to calculate how good was our algorithm of separating plants from not plants.

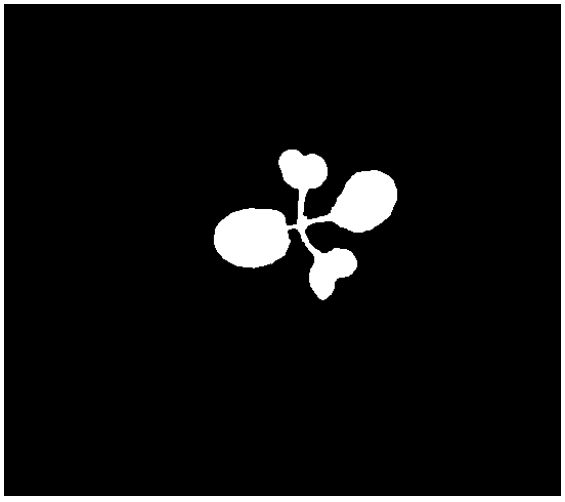

Figure 8 - Binary mask for ground truth example 1

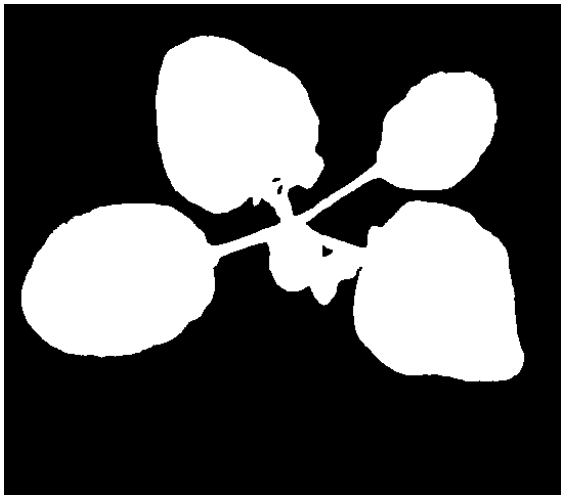

Figure 7 - Binary mask for example 2

Figures 8 and 9 are the resulting masks after converting ground truth images to binary masks.

3) Description of algorithm used

Having those binary masks we can proceed to our actual algorithm. First part of this algorithm is very similar to the one used to obtain ground truth binary masks, however this case is not so straight forward. If we take a look again at some example images:

Figure 9 - Example plant, clear image

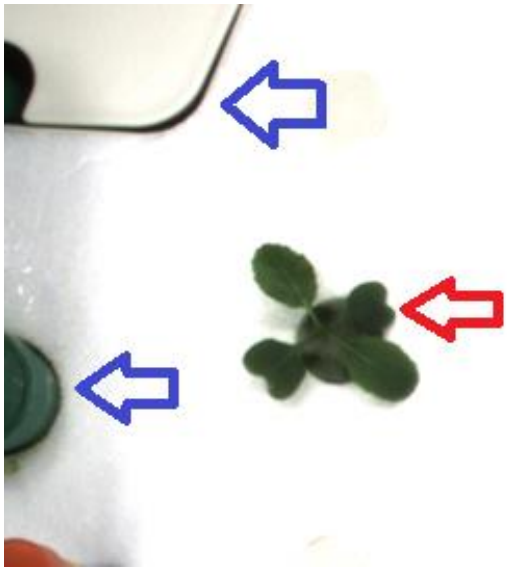

Figure 10 - Example plant, regions of interest may cause problems

In case of actual images of plants we need to take into consideration that plants may not be as prominent as it was the case in ground truth images. As marked on figure 10, there may appear some regions which will have color similar to that of plant (blue arrows) and slightly more dense regions, where the lightning will make plant appear very similar in color to it's surroundings, again making it hard to separate. Therefore we first apply thresholding for some specifically picked range in HSV (in my case the range was picked using mostly trial and error)

```
LOWER_BOUND = np.array([40, 40, 37], np.uint8)
UPPER_BOUND = np.array([70, 200, 150], np.uint8)
```

Which gave me the following results for example from figures 9 and 10;

Figure 9.2 - Green mask of example from figure 9

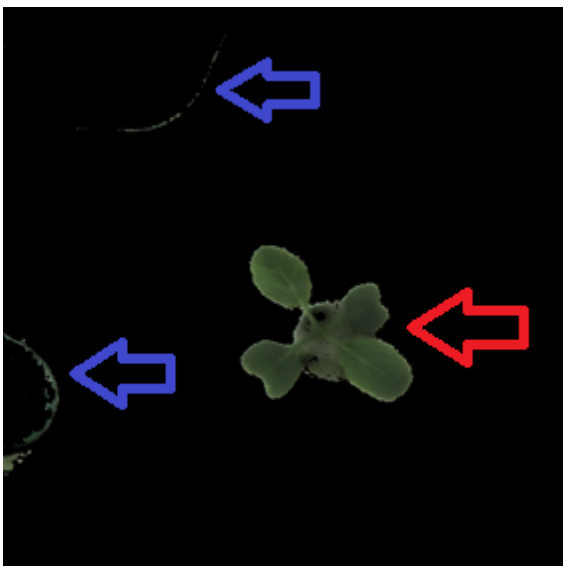

Figure 10.2 - Green mask for fig. 10

We can see that just as we predicted, fig. 9 example gave us rather clear plant with little to no noise and clear image whereas fig. 10 example produced some noise in regions marked by blue

arrows, as well as unclear image in region marked via red arrow (part of plant's pot got also marked as plant). Now we can transform those green images to binary masks:

