Computer Vision Project - Report

Group: Digital Eyes
Authors: Andrea Felline, Arsen Ibatullin, Alessandro Benetti

Introduction

In this project we tried to use classic Computer Vision algorithms and techniques to detect and classify foods and leftovers in canteen trays.
Starting from two images, one with the food before it was eaten and the other with the food after it was (partially) eaten, our program detects the food clusters and then classifies each single food, counting then the pixel of each food before and after it was eaten, to give a score of how much food was left.

The program is divided in three main section:

- Detection of the food pixels (deleting the background) and dividing the pixels among plates, salad cups and bread pieces. This section was written by Andrea Felline.
- Classification of the food in the “before” image, also when there are two or three kinds of foods in the same plate, choosing from the 13 possible food types. This section was written by Arsen Ibatullin.
- Classification of the food in the “after” image, associating each plate with the ones in the “before” image and classifying the pixels inside them. This section was written by Alessandro.

This kind of task would be probably easier to manage with a Machine Learning algorithm, but the lack of example images led us to use classical CV techniques. In the following section there are the instructions on how to use the program and then a detailed explanation of how each of the main sections (and also the testing part) works (each one written by the author, the author of testing is Andrea).

Running instructions

To run the program the user must give the following arguments:

- a number N: the number of tray to be tested (if the user wants to run in “test mode”) or 0 (if the user wants to run in “single mode”)
- if N > 0:
    - Test mode: The program takes in input another argument that is the path of a dataset. It will compute all the images in the dataset and compare with their ground truth, writing in output the mAP and mIoU values
- if N = 0:
    - Single mode: The program takes in input two other arguments that are the paths to two images, ‘before’ and ‘after’ (in this order), and compute them, showing the results with imshow() and printing the leftover score for each plate (#pixels in after / #pixels in before)

The **dataset format** should be the same as the example given in the pdf of the project, so something like this:

- Food_leftover_dataset
  - tray1
    - masks
      - food_image_mask.png
      - leftover1.png
      - leftover2.png
      - leftover3.png
    - food_image.jpg
    - leftover1.jpg
    - leftover2.jpg
    - leftover3.jpg
  - tray2
    - …

Please note that while the dataset folder name can be different from “Food_leftover_dataset” (since it has to be specified as an argument), the other file and subdirectories should have the same exact name and type (also png/jpg file type). In the dataset linked in the pdf there was also a bounding-box folder, that is not necessary with our program (the bounding boxes are computed from the masks) but the presence of other folders or files is not a problem.
To make an example, if the dataset folder name is “Food_leftover_dataset” and the project path is the following:

- Food_leftover_dataset
  - tray1
    - …
  - …
  - tray8
    - …
- src
  - headers
    - …
  - build
    - …
  - main.cpp
  - CMakeList.txt

then, since we are running from the “build” folder, the command to run in test mode would be

```
./project 8 ../../Food_leftover_dataset
```

while, the command to run in single mode would be

```
./project 0 ../../Food_leftover_dataset/tray1/food_image.jpg
../../Food_leftover_dataset/tray1/leftover1.jpg
```

How does it work?

As said, the program is divided into three main sections: detection, “before” classification and “after” classification.

Detection is executed by the function:

```
void detect(
      cv::Mat inputImage,
      cv::Mat& outputImage,
      std::vector<cv::Mat>& masks
);
```

that takes in input an image (any kind, “before” of “after”) as a Mat object, another Mat object where there will be the output image (with background pixels removed) and a vector of Mat objects that will contain the masks for each single plate, salad cup and bread.

“Before” classification is done by the function:

```
void beforeClassify(
      cv::Mat inputImage,
      std::vector<cv::Mat> masks,
      cv::Mat& outputMask,
      std::vector<int>& foodTypes,
      cv::Mat& outputBoxes
);
```

that takes in input a cleaned version of the “before” image (output of detect()) and the corresponding vector of masks (also output of detect()), a Mat object that will contain the output mask (that is an image where each pixel has a value between 0 and 13 that corresponds to the food type), a vector of int with the list of food founded (the vector contains their code) and an image with the bounding boxes and their labels.

