INTRODUCTION TO IMAGE PROCESSING AND COMPUTER VISION

Project 1: Fingernails Segmentation

Emilia Wróblewska

December 29, 2020

Contents

1 Introduction

Image segmentation is the process of partitioning a digital image into multiple segments i.e sets of pixels. The goal of segmentation is to simplify the representation of an image into something that is easier to analyze. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics, such as color, intensity, or texture.[1]
In this project, our goal is to separate human nails from the photos presenting hands on different backgrounds. The images vary significantly in size, background colors and the area occupied by hand. Having input images as below, we want to obtain output masks showing only nails (marked as white shapes on black background) and optionally original pictures with nails marked by rectangles (bounding boxes).

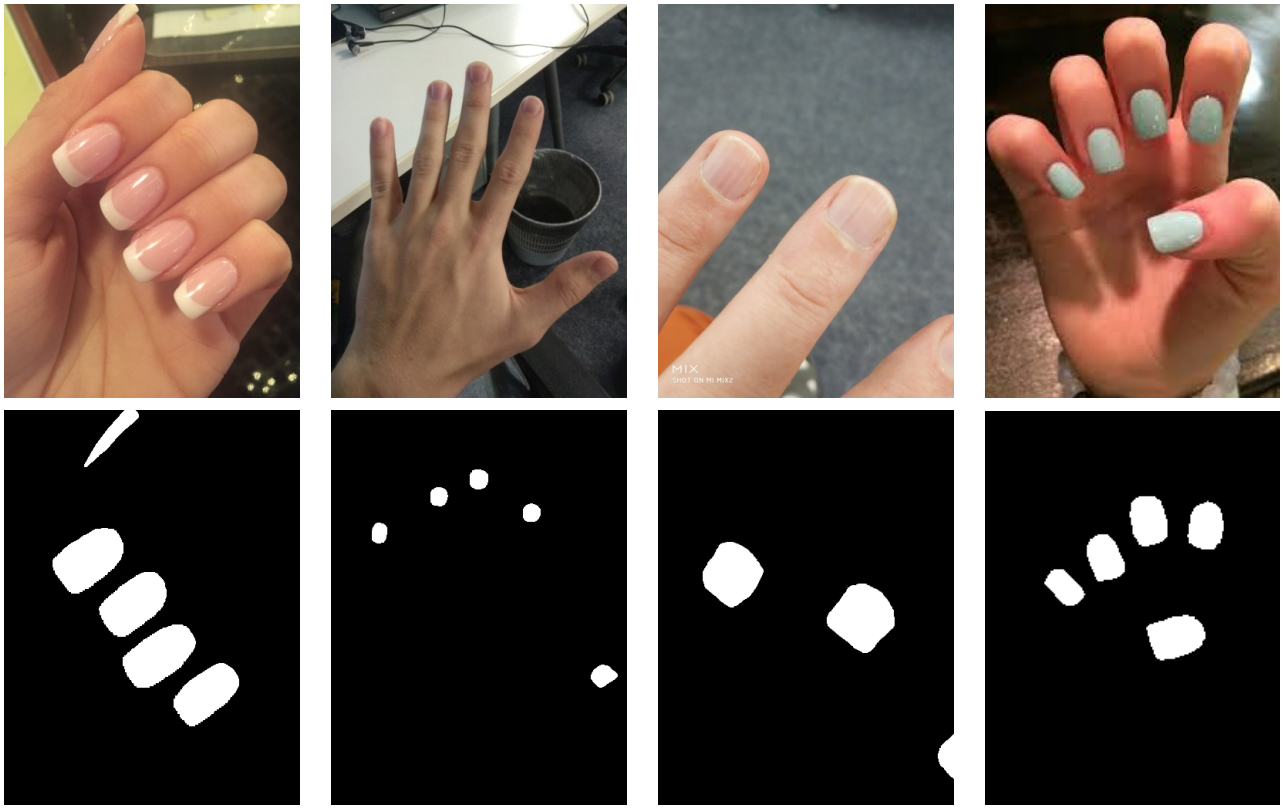
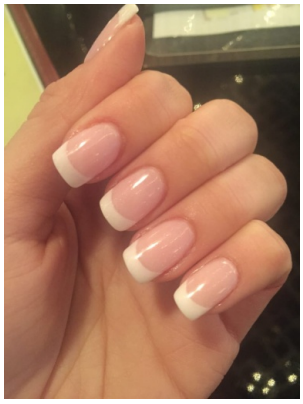


Figure 1: Sample of input images with their target masks

2 Algorithm description

The algorithm is build around an observation that majority of nails in our data set is almost the same color as skin, e.g. light pink. There are three main methods, each of them is responsible for obtaining better result if the previous one failed, and one method handling special cases (like blue nails) which is not universal but was designed to achieve better correctness of final results.
The functioning of the algorithm will be presented using 3 exemplary pictures, each of which needing a different method of segmentation.