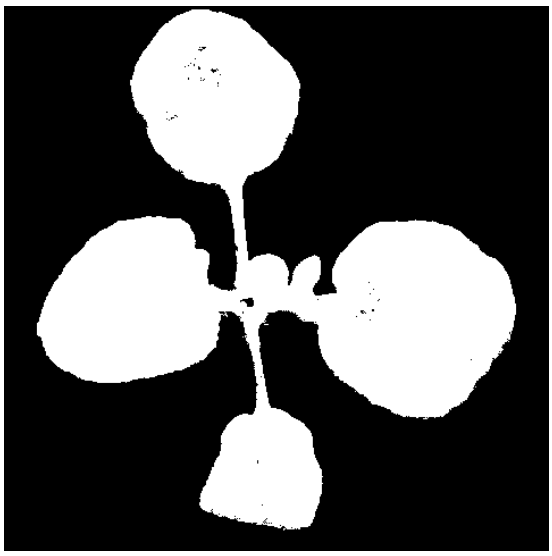

Figure 9.3 - Binary mask for fig. 9.2

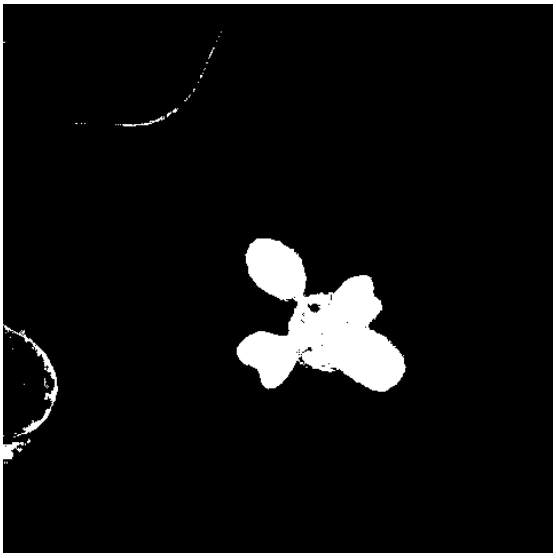

Figure 10.3 - Binary mask for fig. 10.2

Here it's even more evident that there's a vast difference in quality between those 2 images and their masks respectively. We can attempt to slightly correct them using 2 operations: opening and closing. Opening serves as a basic workhorse of morphological noise removal. Opening removes small objects from the foreground of an image, placing them in the background, whereas closing removes small holes in the foreground, changing small islands of background into foreground. First let's see how those operation affect the first plant. After opening we obtain the following result:

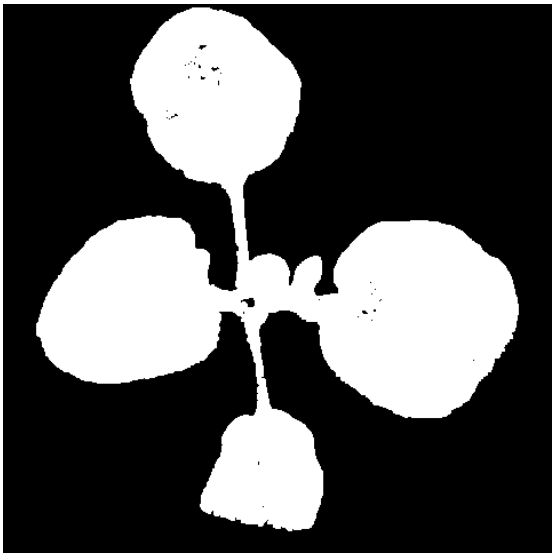

Figure 9.4 - 9.3 after opening

On the first look it might appear that not much has changed, therefore we will look at two specific regions to see what the operation of opening did to the binary mask.

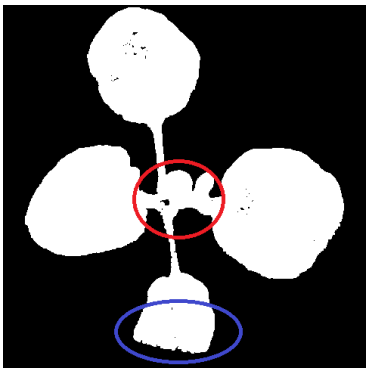

Figure 9.4.2 - 9.4 with regions marked

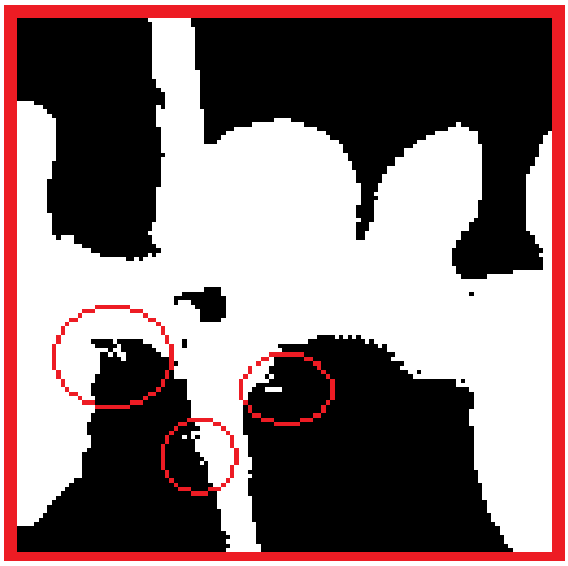

Figure 9.5.1 - before opening

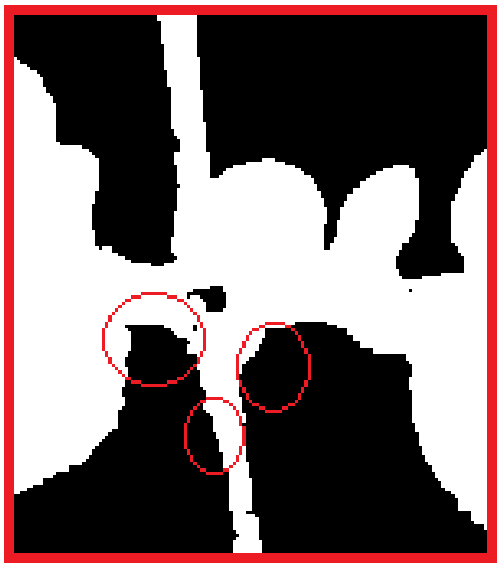

Figure 9.5.2 - after opening

Figure 9.5.3 - before opening

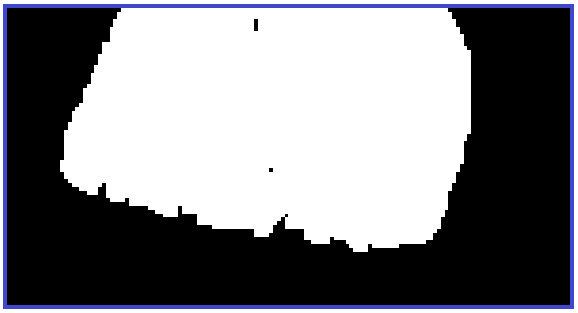

Figure 9.5.4 - after opening

As we can see the process of opening very slightly reduced the noise present in the picture. Now let's see what the process of closing will change.

