Fruit inspection

Computer Vision Project Work Report

Vincenzo Collura

University of Bologna
Italy
September 2022

Contents

1 Introduction

The project I have chosen is *Fruit inspection* whose goal is to create, given the image of the fruits, such a system should be able to isolate the fruit and recognize defects, russets, their size, and their position. The project is composed of three different tasks which will explain the development and the strategies used in their resolution. My code together with the images can be found here.

1.1 Overview of the setup

The project was implemented thanks to the `OpenCV` library [1]. The core of the setup is in the `utils.py` file. Everything was done and explained in the notebook `fruit-inspection.ipynb`.

2 First task: fruit segmentation and defect detection

There are two main goals:

- Outline the fruit by generating a binary mask;
- Search for the defects on each fruit.

The two structures can be seen in figure 1 and 2.

2.1 Brief description

2.1.1 Outline the fruit by generating a binary mask

To solve this sub-task I first threshold the whole image to remove the background, but keeping intact the fruit borders as suggested by the project assignment itself, then I label the connected components to create the mask using the bigger (The image pair has little parallax, so a mask might be computed on one image and then applied on the other one). After that, I fill the holes inside the fruit blob using a flood-fill approach and at the end, I apply the resulting mask to the image to outline the fruit.

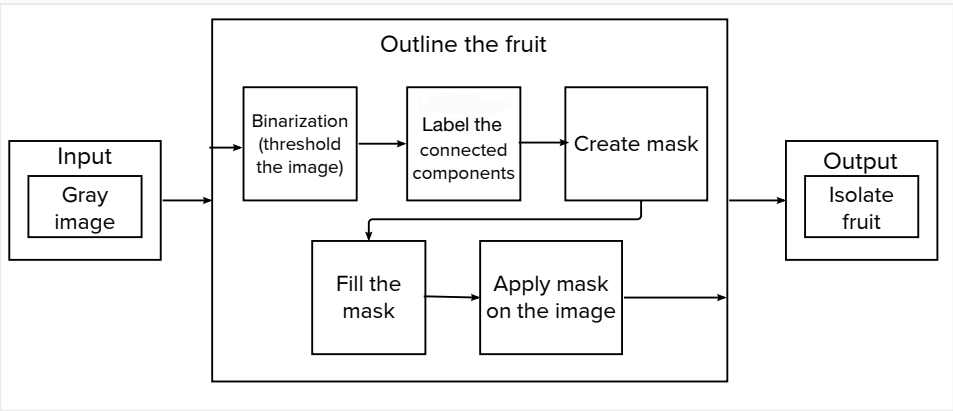

Figure 1: Outline the fruits scheme.

2.1.2 Search for the defects on each fruit

To solve this sub-task I first extract edges using an edge detector (`Canny`), because the defects have strong edges, then I label the connected components in order to create the masks of the defects. After that, I fill the holes inside the defects blob using a flood-fill approach and at the end, I use the masks to compute the corresponding bounding boxes and apply them to the images.

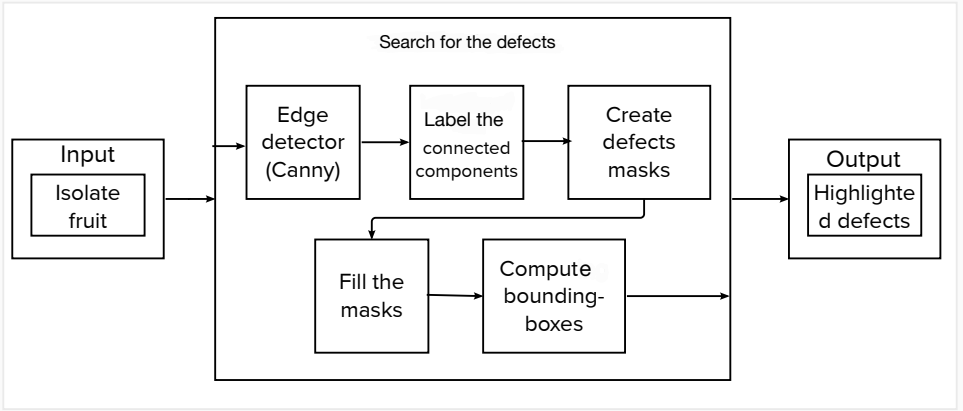

Figure 2: Search for the defects scheme.