2.1 Hand detection

(NOTE: Code from this section can be found in hand.py)
Firstly, all elements which are not part of the hand need to be removed, i.e. only elements which are in skin color range are left (here, YCrCb color space was used, since it gave more accurate results than HSV filtering). Then, after slight blur, morphology opening and threshold, the mask is cleaned from small background noises.

```
# YCrCb Skin detection
imageYCrCb = cv2.cvtColor(image,cv2.COLOR_BGR2YCR_CB)
min_YCrCb = np.array([0,133,77],np.uint8)
max_YCrCb = np.array([235,173,127],np.uint8)
skin = cv2.inRange(imageYCrCb,min_YCrCb,max_YCrCb)
skin = cv2.blur(skin, (2,2))

# Morphology opening to remove small noises
kernel = np.ones((5,5),np.uint8)
skin = cv2.morphologyEx(skin, cv2.MORPH_OPEN, kernel)
ret,thresh = cv2.threshold(skin,200,255,cv2.THRESH_BINARY)
```

Figure 2: Skin detection

Figure 3: Blur and opening

After that, the area of current mask is calculated using contours. The only purpose of this step is to filter out images with colorful nails, where skin detection appeared to clear everything but nails from the picture. This topic will be elaborated more in *Section 2.3*.

```
# Calculate area of obtained mask
area = image.shape[0]*image.shape[1]
whiteArea = 0
segmented = image.copy()
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

for i in range(len(contours)):
    whiteArea += cv2.contourArea(contours[i])
```

Next, the obtained mask is applied to original image to get (hopefully) only hand picture with no background. After that, the segmented image is converted to HSV color space and split into channels, so that histogram equalization could be applied only on saturation channel. This operation balances the color and contrast of images, which allows for using much less (in our case 3) HSV filters to obtain satisfactory results for all pictures in our data set.

```
# Apply obtained mask to original image
segmented = cv2.bitwise_and(segmented,segmented, mask=thresh)

# Split HSV values on obtained image and equalize saturation using
# equalizeHist()
H, S, V = cv2.split(cv2.cvtColor(segmented, cv2.COLOR_BGR2HSV))
eq_S = cv2.equalizeHist(S)
eq_image = cv2.cvtColor(cv2.merge([H, eq_S, V]),cv2.COLOR_HSV2RGB)
```

Figure 4: Applied masks

Figure 5: Equalized saturation channel

We can easily observe that after histogram equalization, nails appear to have higher saturation than the rest of the image. Hence, after manual testing, the following HSV ranges were obtained for filtering. Morphology closing is applied twice to remove small defects that formed on nails.

```
# Apply HSV filter to eliminate background
hsvim = cv2.cvtColor(eq_image, cv2.COLOR_BGR2HSV)
lower = np.array([105, 0, 170], dtype = "uint8")
upper = np.array([179, 65, 255], dtype = "uint8")
mask = cv2.inRange(hsvim, lower, upper)
mask = cv2.medianBlur(mask,3)

kernel = np.ones((3,3),np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
```

Figure 6: HSV filtering

Figure 7: Closing

Next step is to find all contours present on the image and save only those which area is within estimated range. The bounds for nails' sizes were calculated manually, after many trials, and are dependent on image area. This approach was chosen due to the observation that in our data set nails on small images (i.e. with area smaller than ∼460 000 pixels) are much bigger and occupy much larger part of the picture than on huge images (i.e. with area greater than 460 000 pixels).

```
# Prepare variables to find nails
contours, hierarchy = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
nails = []
minRatio = 0.00172
maxRatio = 0.0337
if area > 466000:
    minRatio = 0.0007
    maxRatio = 0.00235
# Find nails using size condition
for i in range(len(contours)):
    if cv2.contourArea(contours[i]) > minRatio*area and
    cv2.contourArea(contours[i]) <= maxRatio*area:
        nails.append(contours[i])
```

Having contours which are our estimated nails, the final step is to draw them on a black mask. Additionally, bounding boxes marking the nails are drawn on the copy of original image to see how good is the result.

```
# Draw found contours (nails) and bounding boxes for them
boxes = []
mask = np.zeros((image.shape[0], image.shape[1], 3), np.uint8)
for i in range(len(nails)):
    rect = cv2.minAreaRect(nails[i])
    box = np.int0(cv2.boxPoints(rect))
    boxes.append(box)
    cv2.drawContours(mask, [nails[i]], 0, (255,255,255), cv2.FILLED)

image_box = image.copy()
for i in range(len(boxes)):
    cv2.drawContours(image_box, [boxes[i]], -1, (0, 0, 255), 2)
mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
```

