Computer Vision Project 2021 - 'Fruit Detection'

Marco Aspromonte
(marco.aspromonte@studio.unibo.it)

Enrico Morselli
(enrico.morselli@studio.unibo.it)

Abstract: In the following report we are describing the work done on fruit's images ,in order to analyze the fruits and detects some features such as edges and defects of the fruits. This is achieved by the usage of some important python libraries such as numpy and opecv and the implementation of some useful functions. The chosen developement enviroment is Google Colab, but also the project was tested on the IDE PyCharm due to some Colab's limits (nothing relevant by the way). The Colab Notebook is divided in three main tasks, the first is about detect the stronger defects of the apples through binarization and edge detection, the second task is about russet detection using the Mahalanobis distance and the third concerns kiwi detection by deploying object detection using template matching.

Keywords: Edge Detection, Binary Mask, Segmentation, Binarization, Mahalanobis Distance, Object Detection, Template Matching, OpenCV, Python, Colab

1 Fruit segmentation and edge detection - first task

1.1 Binarization

The first thing we need to do is to binarize the grayscale image, so performing a background/foreground segmentation, in order to then generate the binary mask. The method we used for binarization is Otsu's Algorithm. With Otsu's we don't choose a fixed value for the threshold, but instead the algorithm determines it automatically. Consider an image with only two distinct image values (bimodal image), where the histogram only consists of two peaks. A good threshold is in the middle of those two values. Similarly, Otsu's method determines the optimal threshold value from the image histogram. In order to do so, we use the `cv.threshold` function, passing `cv.THRESH_BINARY+cv.THRESH_OTSU` as a flag. We do not have to pass a threshold value, because it will be computed automatically based on the given image.

1.2 Generation of the binary mask

After the binarization of the image the foreground (apple) is labelled with white pixels, but the defects are instead marked as background (they are basically false negatives). So in the foregound we see holes which are the defects of the fruit, so the mask is generated applying a dilation operation to the binarized image, in order to fullfill the holes:

```
def generate_binarymask(img):
  _,img_closing = cv.threshold(img,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
  kernel = np.ones((3,3), np.uint8)
  img_dilate = cv.dilate(img_closing, kernel, iterations=9)
return img_dilate
```

Then we can see the effect of the function on the binarized images: Now that

Figure 1: Mask 1

Figure 2: Mask 2

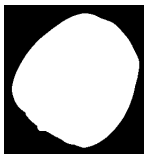


Figure 3: Mask 3

the mask are generated, we can apply them on both color and gray scale image, in order to obtain only the fruit blob inside the region. Here is the function that implements an AND operator between the mask and the image considered:

```
def apply_mask(img_orig, mask):
  result = cv.bitwise_and(img_orig,mask)
  result[mask==0] = 255
return result
```

Now we apply the binarized mask on the image by performing a `bitwise_and` operation. As we can see in the resulting image, along with the apple we have also a portion of the background. That's because we applied Dilation on the segmented image, and so we have expanded the shape of the object. We could also have used a Closing operation (which is a Dilation followed by an Erosion), which can be though of as a "smart" Dilation, and so avoid this effect. By the way we have seen that Closing wasn't really that much effective in filling the hole, unless assinging a kernel size for dilation and erosion, and a number of iterations specifically for each sample image. Then, we decided to go on with a more general approach just applying a Dilation with the same parameters to all our images. So, we will deal with the additional background in the following step, which is Edge Detection.

1.3 Edge Detection

Now, its time for performing Edge Detection, for which we will use Canny. In order to cancel out the background, we basically apply Canny with two different

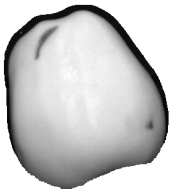


Figure 4: Mask 1

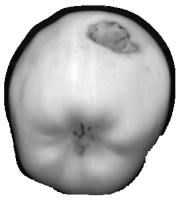


Figure 5: Mask 2

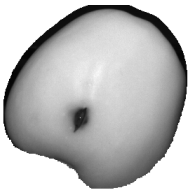


Figure 6: Mask 3

thresholds, producing two different images, `edges_high_thresh` where we find only the outer contours of the apple and `edges_normal` where we also find the contours of the defects in the apple. The difference between the two edge maps will provide us an edge map showing only eventual edge points inside the apple which we identify as defects.

```
def find_edges(t1,t2,n1,n2)
    edges_high_thresh = cv.Canny(mask_ongray, t1, t2)
    edges_normal = cv.Canny(mask_ongray,n1,n2)
    segmentation =  edges_normal - edges_high_thresh
return edges
```