2.2 Computation

2.2.1 Threshold the whole image (binarization)

I threshold the image using the `threshold` function of `OpenCV` where the grayscale image is passed in input. What's happening is that, for every pixel, the same threshold value is applied. If the pixel value is smaller than the threshold, it is set to 0, otherwise, it is set to a maximum value. `OpenCV` provides different types of thresholding but basic thresholding is used for this first task (`cv.THRESH_BINARY`) and the chosen threshold is 38.4.

2.2.2 Labeling of the connected components

The connected components are computed and labeled from the images using the `connectedComponentsWithStats` function of `OpenCV`.

2.2.3 Create the mask

Here, given the connected components, I use the "`create_mask`" and "`create_defects_mask`" functions (that you can find in the `utils.py` file) to create the masks, the first for "Outline the fruit" part, and the second for the "Search for the defects" part of the task. For the first part I just take the connected component that has the biggest area, because it corresponds with the background of the images. While in the second part, I use two thresholds in order to cut out the defects too small with the lower, because they can be due to noise, and cut out with the upper threshold the masks of fruits and of the backgrounds.

2.2.4 Fill the mask

I fill the holes inside the fruit blob using a flood-fill approach, thanks to the `floodFill` function of `OpenCV`, in order to fill the holes into the masks to delete the imperfections.

2.2.5 Apply mask on the image

Just I apply the mask to the image using the `bitwise_and` function of `OpenCV`, in order to isolate the fruit.

2.2.6 Edge detector (Canny)

Here I extract edges using the Canny edge detector because the defects have strong edges. I computed them with the `Canny` function of `OpenCV`, where the lower and upper threshold value in hysteresis thresholding was respectively 70 and 200.

2.2.7 Computing bounding-boxes

Given the defects masks, I first find the contours of each mask (`cv.findContours`) in order to compute the area of the defects (`cv.minAreaRect`), Then I compute the bounding-box points, thanks to this area (`cv.boxPoints`) and at the end draw the contours over the image connecting the points (`cv.drawContours`). Also, I check if there is overlapping between the bbox, to cut multiple bbox for the same defects. All this procedure is in the `get_bbox` function of `utils.py` file.

3 Second task: russet detection

The goal is to identify the russet or at least some part of it with no false positive areas. The structure can be seen in figure 3.

3.1 Brief description

In this second task, I use the same procedure as the first one to isolate the fruits (Fig. 1), but in this case, the threshold to binarize the image is 70.
The rest of the procedure is different, first I compute the Mahalanobis distance between each pixel of the images and some sample of the color of the russets (all in the "lab" color space), highlighting the pixel that has distance under a certain threshold. Then I use an edge detector in order to define the edge of the russets zones. Then I use a "voting system" to delete the false positive pixels, also eroding them to divide some causal link between different zones. As usual, having some "blobs", I create masks for each of them and then fill them. At the end I compute the b-box for each mask, highlighting russets.

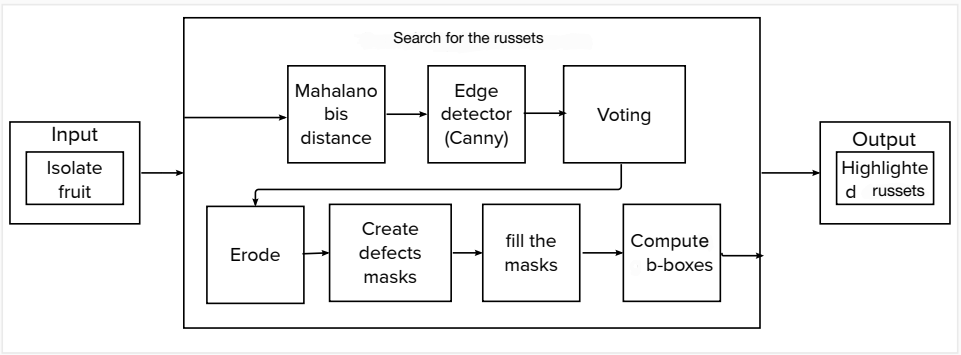

Figure 3: Search for the russets scheme.