Figure 8: Mask with drawn contours

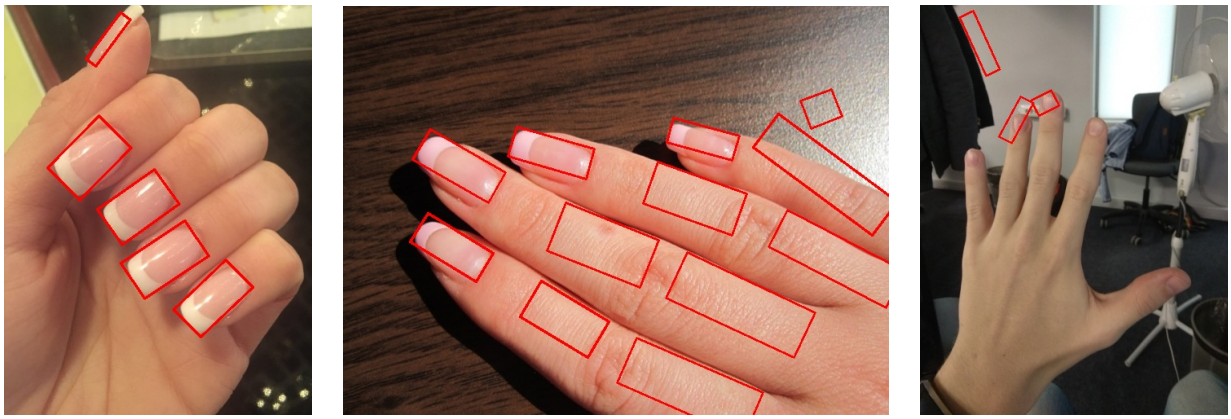


Figure 9: Bounding boxes

It is clearly visible that first picture doesn't need any further filtering - current result is really accurate and can be returned as our final mask. However, the results on second and third picture are far from satisfactory, which is why they need different ranges of HSV filter provided in the separate function. The following conditions were designed to filter out photos that need slightly different approach.

```
# Conditions to check if current solution is good enough
if len(nails) < 4 or (len(nails) > 8 and area > 60000):
    mask,image_box = detect_nails(filepath,eq_image, DEBUG)

if len(nails) >= 3 and len(boxes) <= 5 and area > 100000 and
sum(cv2.contourArea(x) > 0.02036*area for x in nails) >= 1:
    mask,image_box = detect_nails(filepath,eq_image, DEBUG)
```

```
if (len(nails) == 4 or len(nails) == 7 or len(nails) == 8) and
sum(cv2.contourArea(x) > 0.0078*area for x in nails) >= 2
and area in (196608,247000):
    mask,image_box = detect_nails(filepath,eq_image, DEBUG)

if area in (50625,654953) and 4 <= len(nails) < 8 and
sum(cv2.contourArea(x) < 0.0026*area for x in nails) >= 2:
    mask,image_box = detect_nails(filepath,eq_image, DEBUG)

return mask,image_box
```

Firstly, the number of found nails is checked - if it's too small (less than 4) or too large (more than 8), it means last filter didn't work properly. Then, if number of found nails is between 3 and 5 and there is at least one "nail" too big for being a nail, the algorithm also failed. Next, if number of nails is equal to 4,7 or 8 and there are at least 2 nails bigger than calculated part of photo area, the result is rejected. And finally, if number of nails on image with specific area was between 4 and 7 and at least two of the nails are small enough to be mistaken for background noises, current result is again declined.

In brief, if one of the above conditions is fulfilled, it means the algorithm found too many or too few objects with wrong size depending on image area. In such case, the final mask is obtained using different function, which takes the image with equalized saturation channel as starting point and applies other HSV filtering range.

Of course, presented conditions are designed especially for given data set - the areas of all images are known and the small number of input pictures allowed for careful analysis of each and every result. The purpose of such method is to maximize the compatibility of the final outcome with target masks. However, in most cases this is not a good approach. While working with bigger data sets, the requirements would have to be generalized and the algorithm would give drastically worse results.