“After” classification is done by the function:

```
Tray::void detect_foods(
      Tray* trayBefore,
      cv::Mat maskAfter
);
```

where ‘Tray’ is an UDT that describes a Tray, with an image and a vector of ‘Plates’ (another UDT). Tray objects have (also) this function detect_foods() that takes in input another Tray (that represents the “Before” tray) and a Mat object that will contain the output mask with the same format of **`outputMask`** (pixels with values between 0 and 13).

Then there are other little functions that count and compare the number of pixels for each food code, create a bounding box image for the ‘after’ image, compute a list of bounding box for any image by using the foods mask and that compute the mAP and mIoU values:

```
void count(
      std::map<int,int>& pixelCountingBefore,
      cv::Mat maskBefore,
      std::map<int,int>& pixelCountingAfter,
      cv::Mat maskAfter
);

std::vector<double> compare(
      std::map<int,int> before,
      std::map<int,int> after,
      bool verbose
);

cv::Mat afterBB(
      cv::Mat image,
      cv::Mat maskAfter
);

std::vector<cv::Rect> computeBB(
      cv::Mat mask
);

double mAP(
      std::vector<std::vector<cv::Rect>> pred,
      std::vector<std::vector<cv::Rect>> gt,
      double iouTh
);

std::vector<double> mIoU(
      std::vector<cv::Mat> predMasks,
      std::vector<cv::Mat> gtMasks
);
```

where '**verbose**' in compare() function enables or disables the output printing on the standard output stream. For each food code the maps of count() and compare() will assign the number of pixels that are in the masks. The compare function returns a vector with 13 numbers (the score value for each food, with score -1 when the food was never founded and -2 when it was founded only in 'after'). AfterBB() returns an image with bounding boxes and labels for the after image. ComputeBB() returns a vector of bounding boxes for the image whose mask is provided. mAP() returns the mAP score of all bounding boxes of all the images and classes. mIoU() returns the mIoU scores for each class (given the predicted and ground true masks).

Detection

The detection is done by a set of 3 pairs of cpp-header files, ordered by how high-level the functions are. The first set, the most high-level, are detection.h and detection.cpp. Here there is only one function, detect(), that is the one used by the main to isolate the food pixels.

This function uses the functions in the second-level files (highLevelFunct.cpp and highLevelFunc.h). The function detect() does the following:

- Resizes the image to half its size. Originally this was needed to execute and test the code in a faster way, since some methods required more time. Once everything was finished a lot of parameters were setted to specific values that would probably not work if the image had a different size, so it is still resized and then, at the end, it is re-resized back to the original size
- Calls foodSelector() from highLevelFunc.h. This function isolates the pixels of food from the other pixels, but there are often some non-food pixels that persist.
- Calls platesFinder from highLevelFunc.h. This function locates the plates so it is possible to keep only the food pixels that are inside the plates. The only exception is the bread pieces, that are food but are not on a plate.
- Calls breadDetector from highLevelFunc.h. This function tries to locate the bread, if there is any, among the food pixels that are outside the plates. It returns a point of the image and cleans the image so that only food inside plates is left.
- If bread was found:
    - calls breadSelector from highLevelFunc.h. This function tries to select all bread pixels that are around the point found.
    - Divides the image into plates and bread
    - Applies a different gray threshold to bread and to plates
    - Merge the two image again
- If bread wasn't found
    - Applies a gray threshold to the plates
- Calls generateMasks from highLevelFunc.h. This function returns a vector of masks, one for each plate, salad cup or for the bread. When it is possible to classify the food (for example with bread and with salad, when there are more than 2 plates) it uses the food code as mask value.
- Resize back to original size the image and the masks

The functions in highLevelFunc.cpp and in lowLevelFunc.cpp are explained in detail in the "Experience of Andrea" section and in the code comments.

'Before' image classification

Before you can start object detection, you need to perform pre-processing for each type of food. For each dish, a separate function was formed through which the input image passed.