3.2 Computation

Edge detector, create russets masks, fill the masks and computing bboxes are explained in the previous task, respectively in 2.2.6, 2.2.3, 2.2.4, and 2.2.7 sections.

3.2.1 Mahalanobis distance

Given the isolated fruit image and a list of samples of different russets, I just compute the Mahalanobis distance between the color of each pixel of the image and each sample, if this distance is lower than a threshold (1.8) the pixel is part of russet, and its color is set to blank to highlight it for the edge detector step. This approach produces a different output image for each sample that we use. The distance is computed with the function `cdist` of `scipy`, calculating first the inverse of the covariance matrix and the mean of the sample's color, that is needed to compute the Mahalanobis distance. This procedure is implemented in the function `mahalanobis_distance_with_samples` of the `utils.py` file.

3.2.2 Voting

Given the edge of each russet, we need to understand which of these is true or not. In order to do this, I have initialized a matrix of zeros of the dimension of the image, so each element of the matrix corresponds with the pixel of the image at the same position. Then for each edge image and for each pixel of that image, I increment the corresponding elements of the matrix that I create before (vote). At the end when each pixel of each image ended to vote, I can cut the false positive, so the pixel that has few votes, thanks to a threshold (2 in this case). Voting is implemented in the function `voting` of the `utils.py` file. Example in figure 4.

3.2.3 Erode

I use `erode` function of `OpenCV` in order to divide some causal links between different zones.

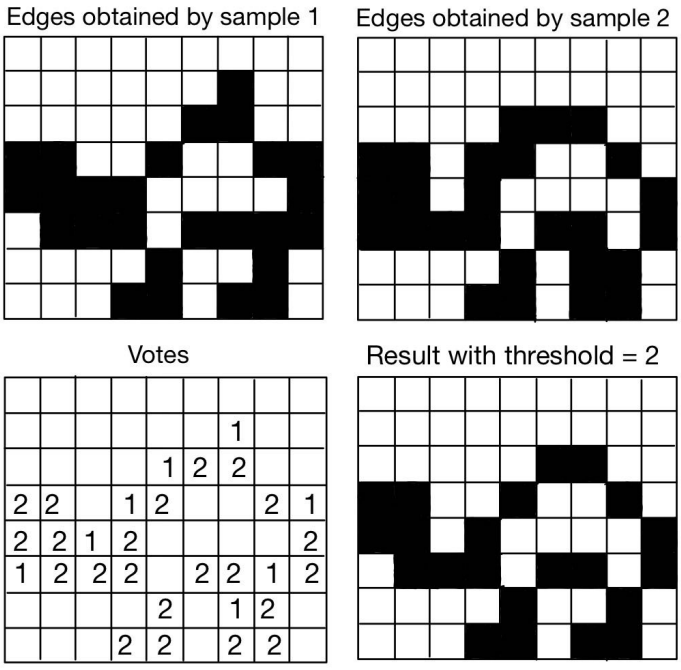

Figure 4: Voting example

4 Final challenge: kiwi inspection

Segment the fruits and locate the defect in image “000007”. Special care should be taken to remove as “background” the dirt on the conveyor as well as the sticker in image “000006”.

4.1 Brief description

To solve this task, I used the same procedure as the initial part of the first task to isolate the fruit, the only difference is in the use of the OTSU (cv.THRESH OTSU) algorithm to set the threshold, instead of a fixed one. To detect the defects I also used the second part of the first task, only to decrease the minimum and the maximum area threshold for creating masks, because the kiwi area is smaller in the images with respect to the apples one, so also the defects are smaller.

5 Results

5.1 First task

5.1.1 Outline the fruit by generating a binary mask

The results of this task are excellent, the fruits have been outlined almost to perfection. The only small problem lies in the area near the petiole, in which a few pixels are missing (how is visible in figure 5), since the color of that area is very close to the background color, it is very difficult to solve it without compromise the edges of the fruit. The final results in figure 6.