2.2 Nails filtering

(NOTE: Code from this section can be found in nails.py and nails2.py)
As previously mentioned, function **detect_nails()** starts from image with equalized saturation channel and applies the following HSV filter (acquired by manual testing). The resulting mask is blurred a little bit to remove small noises and close shapes.

```
# Apply HSV filter to separate nails
hsvim = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
lower = np.array([116, 0, 0], dtype = "uint8")
upper = np.array([179, 130, 255], dtype = "uint8")
mask = cv2.inRange(hsvim, lower, upper)
mask = cv2.medianBlur(mask,5)
```

Figure 10: Starting images with equalized saturation

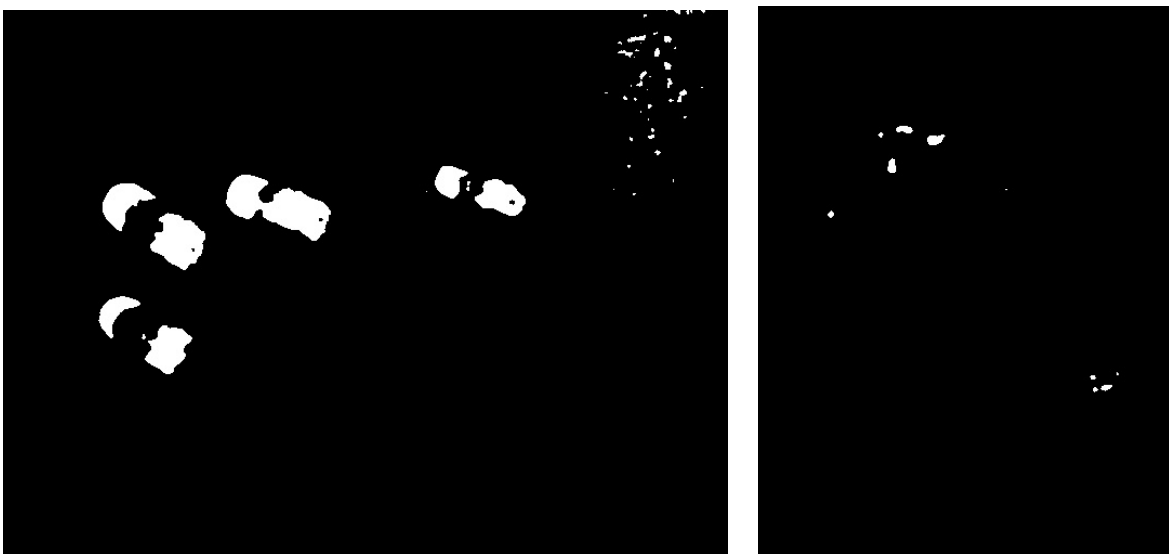


Figure 11: Masks after filtering

Morphology closing and opening eliminates slight defects that formed during filtering, clearing the mask. Then, a small dilation is applied in order to enlarge area of obtained nails, because in most cases the mask crops considerable part of their surface.

```
# Morphology operations to remove background noises and close contours
kernel = np.ones((7,7),np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

# Dilate the obtained mask to enlarge area of nails
kernel = np.ones((3,3),np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)
mask = cv2.medianBlur(mask,3)
```

Figure 12: Morphology operations

Figure 13: Dilation

Then, similarly to the process described in *Section 2.1*, all contours on our mask are filtered according to their size. Again, the limits are dependent on image area and were calculated manually to give the most accurate results possible.

```
# Prepare variables and find nails using size condition
contours, hierarchy = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
area = image.shape[0]*image.shape[1]
nails = []
if area > 500000:
    minRatio = 0.0007
    maxRatio = 0.0052
if 465500 < area <= 500000:
    minRatio = 0.0014
    maxRatio = 0.012
else:
    minRatio = 0.0014
    maxRatio = 0.028
for i in range(len(contours)):
    if cv2.contourArea(contours[i]) > minRatio*area and
    cv2.contourArea(contours[i]) <= maxRatio*area:
        nails.append(contours[i])
```

After that, if the outcome is still not sufficient, i.e. the number of nails found is less than 3 or there are at least 2 nails with area too small for being a nail, it means the current filter was too 'strong' and cleared almost everything from the picture. If there are more than 10 nails, the result is also incorrect. In both cases, yet another HSV range needs to be applied and the initial image is passed to the third method with slightly different HSV bounds.
Otherwise, the result is considered satisfactory, hence the final contours and bounding boxes are drawn identically as in previous section and returned.

```
# If there are too few nails, apply different HSV filter
if len(nails) < 3 or (sum(cv2.contourArea(x) < 0.00177*area
for x in nails) >= 2 and area < 60000) or len(nails)  > 10:
    mask,image_box = detect_nails2(filepath,image, DEBUG)
    return mask, image_box
```

Considering our exemplary photos, the mask obtained for the second one is quite accurate and is, therefore, returned as the final result.