All functions approximately had one template form:

1. The input to each function was the image itself, an empty vector in which the numbers of dishes will be stored, and a dictionary in which the key will be the number of the dish, and the value will be its area in pixels:

```
cv::Mat detectPastaAndPesto(cv::Mat& image, std::vector<int>& myVector, std::map<int, cv::Rect>& objectDictionary) {
```

2. Having received the image, the first step was to clone it. This step allowed me not to clutter the real image with various rectangles, as they could affect object detection::

```
Mat image1 = image.clone();
```

3. Then the image is converted to the HSV color space:

```
// Convert the image to the HSV color space
cv::Mat hsvImage;
cv::cvtColor(image1, hsvImage, cv::COLOR_BGR2HSV);
```

4. Below are the thresholds for a specific product. They were selected manually using a separately written program that used the TrackBar:

```
// Define the lower and upper thresholds for each color in HSV
cv::Scalar lowerGreen = cv::Scalar(20, 85, 50);
cv::Scalar upperGreen = cv::Scalar(60, 255, 255);
```

5. Next, some manipulations were carried out with the image, such as blurring, dilation and erosion:

```
// Add noise with medianBlur
cv::Mat noisyImage;
cv::medianBlur(hsvImage, noisyImage, 85);

// Image Dilation
cv::Mat dilatedImage;
cv::dilate(noisyImage, dilatedImage, cv::Mat(), cv::Point(-1, -1), 45);

// Image Erosion
cv::Mat erodedImage;
cv::erode(dilatedImage, erodedImage, cv::Mat(), cv::Point(-1, -1), 5);
```

6. Next, a mask was formed that included all the previous requirements::

```
// Create a mask
cv::Mat Mask;
cv::inRange(erodedImage, lower, upper, Mask);

// Find Outlines On Our Color Range Mask
std::vector<std::vector<cv::Point>> contours;
cv::findContours(Mask, contours, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
```

7. Using the findContours function, the contours on the image mask are found:

```
// Find Outlines On Our Color Range Mask
std::vector<std::vector<cv::Point>> contours;
cv::findContours(Mask, contours, cv::RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);
```

8. Next, the function goes through all the detected contours and finds the contour with the largest area. The bounding box of this path is saved for further processing.

```
// Finding the Rectangular Path with the Largest Area
double maxArea = 0;
cv::Rect boundingRect;
for (const auto& contour : contours) {
    double area = cv::contourArea(contour);
    if (area > maxArea) {
        maxArea = area;
        boundingRect = cv::boundingRect(contour);
    }
}
```

9. Next, we calculate the area of the entire image, which allows us to impose some restrictions on the search for unnecessary objects.

10. The bounding box is slightly expanded by adding more pixels to each side, which helps to better cover the entire detected object, since in some cases we may not detect the object completely:

```
// Inflate the bounding rectangle
int inflatePixels = static_cast<int>(std::round(std::max(boundingRect.width, boundingRect.height) * 0.1));
boundingRect.x -= 1.5*inflatePixels;
boundingRect.y -= 0.5*inflatePixels;
boundingRect.width += 3* inflatePixels;
boundingRect.height += 1* inflatePixels;
```

11. Next, it was checked whether the area of the largest detected edge occupied the required area of the total image area. If so, then it is considered a valid discovery. Moreover, we are already starting to select certain characteristics for a given rectangle, as well as enter data into our vector and dictionary. If the initial requirement is not met, we send the image to the next function, where we will look for another dish:

```
if (maxArea / imageArea > 0.01) {

    // Draw a rectangle around the object
    cv::rectangle(image1, boundingRect, upper, 2);

    // Adding text to the rectangle
    std::string text = "Pilaw rice with peppers and peas";

    cv::Size textSize = getTextSize(text, cv::FONT_HERSHEY_SIMPLEX, 0.9, 2, nullptr);
    Point rectangleP (0, textPos.y+5);
    rectangle(image1, rectangleP, rectangleP + Point(textSize.width+10, -textSize.height-10), upper, FILLED);
    cv::putText(image1, text, textPos - Point(1,1), cv::FONT_HERSHEY_SIMPLEX, 0.9, Scalar(255,255,255), 2);
    cv::putText(image1, text, textPos, cv::FONT_HERSHEY_SIMPLEX, 0.9, Scalar(0,0,0), 2);
    textPos.y -= (textSize.height+11);

    // Add the object's ID and bounding rectangle to the object dictionary
    int objectId = 5;
    objectDictionary[objectId] = boundingRect;

    // Add the object's ID to the vector
    myVector.push_back(objectId);
}
else {
    // If the rectangle is not found, run another function
    image1 = detectPastaWithRagu(image, myVector, objectDictionary);

}

return image1;
```

'After' image classification

The “after” image classification uses the concept of similarity in RGB spectrum color of any food to compare the foods in ‘after eating’ images with the foods in ‘before eating’ images.