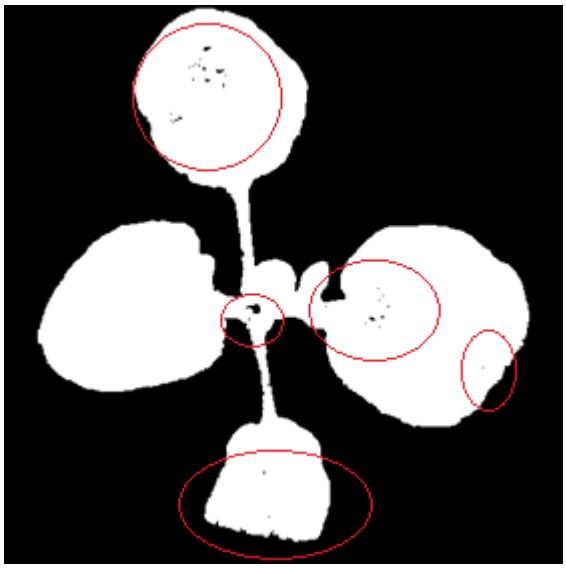

Figure 9.5.5 - before closing

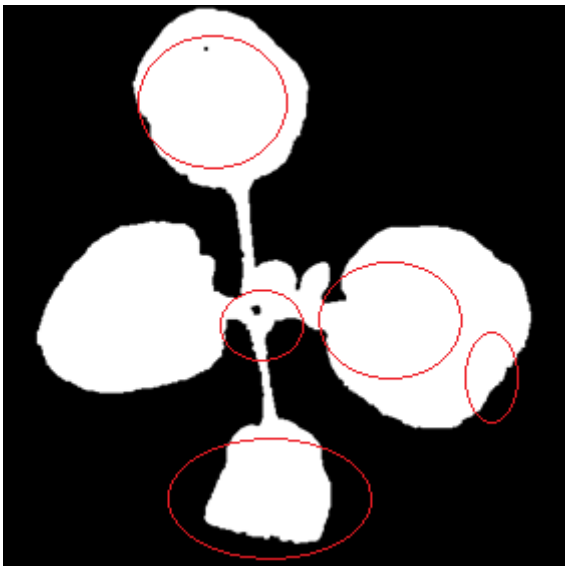

Figure 9.5.6 - after closing

In this case we can clearly see regions of interest at the first glance. We immediately notice that most of the holes present in gaps before closing operating were, in fact, closed after it and the image looks much smoother thanks to this.

Figure 9.6.1 - original image

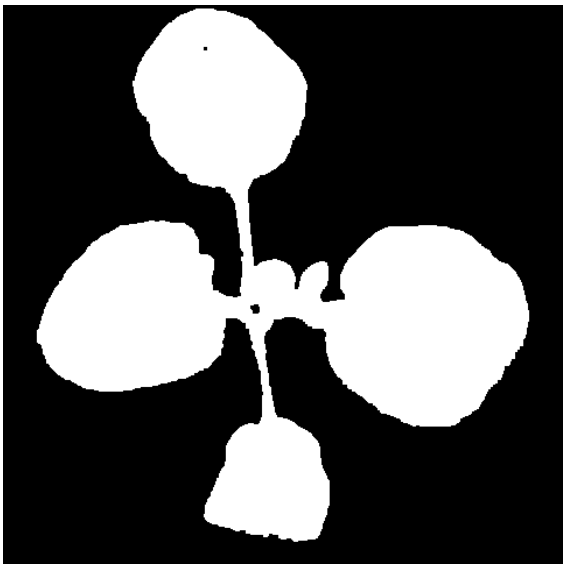

Figure 9.6.2 - obtained binary mask

As we can see, for this particular image our result was very satisfactory.

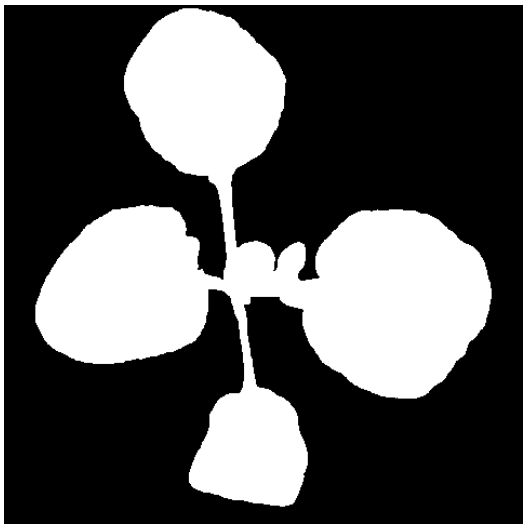

Figure 9.6.3 - ground truth binary mask

Actually we can also compare our result to the ground truth binary mask we obtained earlier. Clearly we see that our result is nearly perfect in this case. Although we need to remember that this example was pick specifically to showcase example with good conditions, where the plant can be easily separated from the rest of the image. Now, knowing what to look out for we can move on to the second example, quickly going through each stage of the process.

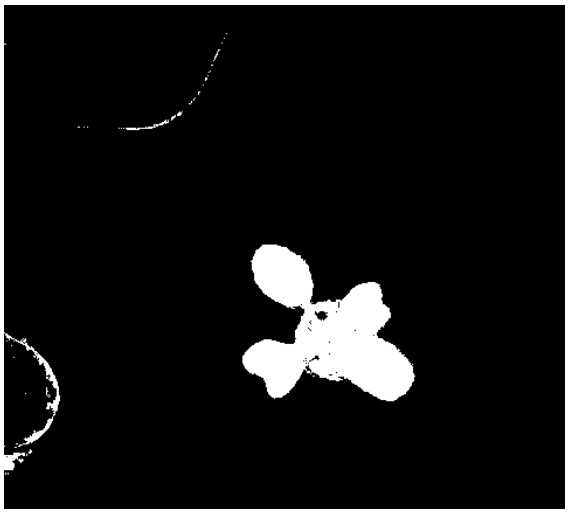

Figure 10.4.1 - before opening

Figure 10.4.2 - after opening

We can see that opening managed to get rid of the noise in the upper part of the image, however it wasn't able to do anything useful about the mess in the middle of the plant.