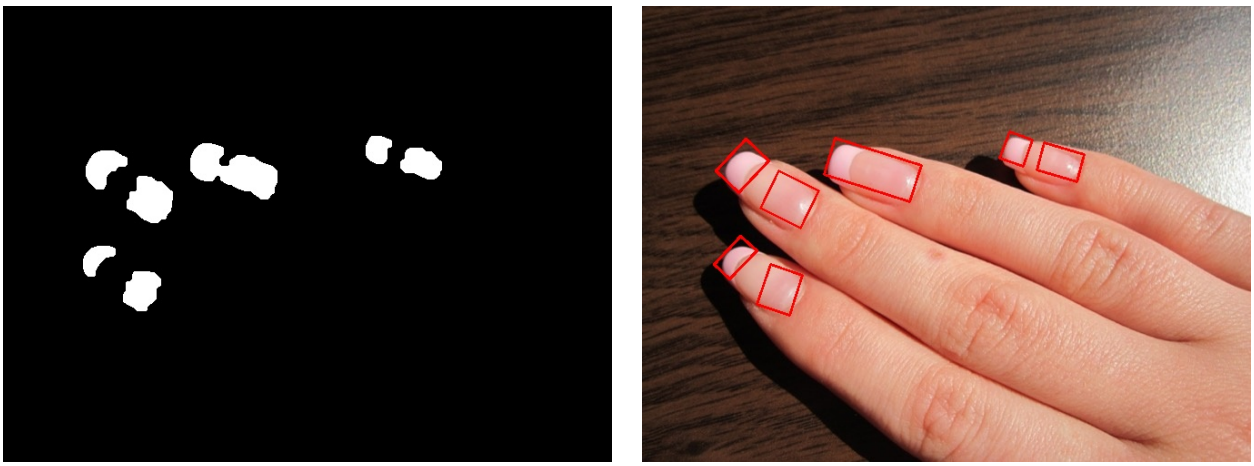


Figure 14: Final output

However, the third mask after filtering shows almost nothing, which means the filter was too strong. The image is passed to the third function **detect_nails2()** and processed further.

Figure 15: Second mask and starting image for **detect_nails2()**

The function **detect_nails2()** is almost identical to **detect_nails()** with the difference of HSV range applied for starting image. As we may observe, the lower bound for hue channel is a little smaller, since after manual testing it proved to be the most optimal solution for our data set. Next, morphology closing and opening remove small noises from image and slight dilation enlarges a little area of obtained nails.

```
# Apply HSV filter to filter out nails
hsvim = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
lower = np.array([112, 0, 0], dtype = "uint8")
upper = np.array([179, 130, 255], dtype = "uint8")
mask = cv2.inRange(hsvim, lower, upper)
mask = cv2.medianBlur(mask,5)

# Morphology operations to remove noises
kernel = np.ones((7,7),np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

# Small dilation to enlarge area of nails
kernel = np.ones((3,3),np.uint8)
mask = cv2.dilate(mask, kernel, iterations=1)
mask = cv2.medianBlur(mask,3)
```

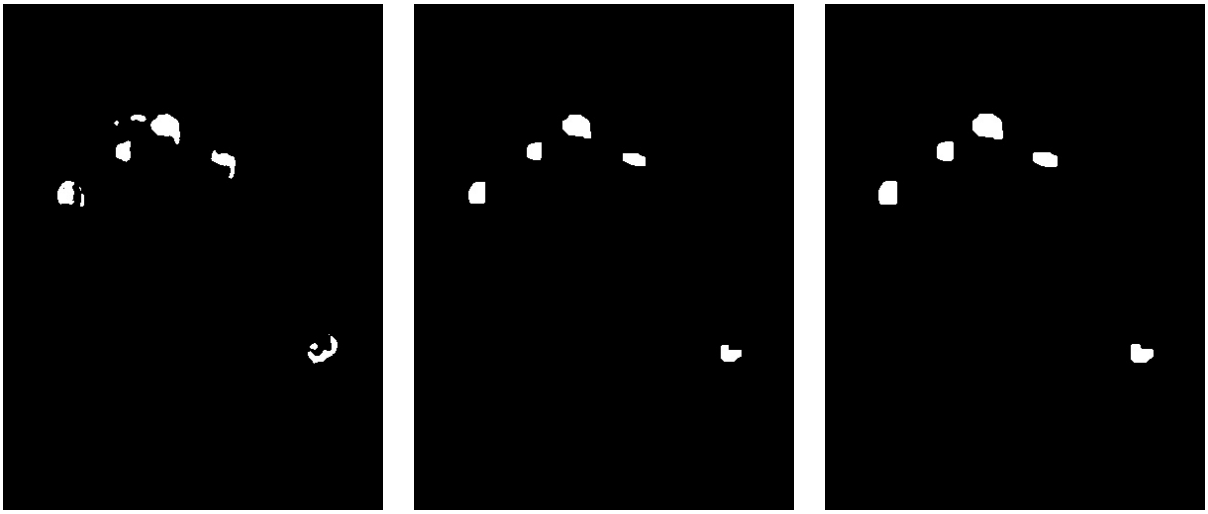


Figure 16: Initial mask, closing and dilation

Clearly, the mask obtained after third HSV filtering is much more accurate. Even though the mask doesn't cover the whole area of true nails, it is really hard to achieve higher precision. It could be possible if every image would have been processed individually, but generally current result is definitely satisfactory and the method ends.