But some foods like bread and salad can be detected immediately from the previous part in food segmentation for ‘after eating’ images because they have particular plates, for example salad, whose plate is often the smallest, or bread, that doesn't have any plate.

Similarity score

For the RGB spectrum color this section of the program manages the total quantities of red, green, blue from the interval 0 to 255 used in the image of the food.

It scans any pixel in the image and increases the 3 components RGB for the specific value RGB found in the pixel (Eg. R = xxx, G = yyy, B= zzz this increases a counter for the component xxx in R, yyy in G and zzz in B).

The similarity between foods is calculated with a score that calculates the variance between the current RGB color spectrum with another one.

```
double Color_spectrum_layer::similarity_score(Color_spectrum_layer* col_spect_layer){
    double score = 0;
    //Variance of difference RGB spectrum
    for (int i = 0; i < 256; i++) {
        score +=    pow(((double) col_spect_layer->get_R(i)/col_spect_layer->get_total_pixels() - (double) get_R(i)/get_total_pixels()),2) +
                    pow(((double) col_spect_layer->get_G(i)/col_spect_layer->get_total_pixels() - (double) get_G(i)/get_total_pixels()),2) +
                    pow(((double) col_spect_layer->get_B(i)/col_spect_layer->get_total_pixels() - (double) get_B(i)/get_total_pixels()),2);
    }
    return score;
}
```

The formula is : $\sum_{i=0}^{255}(R_1[i] - R_2[i])^2 + (G_1[i] - G_2[i])^2 + (B_1[i] - B_2[i])^2$

where R_1 , G_1 , B_1 are the 3 RGB components for the first layer
R_2 , G_2 , B_2 are the 3 RGB components for the second layer
The lower score between 2 layers corresponds to the best match between them in terms of color similarity.

Find the best matches between foods in the plates

The first step in the ‘after eating’ food detection is to find the best match (in terms of similarity score) between any plate in ‘after eating’ tray with ‘before eating’ tray.
So in this step there is a comparison of the similarity of the RGB color spectrum for any plate in the ‘after eating’ image with any plate in the ‘before eating’ image.
When the minimum score is found, the two plates in ‘after eating’ image and in ‘before eating’ image are matched together.
The input in this step are the relative masks of segmented food for any plate in ‘after eating’ image, that allows any single plate to be isolated and also to automatically detect bread and sometimes also salad.
Since for one plate in ‘after eating’ we need to have only one corresponding plate in ‘before eating’, the program needs to exclude, in the following iterations, the plates that have already been matched.
A vector of already-matched plates allows the function to exclude them and improve the detection.

Single food in a plate versus Multi food in a plate

Until now the program finds a matching “1:1” between “before” plates with “after” plates; but a plate can have more than one food so I need to do other steps of elaboration to create a matching between “after” and “before” about food, and not only plates.
We can assume that the corresponding plates between “after” and “before” have the same foods, or, more precisely, the “after” plate can have a subset of foods of the “before” plate.
This means that it is possible to iterate the same idea of similarity, but this time comparing some parts of the food in the “after” plate with the full food cluster found in the “before” plate.
Because of the irregularity of remaining foods in a plate I need to work “pixel by pixel”, or with pads of n*n pixels of the remaining foods in the plate.