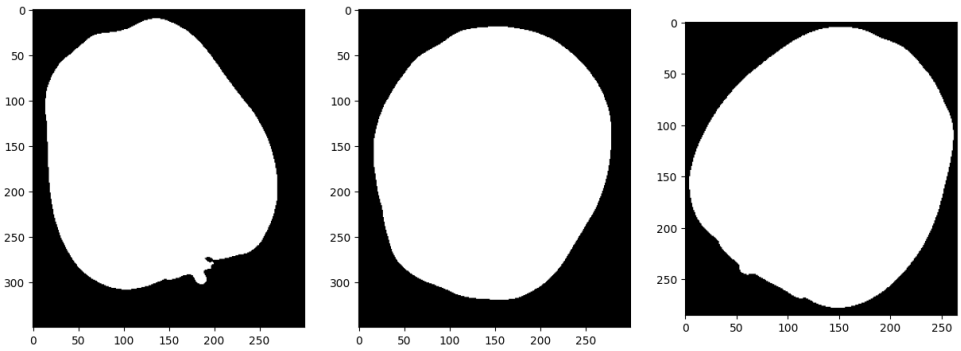

Figure 5: Final masks.

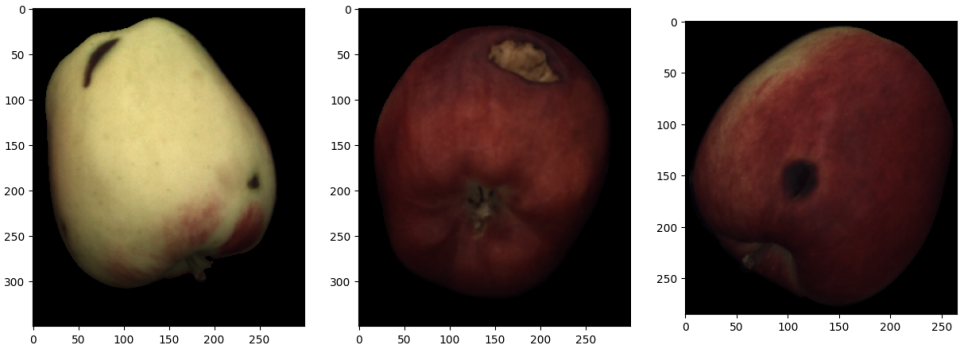

Figure 6: Outlined fruits.

5.1.2 Search for the defects on each fruit

Also in this part of the task, the results are good, with an error on the second, due to the shade caused by the sepals on the fruit, which has a color very similar to that of the defects. Even the area of that area is not negligible, indeed it turns out to be larger than the smallest defect found in the first fruit. Because of these reasons error arises. Results in figure 7.

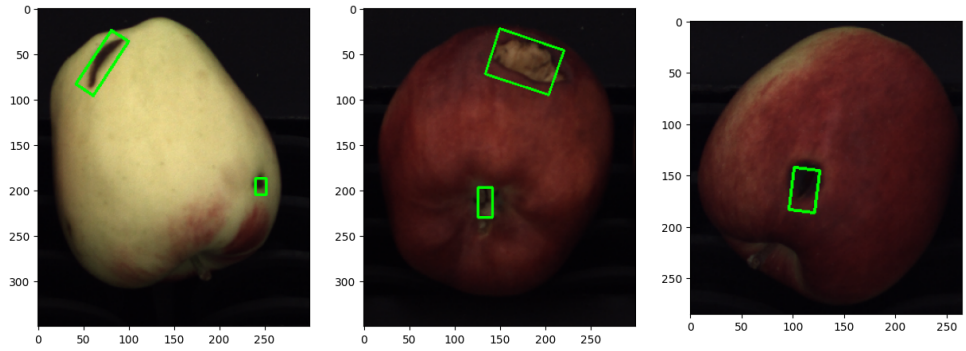

Figure 7: Defects highlighted.