Figure 10.5.1 - original image

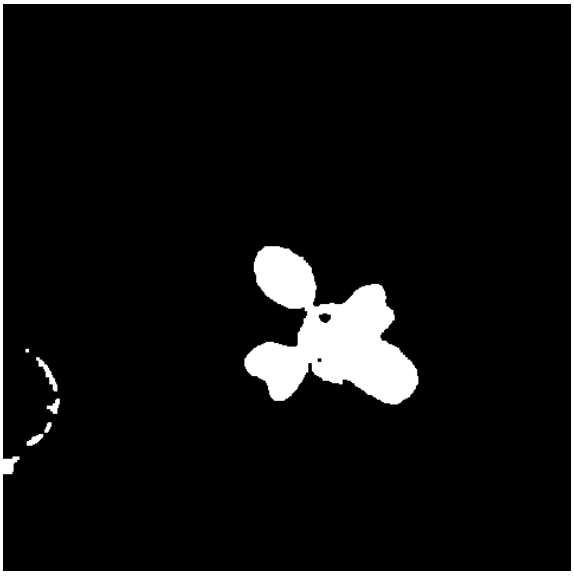

Figure 10.5.2 - after closing - final result

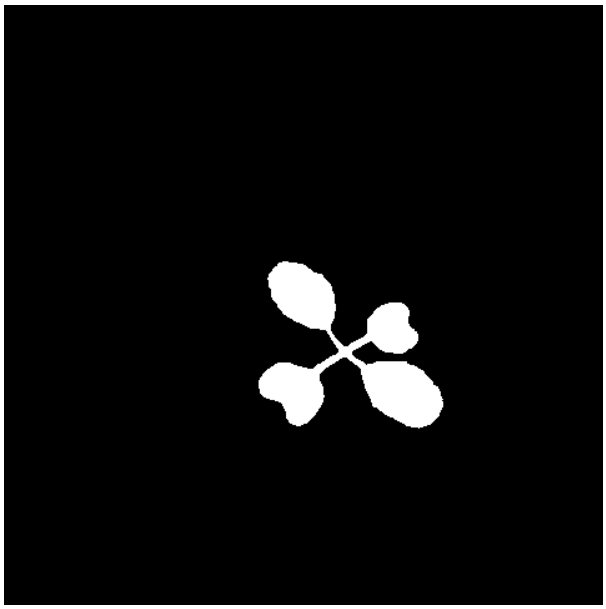

Figure 10.5.3 - ground truth binary mask

In contrast with the previous example, in this case our result could be labeled as rather terrible. The bottom right part of our plant looks like one big mess, whereas on the ground truth image It has clear stems going to each individual leaf. The lack of quality of the mask we obtained comes from the fact that on original image the pot in the near proximity of bottom right leaves has nearly the same color as the leaves, making it hard to segment the leaves without the pot. Additionally we were only able to get rid of the noise from the top of the image, however we the one on the lower left is still there, making the result worse. Those 2 example showed pretty well that even though we use the same algorithm for both images, results for one can be almost perfect, even when results for the other one are terrible.

4) Results

There are 2 types of assessment method we use to determine the quality of our prediction.

- Intersection over Union metric (Jaccard index) – statistic used in understanding the similarities between sample sets. It emphasizes the similarity between finite sample sets.

$$IoU = \frac{target \cap prediction}{target \cup prediction}$$

- Dice coefficient – statistic also used to gauge the similarity of two samples. Where A is target and B is prediction.

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

The results obtained for example used in algorithm description are as follows:

For the first example, where our result was very good, we obtained:

```
$ python comp_good.py

Dice: 0.9873771824658591 Jaccard: 0.97506906359294
```

Where for the 2nd example we got:

```
$ python comp_bad.py

Dice: 0.8440679771988706 Jaccard: 0.730205548898516
```

As we predicted, the difference here is rather significant – 14 percentage points. The difference appears due to the reasons I mentioned earlier, during algorithm analysis.

To calculate Dice coefficient and Jaccard's index I used simple script and the fact that $J = \frac{D}{2-D}$, where J is Jaccard's index and D is Dice coefficient.

The results for each individual plant are as follows:

```python
def calculateDiceC(im1, im2):
    im1 = np.asarray(im1).astype(np.bool)
    im2 = np.asarray(im2).astype(np.bool)
    intersection = np.logical_and(im1, im2)
    return 2 * intersection.sum() / (im1.sum() + im2.sum())
```

Plant Id	Dice coefficient	Jaccard's index
0	0.9253934259647113	0.8611462541958379
1	0.9348765191709363	0.8777165615044495
2	0.9311788492832148	0.871220455039402
3	0.9178763088482359	0.848217552534396
4	0.922676941918169	0.8564533498066876

Figure 11 - Part 1 results per plant

Whereas for the whole set, the result is:

Dice coefficient	Jaccard's index
0.9264004090370536	0.8628919169074337

Figure 12 - part 1 results for the whole data set

As we can see the algorithm to separate plants from given image works rather well for this set, however it's definitely not perfect. It takes between 7 and 7,5 seconds to execute python script calculating this for all 900 photos (not taking into account writing results to files and calculating dice coeffs and jaccard's indexes).

2. Part 2 – leaf labeling

1) Short introduction

Having obtained the binary masks for each plant in the first part of this project, now we have to find and label each individual leaf for each plant. This task is much more sophisticated since we have to first find leaves, then mark them and then use that information to color each leaf in a given color.

2) Description of algorithm

The algorithm takes binary masks obtained in the 1st part of the project and then performs few operations on them in order to obtained desired result. First we erode the image, with iterations parameter bigger for later days (so that once the plant gets bigger, we erode a little more than for smaller plants). After erosion, ideally we have all leaves split so we can begin to actually find each leaf individually hence we find contours using cv2.findContours() function. Then, if the contour area isn't too small, we find center of each contour, which is in theory the center of the leaf and then we use that information to color every pixel in the binary mask according to to which leaf center it's closest.

Similarly to par 1, we will use 2 examples to present how the algorithm works step by step. First, lets take a look at original image and it's binary mask obtained after plant separation.

Figure 13.1.1 - part 2 example 1 orginal image

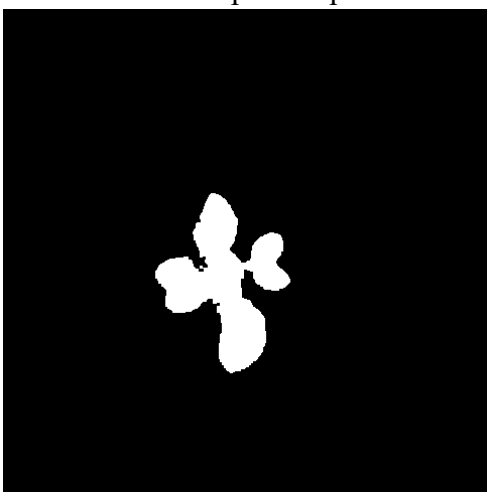

Figure 13.2.1 - part 2 example 1 binary mask

Figure 13.1.2 - part 2 example 2 original image

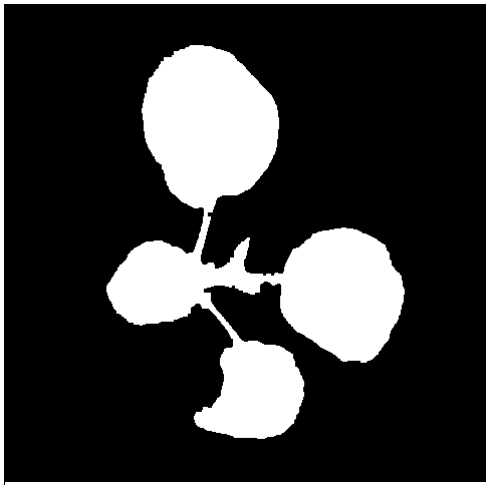

Figure 13.2.2 - part 2 example 2 binary mask

Right off the bat we notice that in example 1 the binary mask looks sort of "fat" which may cause troubles later on as it might be hard to distinguish separate leaves. In fact I would risk saying that even for a naked eye, in this form the image makes it sort of hard to actually distinguish what should be a leaf and what should not be a leaf. Example 2 on the other hand looks much better in that regards, we can clearly see 4 bigger masses in the image which are most likely individual leaves.