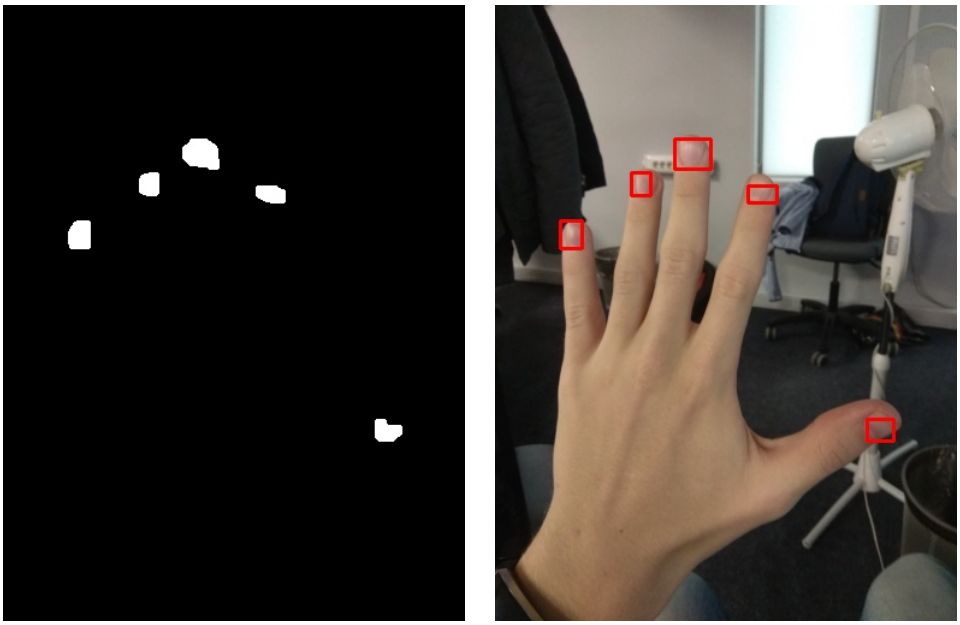


Figure 17: Final output

2.3 Special cases

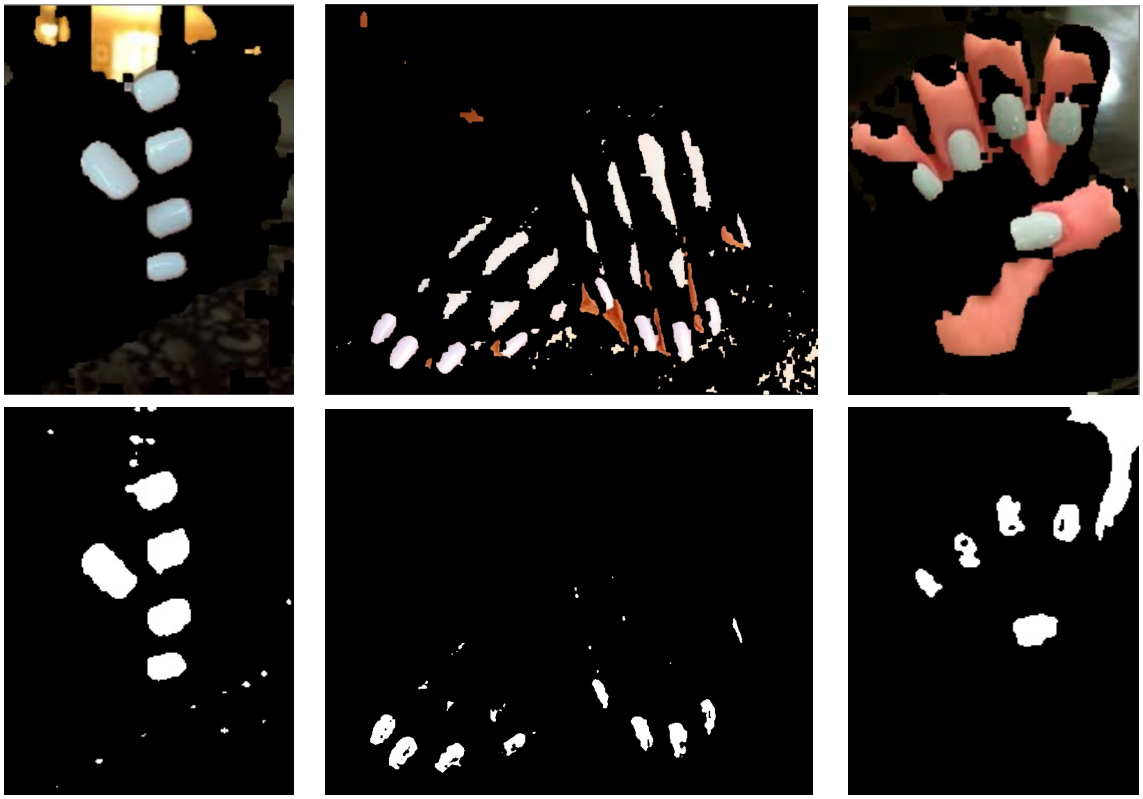
(NOTE: Code from this section can be found in nails3.py)
Now, let's consider the following photos and their masks obtained after first YCrCb skin detection described in *Section 2.1*. It is apparent that, since nails are painted with distinct colors very different from skin color, the filter clears almost everything but nails from the picture.