And we obtain the following edges results. The advantage of this method is that

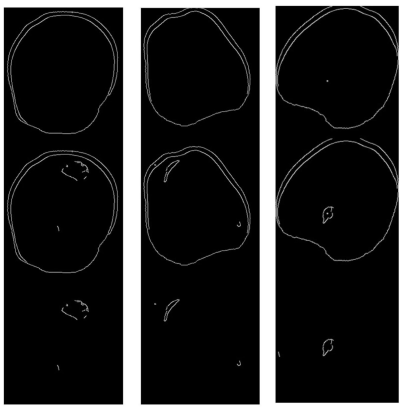
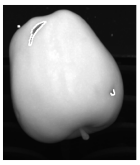
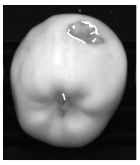
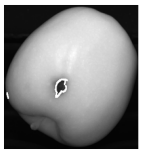


Figure 7: Caption

we can get a good evaluation of edges , even if the mask does not perfectly fit to the fruit the excess of the contours is removed with the previous function. The edges are drawn by a simple function that replace the pixels on the original grayscale image (We will apply a dilation step on the edges in order to expand the contours):

```
def color_edges(edges,img):
  for i in range(0,img.shape[0]):
    for j in range(0,img.shape[1]):
      if edges[i,j]==0:
        continue;
      else:
        img[i,j] = edges[i,j]
  kernel = np.ones((3,3), np.uint8)
  img=cv.dilate(img,kernel,iterations=1)
return img
```

So we can obtain the original images with the detected edges!

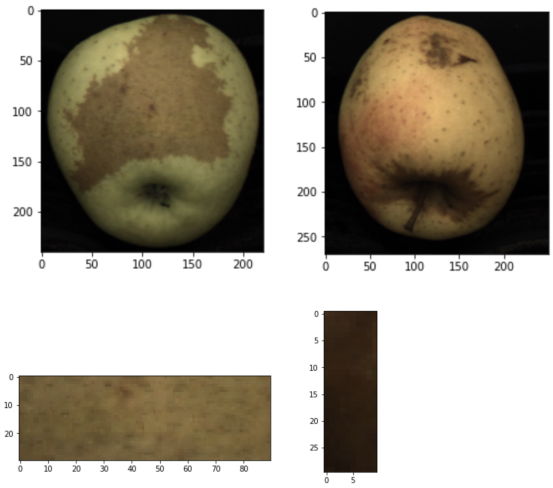




2 Russet Segmentation - Second Task

The second task consists in divide out of the 'good apple' all the regions that are defined as 'russet', in order to detect the defects of the apple based on colors. The first approach used to detect the russet consists in finding a suitable Mahalanobis distance in order to employ segmentation by colors. We will work in the standard RGB space, by the way, also other spaces were tested, in particular HSV space, and the results, except for the threshold setting, were similar.

So first we will import the target images, and we will select a region of interest in each image, which will be our training samples. Here we can see the two images used for this task

As training image we will select a russet region in the apple. Then, the reference color will be computed based on the russet.



This is the core of the second task. As suggested by the Hint in the trace of the project, we implemented the `mahalanobis` function, to compute the Mahalanobis distance.