Ok having seen those images, we can proceed through to the next step which is erosion.

```
day = int(file.split("_")[3])
if day <= 2:
    iters = 2
elif day <= 4:
    iters = 5
elif day <= 7:
    iters = 7
else:
    iters = 10
erosion = cv.erode(img, kernel, iterations=iters)
```

Function erode from Open CV library takes a parameter iterations which basically indicates how much we actually want to erode, for example:

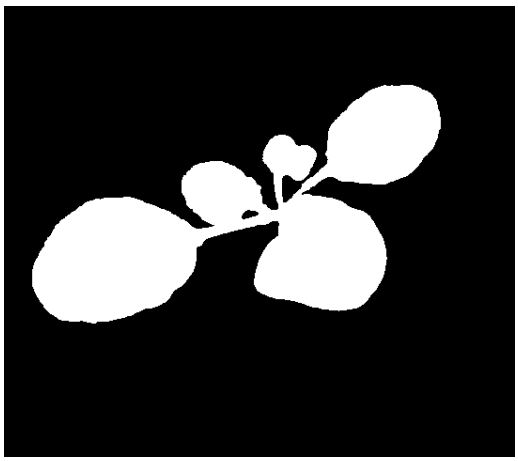

Figure 14.1 - erosion example, original binary mask

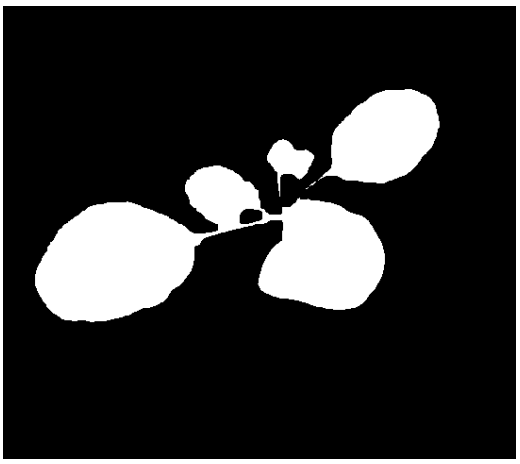

Figure 14.2 - erosion, iterations == 2

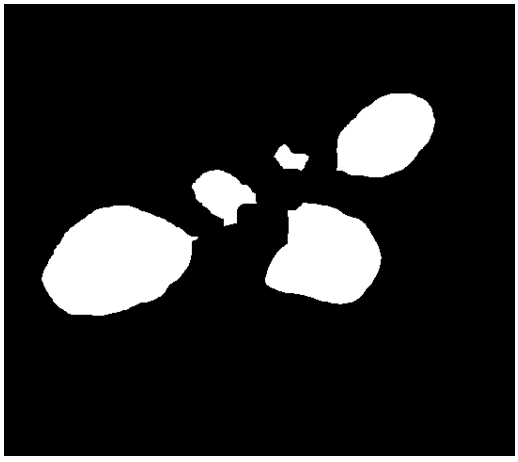

Figure 14.3 - erosion example, iteartions == 5

Figure 14.4 - erosion example, iterations == 7

As we can see the iteration parameter changes quite a bit, in this particular case, if we used iterations == 2, we would've gotten only 3 leaves (since there are only 3 fully disconnected shapes), whereas for iterations == 5 we get 5 leaves. However we mustn't forget that if we go too far with iterations it's not good either. In this example if we took iterations == 7, there's a chance that the small leaf (red circle) would be too small to pass MIN_AREA condition and thus wouldn't be actually marked as a leaf. Also knowing that as the day progresses the leaves grow, and we get more cluttered images and so on, I use simple set of ifs and elifs to pick adequate iterations parameter. It's obviously not ideal since not every case is the same, even if the day is the same, however it's something.

Now back to our examples, after erosion we get the following results:

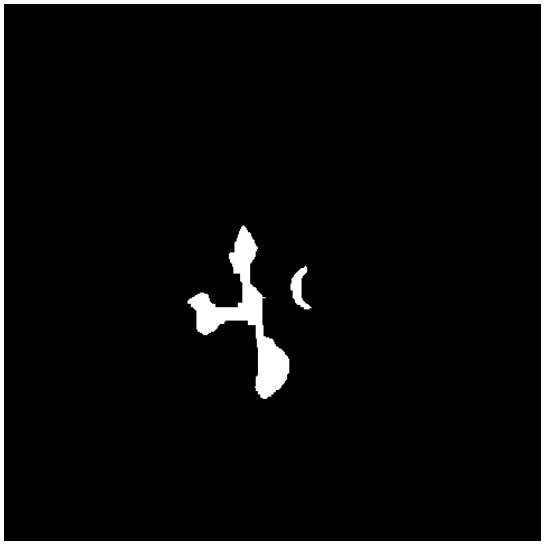

Figure 15.1 - example 1, erosion

Figure 15.2 - example 2, erosion

We can already tell that the results for example 1 are going to be utterly terrible due to the fact that after erosion we have only 2 closed shapes on the image, meaning that we will detect at most 2 leaves. From this image we can deduct that increasing the number of erosions for this image would be helpful, however we can't really set iterations parameter individually for each plant. Anyhow let's see results of the next step of the algorithm, meaning finding shapes and their centers.