Working pixel by pixel should be more accurate but most costly in terms of time consumption of elaboration, and also most foods will likely be the same in a little neighborhood of pixels. Tests can show that the time of computation decreases with a small pad, and the accuracy depends on the size of the pad (a small pad doesn't affect the accuracy too much). The best results were given by a pad of 3*3 pixels (in terms of accuracy and time).
Pixel by pixel was anyway not really a good choice since the RGB color spectrum is really limited from only 3 values RGB, so it is really not accurate.
In any pad the background pixels are excluded from the RGB color spectrum and from the similarity score calculations.
With a single food in a plate the pad-by-pad similarity score is unnecessary and we already have the matching between foods.
So, after these elaborations, the "after" mask is created. It uses a different gray level for each different food (the gray level is the same as the food code).

Classes and Methods

To manage everything and to make the code reusable and maintainable, the program uses an object-oriented approach. In detail it uses these classes with their relative scope:

1) **Color_spectrum_layer**
   This class manages the RGB color spectrum of a generic layer (subset of pixels) of an image.
   It manages the layers of the specific foods in the image, the layers of all foods found in a plate and the pad of remaining food in a plate.
   This class give the following functionality:
   1) Similarity score calculus
   2) Total pixel calculus for the layer
   3) Adding multiple quantities of RGB components in the color spectrum layer
   4) Add the quantities of RGB components for a specific pad

2) **Food**
   This class manages the IDs of the food found, that are numbers between 1 to 13 and that specifies the possible classification of food.
   The value is setted to 255 when it is a generic food that still needs to be classified from before images.
   This class give the following functionality :
   1) Set and return the ID of the classification of the food
   2) Set and return the relative RGB color spectrum for the food

3) **Plate**
   This class manages all foods inside one plate.
   In particular it can manage all the foods inside the plate, the relative mask of foods in 'leftover' image and return a global RGB color spectrum.
   So the functionalities are:
   1) Get an RGB color spectrum for all foods inside the plate with the total number of pixels
   2) Get an RGB color spectrum for any food inside the plate
   3) Get a vector of foods

4) Get the mask with the foods, useful in particular for the “before” plate but it will be updated in the “after” plate when the foods are detected.
5) It can use the ground truth mask (only for testing purposes) to detect “before” foods.

4) Tray

This class uses the concept of tray, in fact it is a set of plates (where ‘plates’ can be a more abstract concept, a food can be a plate if it is far from others, like bread).
In particular there are two plates (bread and salad) with only one food inside it, while the other plates can have one or more foods.

The tray is the structure with the highest level and makes the ‘after’ classification work with respect to the ‘before’ tray.
Indeed the program can have 2 different kind of trays :

1) “After” tray: It is the tray with the remaining foods from ‘leftover’ image
2) “Before” tray: It is the tray with the whole foods from the ‘before’ image

So the offered functionalities are:

1) Get all the plates of the tray
2) Detect foods in the leftover image with a personalized pad (the program use pad 3*3 for the classification, but it is given as a parameter)
3) Get the mask image and the ground truth image (only for the testing purpose)
4) Get the leftover segmented image
5) Reorder the plates, putting in the first position the already detected plates and then the plates with an undetected food.

Testing

As requested in the pdf, the testing of the program calculates the mAP and mIoU scores of every before-after couple of images in the dataset.
For each tray the testing function does the following:

1. Takes the ‘before’ image and compute its mask with our functions
2. Reads its ground truth mask
3. Compute the bounding boxes of both masks
4. Save the bounding boxes in a vector (sorted by food class)
5. Save the masks in a vector

6. For each ‘leftover’ image:
    a. Reads it and compute its mask with our functions
    b. Reads its ground truth mask
    c. Compute the bounding boxes of both masks
    d. Save the bounding boxes in a vector (sorted by food class)
    e. Save the masks in a vector

7. Saved bounding boxes are given to the mAP function that returns the mAP score
8. Saved masks are given to the mIoU function that returns the mIoUs scores for each food

The function mAP() takes the predicted and ground truth boxes and a IoU threshold and, for each one of the 13 classes, calls the function AP(), then sums the results and divides them

by 13. The AP() function calculates the average precision score over a single class, it goes over every couple of boxes and calculates their IoU, saving the best match. If the best match is more than the threshold (0.5) then it is considered a true positive, otherwise it's a false positive. For each predicted box then it sums "TP/(TP+FP)" and at the end it returns this sum over the number of ground truth boxes.