In those particular cases it is much better to reverse the obtained mask before starting any further processing. That is why the following conditions were designed to identify the 'special' images. They utilize calculated area of skin mask and image area, which is definitely not a good approach in more general cases. After the inversion, the mask is applied to original picture which is then passed to function **detect_nails3()** as starting point (there is no need for histogram equalization in this case). Next, the resulting mask from **detect_nails3()** is returned immediately, since any further operations performed on the image would only corrupt the outcome.

```
# Reversed mask and different function for nails segmentation
if whiteArea > 0.91*area or (area == 50246 and whiteArea in
(33015,36175,38124.5)):
    thresh = 255 - thresh
    segmented = cv2.bitwise_and(segmented,segmented, mask=thresh)
    mask, image_box = detect_nails3(filepath, segmented, DEBUG)
    return mask, image_box
```

Once more, the presented conditions are not proper for general solution and only serve the purpose to detect exactly 4 'special' images present in our data set. Thanks to that, the accuracy of final masks is increased considerably. Then, the function **detect_nails3()** applies the following HSV filter (also acquired by manual testing) on the partially segmented images.

```
# Apply special HSV filter to remove everything but nails
hsvim = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
lower = np.array([40, 0, 0], dtype = "uint8")
upper = np.array([179, 66, 255], dtype = "uint8")
mask = cv2.inRange(hsvim, lower, upper)
mask = cv2.medianBlur(mask,3)
```



It is clearly visible that after this operation the masks are almost ready. Next, morphology operations remove small noises left in the background.

```
# Morphology operations to remove noises and enlarge area of nails
kernel = np.ones((5,5),np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
mask = cv2.dilate(mask, kernel, iterations=1)
mask = cv2.medianBlur(mask,3)
```

Remaining contours are sorted according to their size dependent on image area - the ratio coefficients are slightly different and adjusted for the 'special' pictures.

```
# Find nails
contours, hierarchy = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
area = image.shape[0]*image.shape[1]
nails = []
minRatio = 0.004
maxRatio = 0.028
if area > 300000:
    minRatio = 0.0017
    maxRatio = 0.02
# Find nails using size condition
for i in range(len(contours)):
    if cv2.contourArea(contours[i]) > minRatio*area and
    v2.contourArea(contours[i]) <= maxRatio*area:
        nails.append(contours[i])
```

Finally, the acquired nails and bounding boxes are drawn identically as in previous cases and the method ends.

3 Results

3.1 Results assessment

The assessment criteria was the Intersection over Union metric and Dice coefficient.

Even after precise analysis of each picture the results are barely acceptable - the mean score for IoU was 39.32% while the mean Dice coefficient was 52.43%.

Looking at the highest scores: IoU - 79.52% and Dice - 87.085% we can observe that the algorithm works best when the nails aren't natural (e.g. painted in pink) or the hand is easily distinguishable from the background.

Figure 18: Image with best percentage result (869CDA2E-8251-4880-89D6-9409CBC416F3.jpg)

However, only few masks achieved such high accuracy. In most cases the algorithm failed to detect all nails or to clear some background noises. There are also images where parts of the skin are mistakenly perceived as nails because of the lighting.

Figure 19: Images with imperfect masks

3.2 Failed cases

As previously mentioned, the algorithm was designed specifically for given data set so that it would work for the greatest number of pictures possible. However, there are still images where the segmentation fails completely and the output is an empty mask or nails are missed entirely.

Figure 20: Images for which the algorithm failed

The reason for such behaviour is because on part of the images the skin detection doesn't work properly. On third and fourth photo, the YCrCb filtering detected background instead of the hand or separated the hand completely excluding nails. In such cases, it is not a surprise that further processing doesn't lead to any acceptable result.