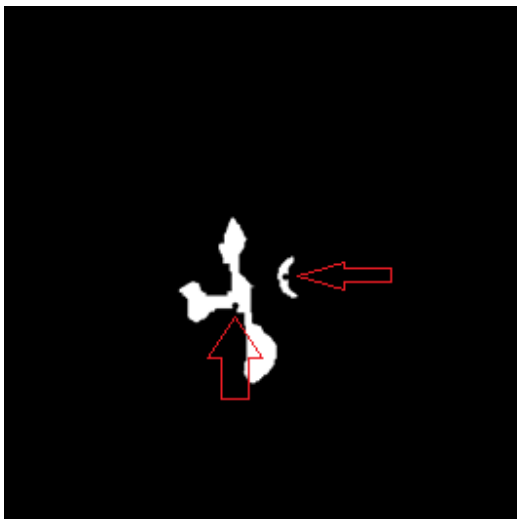

Figure 16.1 - example 1, center of shapes

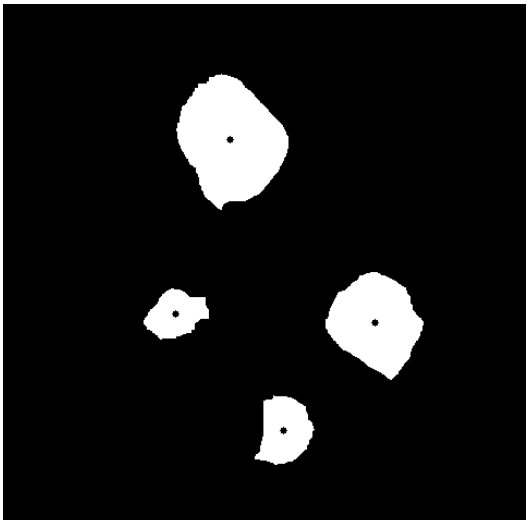

Figure 16.2 - example 2, center of shapes

Centers of shapes are marked with black circles on their respective shapes (marked with red arrows on example 1 as they might be hard to notice at first glance. Well here again we can see that example 1 is not going anywhere good or useful whereas example 2 looks promising. Next step in the algorithm is to use these centers that we've just calculated to color each pixel in the initial mask in it's own color, via calculating the distance between each individual pixel and all the centers and then picking the closest one. This is the last step since after this we should, at least in theory, get labeled leaves. One of biggest downfalls of this algorithm is the fact that I chose to go through each pixel individually meaning that for each image I do loop inside a loop where both loops have around 500 iterations.

```
rows = img.shape[0]
columns = img.shape[1]
for m in range(rows):
    for n in range(columns):
        if img[m][n] > 0:
            result[m][n] = leafColors[findClosestCenter(m, n, centres)]
```

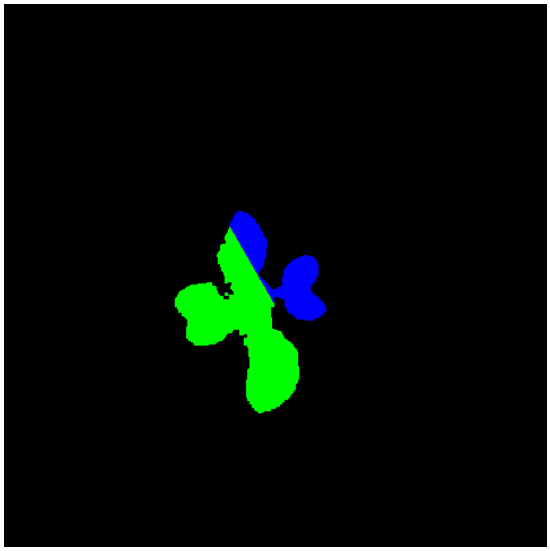

Figure 17.1 - example 1, final result

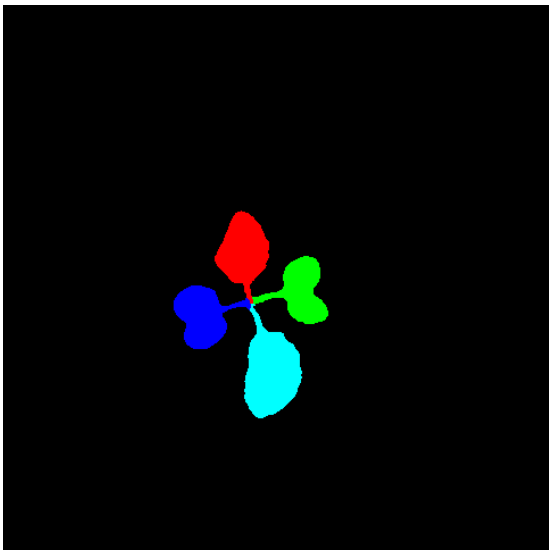

Figure 17.1 - example 1, ground truth

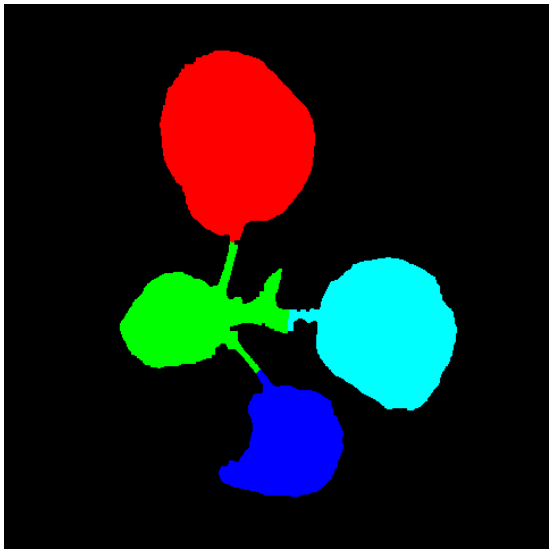

Figure 17.2 - example 2 final result

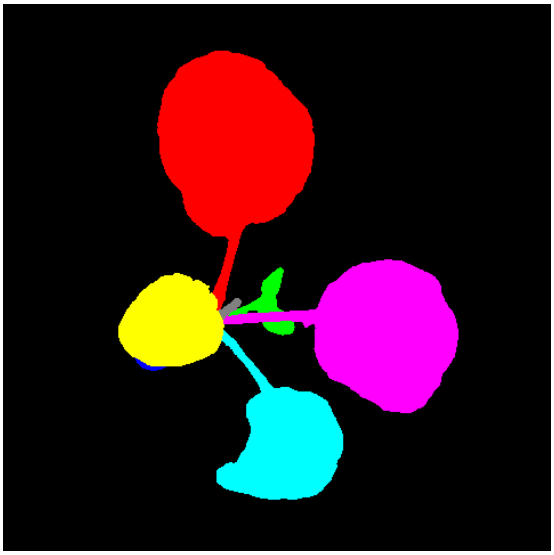

Figure 18.2 - example 2, ground truth

Now we clearly see that results for example 1 are indeed terrible, however the main reason for that is that the erosion failed to separate leaves as shapes. Issue with that comes directly from part 1, since the initial binary mask that the algorithm received left much to desire itself. In fact to see how the algorithm would perform under better conditions (meaning with better initial binary mask) we can use the ground truth masks for the part 1. Let's see how it would go using