5.2 Second task

Surprisingly, the results of the second task are better than those of the first. All is good, except for some imperfections in the orientation and the bounding-box area that leave some areas that are part of the russets uncovered. Some of these areas were, in fact, recognized as russets but subsequently eliminated due to the area being too small or due to the overlap with larger b-boxes. Results in figure 8.

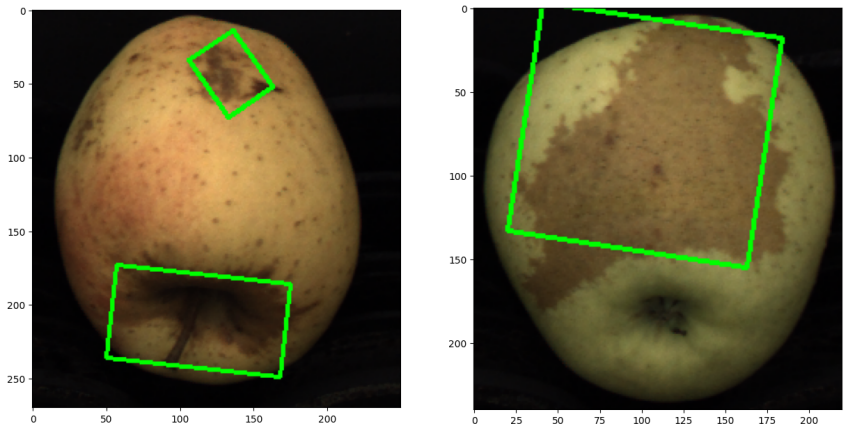

Figure 8: Russets highlighted.

5.3 Final challenge

As for the final challenge, everything was perfect, with no mistakes. Only a few imperfections in the calculation of the mask that lets a small piece of dirt pass in the part closest to the fruit. Results are in figures 9 and 10.

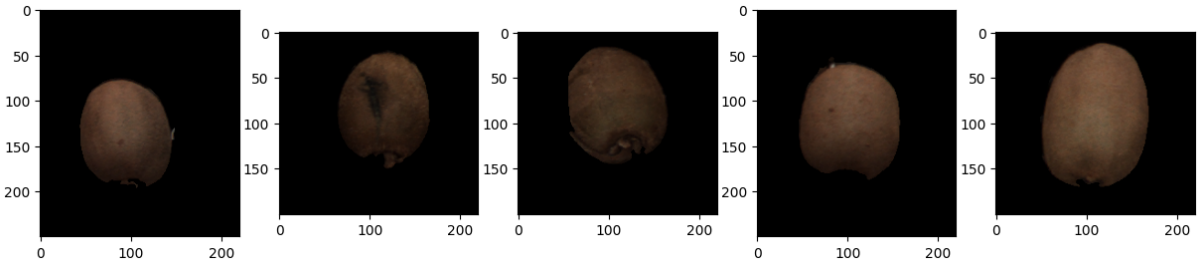

Figure 9: Isolated kiwis.

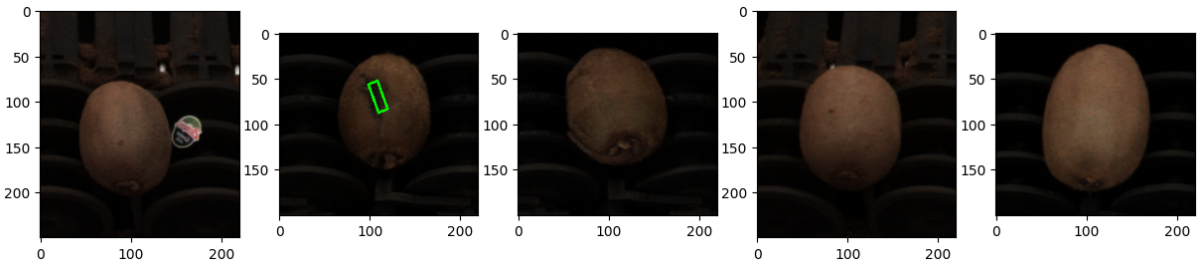

Figure 10: Defects on kiwis.

Bibliography

References

[1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.