The mIoU() function goes over every mask, isolates the sub-mask of a specific food in both predicted and ground truth mask, compute the IoU pixel by pixel and sums up the results (counting how many times it is more than 0), then at the end divides the result by that counter for each food and returns the results.

What did we do?

Here each member of the group explains his experience in the project, listing all the steps he went through to write the functions that, together, composes this program.

Experience of Andrea

During the development of this project, I went through several stages. Initially, after receiving preliminary instructions from the professor but before the release of the PDF with the guidelines, I started exploring potential solutions for food detection using images found online. The first thing that stood out when looking at this type of image was the plates with food in them. Therefore, my initial approach was to apply the **Hough algorithm** to detect circles (and thus plates).

Once the PDF was released, I continued implementing the Hough algorithm, adapting it to the specific context. For the Piovego canteen trays, the tray bottom and plates had particularly light colors. So, I thought it would be possible to isolate the food by **removing the regions with lighter colors**. I implemented an algorithm to remove unwanted white regions from the images by applying a threshold on the maximum value of the RGB channels (considering them "white" if all three channels exceeded a certain threshold).

The results were not very good, but I noticed that the remaining shadows of plates and food after white removal made it easier to find the right circles. So, I **combined the two algorithms** and tested them on various food_images from different trays (see attached screenshot). This combined algorithm seemed to work particularly well for plate detection, but it still needed improvement.

A few days later, we had the first Zoom meeting to discuss the approaches to be adopted. During this meeting, I presented an initial version of the algorithm using Hough Circles combined with white removal. We started listing proposals for methodologies and approaches to try, and we found some relevant papers. However, most of the material suggested machine learning approaches, which we preferred to avoid.

At this point, since I was focusing on food cluster detection while the others had not started yet, I proposed dividing the work into three parts: detection (approximate detection of food position), segmentation (accurate removal of all non-food pixels and division by different food types), and classification (assigning a specific food category among the 13 possible

categories to each pixel). I would have taken care of the first part, while my two colleagues would handle the other two. The proposal was accepted.

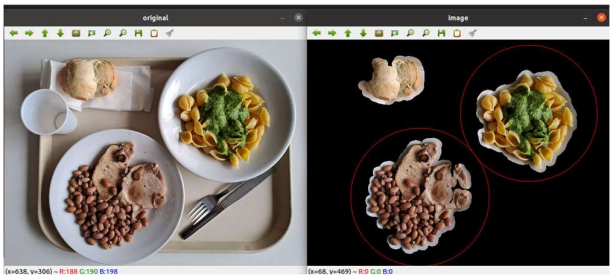
While testing other possible approaches, I noticed that there are many small edges in food (e.g., rice consists of hundreds of grains, each with its own edge). So, I started experimenting with the idea of using a combination of **Canny edge detection and dilation** to create a food mask. It worked particularly well, unlike other approaches like k-means clustering (I tried with different combinations of elements in feature vectors and different preprocessed versions of the image, none of which gave good results).

I then implemented an algorithm based on Canny, dilation, and removal of unwanted gray components. In this case, instead of removing white as before, I **removed gray**. I checked the difference between each pair of the RGB channels, and if the difference of at least one pair did not exceed a threshold, it was considered gray. This new version allowed for the removal of darker grays, shadows, and varying background brightness. This approach produced excellent results, but it tended to include yogurt, paper sheets, and other non-food objects that were not part of the plates or bread.

Until this point, the work was quite informal and unstructured. Each technique had its own CPP file, which implemented it and added a series of trackbars for parameter selection. So, I created a more structured version of the project, dividing the activities and creating headers for each of them, as well as a CMakeLists file. I created a GitHub repository and shared everything with the rest of the team (who had only received screenshots of my progress until then).

A colleague then introduced me to a useful OpenCV function simulator for Windows, which made it particularly easy to test different methods. At this point, I had a good detection of food on plates, but I still needed to find the bread accurately. Therefore, I used this tool to run some tests, including using features to locate the bread. However, since each piece of bread has slightly different characteristics, it was not possible to use a single sample that worked on all images. I also tried using Hough Lines to detect the tray (with the idea of locating and classifying all objects on it), but it rarely worked.