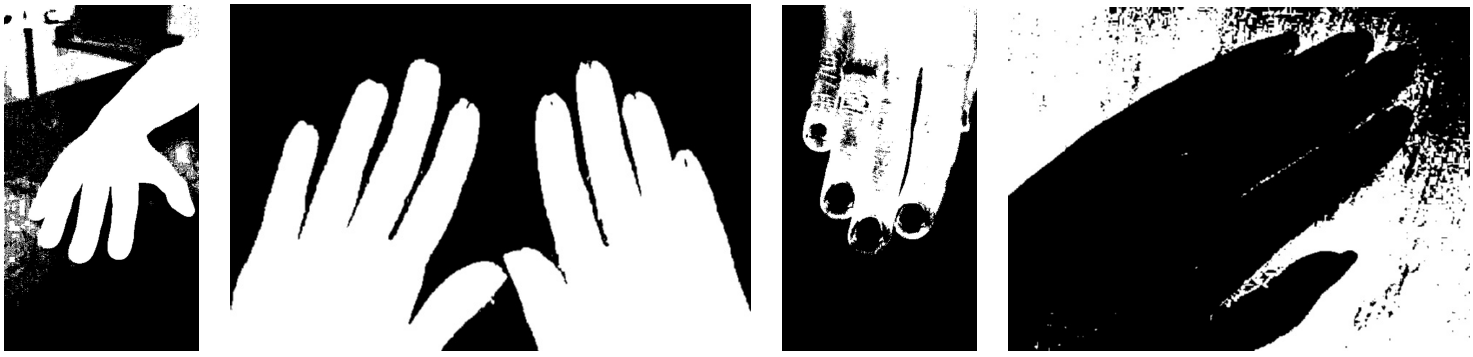


Figure 21: Masks after first skin detection

As shown above, there are also cases where skin detection gives quite correct masks. Nevertheless, none of the presented HSV ranges is adequate for those particular images. On the first picture nails are marked in correct places but they are too small and, therefore, ignored. On the second image there is another problem - after histogram equalization fingers and nails appear to be much darker and have much lower saturation than the rest of the hand. This is a completely opposite effect to majority of photos and contradicts assumptions of the algorithm which is why the method fails.

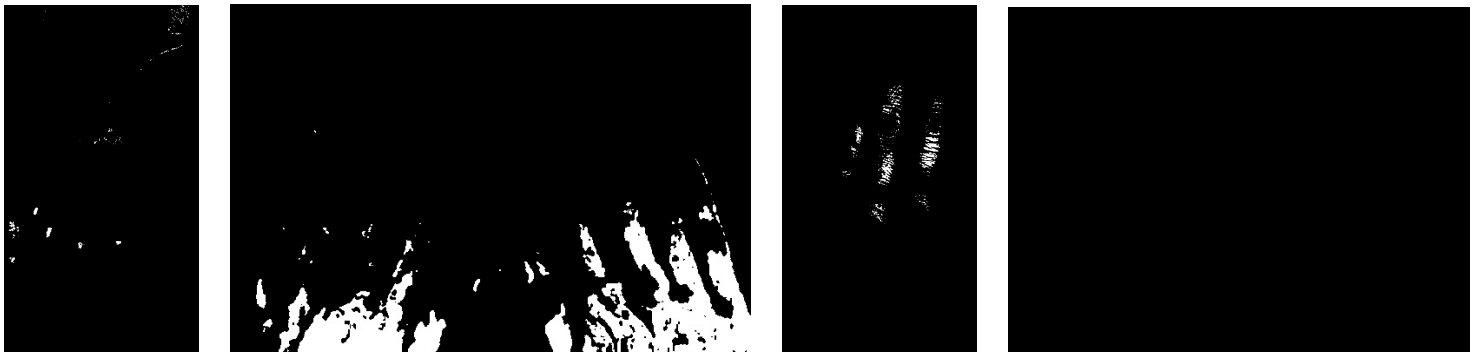


Figure 22: Masks after first HSV filtering

Figure 23: Final failed masks

4 Conclusion

Looking at achieved results, we can draw the conclusion that color based image processing is not a good method for segmenting human nails from photos. The algorithm described above was especially customized for given data set, so that it would work for the greatest number of images possible and yet it didn't even reach 50% of correctness compared to desired results. Due to the fact that natural color of human nails is very similar to skin color, any change in the lighting of the photo can completely ruin the result. Moreover, human nails can be painted in a large variety of flashy colors (e.g blue or red) which requires a whole new solution for filtering them out. Not to mention that tones of human skin may also greatly differ depending on one's origin. Therefore, it is extremely hard, if not impossible, to write a universal color based image processing algorithm for human nails segmentation.

References

[1] https://en.wikipedia.org/wiki/Image_segmentation#Edge_detection

[2] https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html

[3] https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html

[4] https://lmcaraig.com/understanding-image-histograms-with-opencv

[5] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_trackbar/py_trackbar.html