The Mahalanobis distance is defined as

$$d_M(I(p), \mu) = ((I(p) - \mu)^\top \Sigma^{-1} (I(p) - \mu))^{\frac{1}{2}}$$

where $I(p)$ represents the RGB values of the given pixel p, μ is the reference color, i.e. the mean of each of the RGB channels of the training image, and Σ^{-1} is the inverse covariance matrix of the RGB channels of the training image. If we explode this formula, we get

$$d_M(I(p), \mu) = \left(\frac{(I_r(p) - \mu_r)^2}{\sigma_{rr}^2} + \frac{(I_g(p) - \mu_g)^2}{\sigma_{gg}^2} + \frac{(I_b(p) - \mu_b)^2}{\sigma_{bb}^2} \right)^{\frac{1}{2}}$$

Where $\sigma_{rr}^2, \sigma_{gg}^2$ and σ_{bb}^2 are the variances of the Red, Green and Blue channel respectively. Then, we set a threshold T, and all the pixels p where $d_M(I(p), \mu) \leq T^2$ are classified as russet.

In the basic formulation, we suppose this matrix to be diagonal, so basically, we assume all colors to be uncorrelated:

$$\Sigma = \begin{pmatrix} \sigma_{rr}^2 & 0 & 0 \\ 0 & \sigma_{gg}^2 & 0 \\ 0 & 0 & \sigma_{bb}^2 \end{pmatrix}$$

By the way, the covariance matrix is real and symmetric, so it can always be diagonalized. More precisely, we can apply the Eigenvalue decomposition, and write the matrix Σ as:

$$\Sigma = RDR^\top, \qquad R = (\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3), \quad D = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}$$

Where R is an orthogonal rotation matrix, where $\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3$ are the eigenvectors of Σ and the elements λ_i of the matrix D are the eigenvalues of Σ. Then, the formula for the Mahalanobis becomes the following:

$$d_M(I(p), \mu) = ((I(p) - \mu)^\top R D^{-1} R^\top (I(p) - \mu))^{\frac{1}{2}}$$

In the end, we get:

$$d_M(I(p), \mu) = \left(\frac{(I'_r(p) - \mu'_r)^2}{\lambda_1} + \frac{(I'_g(p) - \mu'_g)^2}{\lambda_2} + \frac{(I'_b(p) - \mu'_b)^2}{\lambda_3} \right)^{\frac{1}{2}}$$

were $I'_r(p) - \mu'_r, I'_g(p) - \mu'_g, I'_b(p) - \mu'_b$ are obtained by the dot product between the vector $(I(p) - \mu)$ and the matrix R. Basically, we are rotating our original data by the rotation matrix R, and we obtain a new reference systems with the axes orientated as the eigenvectors of the matrix Σ, in which the covariance matrix is diagonal, and the variances are exactly the eigenvalues of Σ.

We will implement this in the 'mahalanobis' function. First we select the values for each color channel of the training image `img_train`, and we store them into three vectors. We then construct:

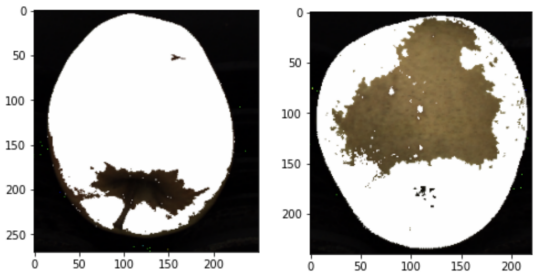
1. The mean vector containing the mean value associated to each color channel;
2. The covariance matrix, by `np.cov`, flattening the color channel vectors with `ravel` function of numpy;

Then, we compute the inverse of the covariance matrix, and the eigenvalues and eigenvectors of the inverse covariance matrix.

```
def mahalanobis(img,img_train,threshold):
    red = img_train[:,:,0] #red
    green = img_train[:,:,1] #green
    blue = img_train[:,:,2] #blue

    mean = np.array([red.mean(),green.mean(),blue.mean()])
    covariance = np.cov([red.ravel(), green.ravel(), blue.ravel()])
    inv_cov = np.linalg.inv(covariance)
    eigval,eigvec = cv.eigenNonSymmetric(inv_cov)

    eigvec=np.array(eigvec)
    img=np.array(img)

    for i in range(0,img.shape[0]):
        for j in range(0,img.shape[1]):
            img_prime= np.dot(img[i,j,:]-mean,eigvec)

            if ((img_prime[0])**2/eigval[0]
```

```
            +  (img_prime[1])**2/eigval[1]
            +  (img_prime[2])**2/eigval[2]) <= threshold**2 :
              img[i,j] = 0
return img
```