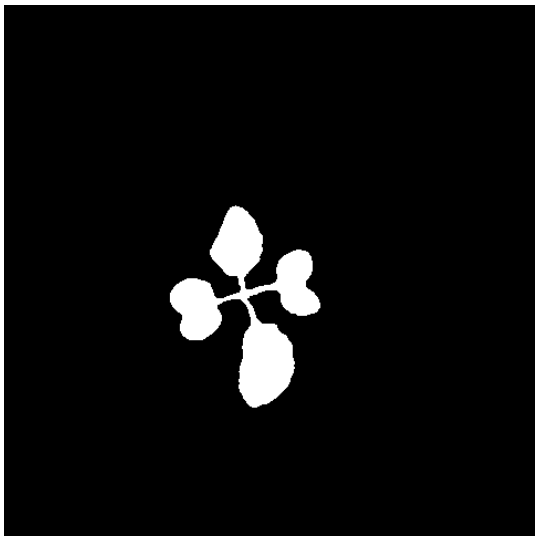

Figure 18.1 - example 1, ground truth binary mask

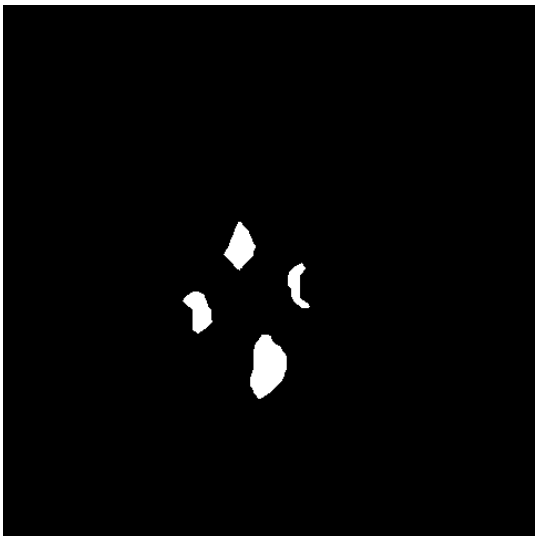

Figure 198.2 - erosion

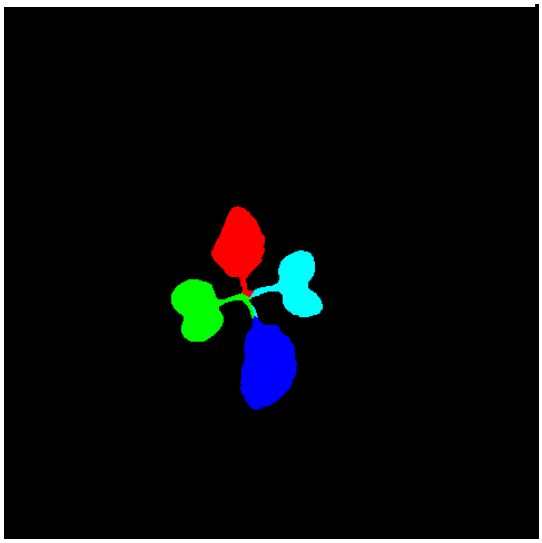

Figure 18.3 - result

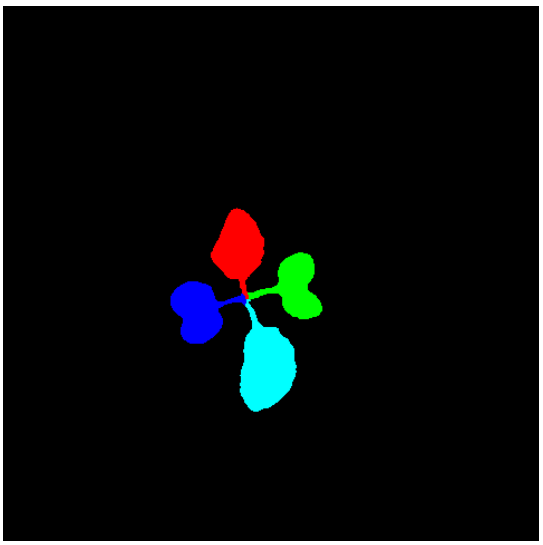

Figure 18.4 - ground truth

them.

Clearly we see that had we used the ground truth binary mask as our input binary mask we would have gotten nearly perfect result. Each leaf is colored individually, no two leaves overlap (color wise), the only issue is that the green of west leaf leaked into the stem that should be blue and the colors aren't exactly the same for each leaf. Other than that, the algorithm worked about as well as one could expect it to.

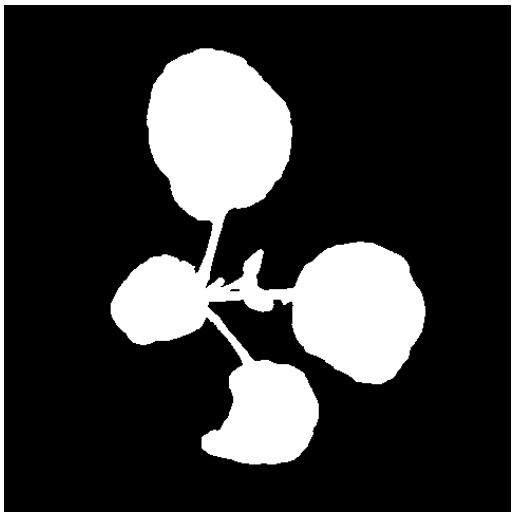

Figure 19.1 - example 2, ground truth binary mask

Figure 19.2- erosion

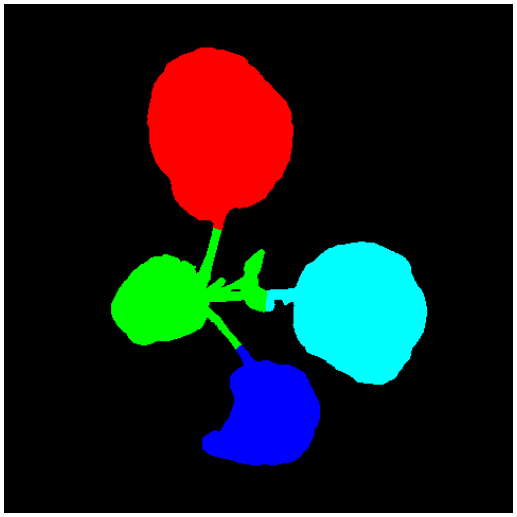

Figure 19.3- result

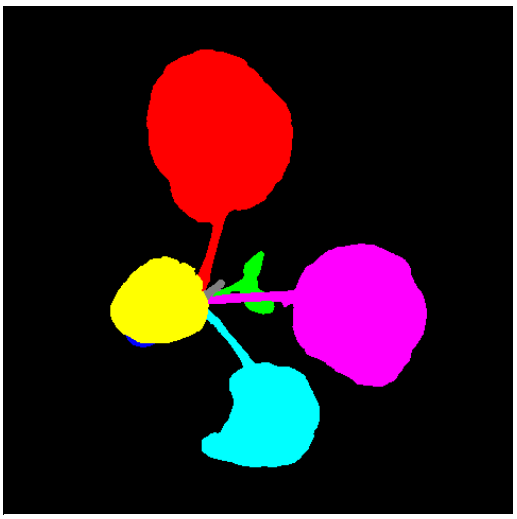

Figure 19.4 - ground truth

This comparison highlights something entirely different. Even though we were working on nearly perfect binary mask of the actual plant, our result is not all that good. Well it is decent, however we got exactly 4 leaves, whereas we should have ideally gotten 7 of them. This is one of the issues of this algorithm, if one of the leaves is too small, or too clumped together with something else, the erosion won't separate it from the rest at all, or it'll erode it too much, making it smaller than MIN_AREA and thus not labeled as a separate leaf. There's also one more issue which could be subtly seen on those 2 examples, let's call it color leaking. Figure 20 showcases one of the more extreme cases. If the distance of some part of given leaf to the center of another leaf is smaller than to the center of it's own leaf (the calculated center obviously), then it'll get colored in the other leaf's color. It can also happen if erosion doesn't split something up properly. There are few variables at play when using this algorithm. In some cases it'll work very well and in some other it'll be pretty much useless.