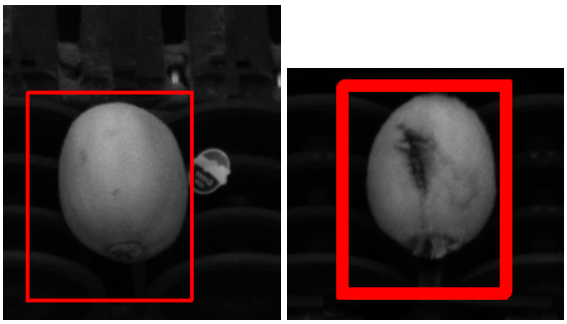
The result given by the mahalanobis function for the first image is quite good, but for the second the russet is not too strong in terms of colors intensity, so it's more difficult to find a sensible change of signal between the russet and the apple.



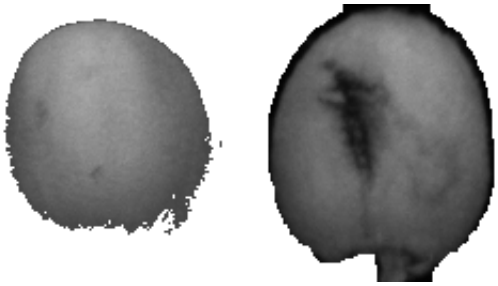
3 Kiwi Detection - Third Task

For the Third task, in order to segment only the kiwi fruit and keep out the sticker of the sixth sample as mandatory, it is useful to isolate the region of interest in wich the kiwi fruit belongs. The technique that we decide to employ belongs to Template Matching theory, and opencv has some useful functions that develop some of the Dis(Similarity) functions. In particular, because we are taking a kiwi image in training (the 10th sample) that is the most clear image between the samples. In order to compute the matching between the training and the target, we use the opencv `matchTemplate` function, that takes as parameters the target, the training and the chosen similarity function. Here we choose the Normalized Cross Correlation, which is applyied by specifying the flag `cv.TM_COEFF_NORMED`, because we have differences in terms of intensities between the training image and the target image. In fact, the NCC it is more robust with respect to noise and intensity changes between images (more specifically it is invariant to linear intensity changes). Then a threshold is set in order to detect 'how much' kiwies are similar and a rectangle is drawn in correspondence of the coordinates of the rectangle.

```
def kiwi_detection(img_rgb,template):
    img_gray = cv.cvtColor(img_rgb, cv.COLOR_BGR2GRAY)
    w, h = kiwi_template.shape[::-1]
    res = cv.matchTemplate(img_gray,template,cv.TM_CCOEFF_NORMED)
    threshold = 0.79
    loc = np.where( res >= threshold)
    for pt in zip(*loc[::-1]):
        cv.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), (0,0,255), 1)
    rectangle=img_rgb[pt[1]:pt[1]+w,pt[0]:pt[0]+w]
    return img_rgb,rectangle
```



After that, we can do a more conveniently binarization of the image only considering the region of interest containing the kiwi, given by `rectangle` in `kiwi_detection` function, and keep out most of the other elements inside the target image. Then we can apply the mask on the region of interest given by `rectangle` and extract only the fruit by calling the `apply_mask` function of the first task.



As we have seen in the first task, is better to apply a dilation for dealing with false negatives, so for the 7th sample that have a strong defect, the flag of dilation is set on True, while for the 6th is set on False. Then, in order to find the defect of the 7th sample, we apply the `find_edges` function of the first task, by tuning other values of the thresholds

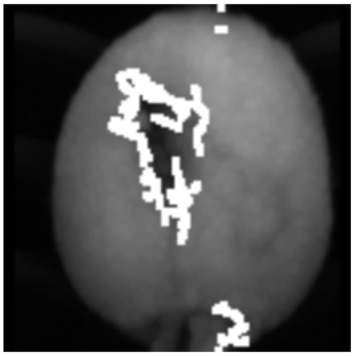


Figure 8: Defect detection on kiwi

References

[Luigi di Stefano] Image Processing and computer vision slides.

[OpenCV] Documentation on https://opencv.org

[Numpy] Documentation on https://numpy.org