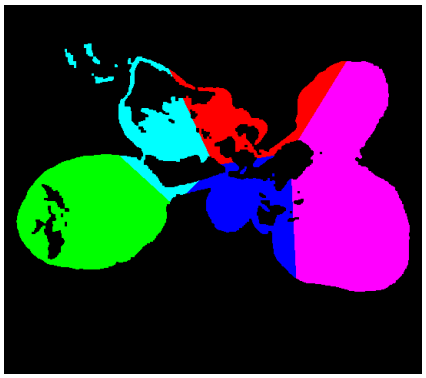

Figure 20 - color leaking extreme example

3) Results

Once again we use Dice coefficient and Jaccard's index to calculate the accuracy of our algorithm, however for that calculation we need to compare masks that are actually meant to be the same, even if their colors are not the same.

In general we are meant to calculate data for the whole set, each individual plant, so same as in part 1, however we additionally have to calculate the accuracy for each individual leaf, which is slightly more complex to do.

Plant 0

	My binary masks		Ground truth binary masks	
Leaf [R, G, B]	Dice coefficient	Jaccard's index	Dice coefficient	Jaccard's index
[0, 255, 0]	0.4549	0.3575	0.6559	0.5857
[255, 0, 0]	0.2291	0.2032	0.4371	0.4297
[255, 255, 0]	0.0567	0.0519	0.1742	0.1621
[0, 0, 255]	0.0668	0.0629	0.0700	0.0687
[255, 0, 255]	0.0000	0.0000	0.0000	0.0000
[0, 255, 255]	0.0000	0.0000	0.0000	0.0000
[128, 128, 128]	0.0000	0.0000	0.0000	0.0000

Dice coefficient	0.1615	0.2674
Jaccard's index	0.1351	0.2492

Plant 1

	My binary masks		Ground truth binary masks	
Leaf [R, G, B]	Dice coefficient	Jaccard's index	Dice coefficient	Jaccard's index
[0, 255, 0]	0.0455	0.0286	0.4904	0.4349
[255, 0, 0]	0.3242	0.2915	0.5001	0.4891
[255, 255, 0]	0.0056	0.0051	0.0181	0.0178
[0, 0, 255]	0.0886	0.0808	0.1054	0.1041
[255, 0, 255]	0.0389	0.0328	0.0095	0.0092
[0, 255, 255]	0.0149	0.0141	0.0000	0.0000
[128, 128, 128]	0.0000	0.0000	0.0000	0.0000
[128, 0, 0]	0.0000	0.0000	0.0000	0.0000

Dice coefficient	0.1035	0.2247
Jaccard's index	0.0906	0.2110

Plant 2

	My binary masks		Ground truth binary masks	
Leaf [R, G, B]	Dice coefficient	Jaccard's index	Dice coefficient	Jaccard's index
[0, 255, 0]	0.1402	0.1217	0.5437	0.5165
[255, 0, 0]	0.1042	0.0980	0.2299	0.2176
[255, 255, 0]	0.1195	0.1055	0.2401	0.2273
[0, 0, 255]	0.0000	0.0000	0.0000	0.0000
[255, 0, 255]	0.0000	0.0000	0.0000	0.0000
[0, 255, 255]	0.0000	0.0000	0.0000	0.0000
[128, 128, 128]	0.0000	0.0000	0.0000	0.0000
[128, 0, 0]	0.0000	0.0000	0.0000	0.0000

Dice coefficient	0.0728	0.2028
Jaccard's index	0.0651	0.1923

Plant 3

	My binary masks		Ground truth binary masks	
Leaf [R, G, B]	Dice coefficient	Jaccard's index	Dice coefficient	Jaccard's index
[0, 255, 0]	0.0442	0.0290	0.5543	0.4991
[255, 0, 0]	0.0942	0.0861	0.3567	0.3524
[255, 255, 0]	0.0590	0.0540	0.1514	0.1439
[0, 0, 255]	0.1086	0.1009	0.0062	0.0058
[255, 0, 255]	0.0000	0.0000	0.0000	0.0000
[0, 255, 255]	0.0000	0.0000	0.0000	0.0000
[128, 128, 128]	0.0000	0.0000	0.0000	0.0000

Dice coefficient	0.0612	0.2137
Jaccard's index	0.0540	0.2002

Plant 4

	My binary masks		Ground truth binary masks	
Leaf [R, G, B]	Dice coefficient	Jaccard's index	Dice coefficient	Jaccard's index
[0, 255, 0]	0.4583	0.3771	0.6733	0.6200
[255, 0, 0]	0.2598	0.2319	0.3713	0.3647
[255, 255, 0]	0.1048	0.0959	0.3075	0.2946
[0, 0, 255]	0.0461	0.0441	0.0945	0.0927
[255, 0, 255]	0.0273	0.0233	0.0323	0.0304
[0, 255, 255]	0.0194	0.0171	0.0000	0.0000
[128, 128, 128]	0.0000	0.0000	0.0000	0.0000
[128, 0, 0]	0.0000	0.0000	0.0000	0.0000

Dice coefficient	0.1831	0.2958

Jaccard's index	0.1579	0.2805

And then results for the whole sample:

Dice coefficient	0.1164	0.2409
Jaccard's index	0.1005	0.2266

The results we obtained using our masks and ground truth masks are vastly different. For ground truth binary masks we got dice coefficient equal 0.2409, compared to our masks which yielded 0.1164 dice coefficient. That suggests that even though masks play a very big role in obtaining good results, even if we have perfect ones we won't get to 0.3 dice coeff, which in turn means that the algorithm should be heavily improved.

Plant 4 gave the best result – 0.1831 whereas plant 3 was the worst, barely reaching 6%. Plant 2 was also terrible in the sense that we only detected 3 out of 8 leaves.

4. Conclusions

All in all I think it's safe to say that there are a lot of variables when it comes to separating certain objects in images. In our case the conditions on the picture had to be very good to obtain good results, especially when it comes to leaf labeling, since even a little of noise left in the image could cause some part of the image to not be eroded enough to separate leaf from some other element of plant. On the other extreme, if the plant was not very pronounced in the image, after we created it's binary mask we could find more leaves than there were, since after erosion some elements are split in two instead of being actually one leaf.

The algorithm gets worse for older leaves, which makes sense as at first they are clumped together with old ones and often can get marked as separate ones only after they grow for some time.

T in results section paint the picture that I wasn't really that successful in finding good algorithm for this process, however we have to remember that we're comparing results of something pretty much fully automatic to ground truth which was achieved via manual work so I think that it's entirely expected that the labeling was so far off.