I then **combined the two best techniques found so far** (Canny+dilate and Hough circles), achieving promising results (see example in screenshot with red circles). However, circles outside the plates were also detected (often due to the tray's curves). To improve the Hough algorithm, I implemented the removal of empty circles (with a number of non-black pixels inside below a threshold) or circles that were too overlapped. I updated the official version of the detection function, called "detect(Mat)," which from this point onwards was available to my project colleagues as well.



Now that all the plates (and salads) in the available dataset were being correctly detected, I started working on bread detection, trying to find RGB values using trackbars, but without success. I also attempted to compare the relative difference between the color channels (e.g., ensuring that red was at least 50 units higher than blue), but this approach also did not yield the desired results.

To achieve accurate **isolation of the bread**, I used Photoshop to crop all bread portions and wrote a program to find common RGB values among the pixels in different images. I calculated the median, minimum, and maximum values of these pixels and used them to isolate a representative bread pixel in each tray, excluding pixels outside the plates. Since this approach often selected the tray, I added a new step of Canny, erode, and dilate (since the tray has few distinct edges compared to the bread). At this point, I could find the **position of the bread** (a single point in the image) in each tray of the dataset. I then created a **circular mask with a radius of 100 pixels around that position** (to capture the entire bread and its surroundings, see example in screenshot). From these "bread circles," I created a separate dataset by cropping each of them, allowing me to focus on bread-related tasks.

I made further attempts to **remove the bread's background**, exploring different techniques such as using findContours (applied to various preprocessed versions of the image) and switching to the YCbCr color space, as ultimately adopted in the final version.

At this point, the group had another call, during which the new tasks were assigned to each team member (since Arsen had already started working on image classification). The focus was on food detection and classification for the "before" and "after" images.

I created several diagrams to illustrate the structure and composition of the project, providing a clear and concise explanation of each team member's tasks (see diagram on “how it works?” section). I then added Arsen's code to mine.

Regarding bread processing, I tried a combination of different techniques. Starting from the "enlarged" initial point of 120 pixels, I applied a **threshold on the YCbCr channels**. The results were then merged, and from the result, a new circle mask (similar to the initial 120-pixel one but more precise) was obtained. Subsequently, starting from this mask, I again used **Canny,** combined with a series of **morphological operators and findContours**. This approach detected bread quite well, but the contours appeared very jagged (see green lines in the screenshot). Finally, a strong **median blur filter** was applied to round the selection, making it decent (although imperfect) in almost all cases.

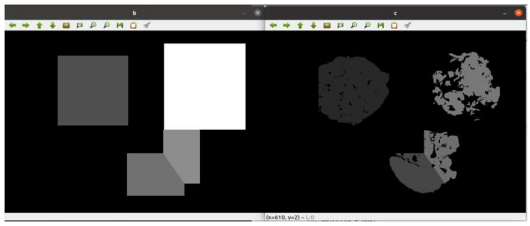
The output of my section, upon the colleagues' request, became a color image with only food pixels and a vector of masks dividing plates, salads, and bread.

One last little improvement was using again the gray deleting function on food and bread, to remove the last pixels of background, with a different threshold for bread and for plates.

Until this point, for faster testing, I worked on a downscaled image at 50%. Changing this detail would have required updating each section's parameters individually. Therefore, I decided to leave the images downscaled and then, at the end, return to the original size,

applying the masks to the original figure (resulting in slightly pixelated edges, but it should have minimal impact).

We then decided to use dictionaries to establish correspondences between food items and the number of pixels. I added a function to count the number of pixels for each food in the masks “before” and “after”, and one function to compare the obtained results and generate scores for each food item using the formula requested in the PDF.



On the last days I helped Arsen with some last details of his functions, since the deadline was near. I wrote for him a function that manages in a very simple way the intersection of bounding boxes (the intersection is just splitted in half between the two boxes) and a function to avoid having “dirty pixels” (some boxes were too big and painted with the wrong color also some pixels of other near plates, this caused some problems in Alessandro’s classification) that just merged the mask of the plates with the mask of the food (so other plates are safe and can’t be compromised). In order to finish in time I also wrote some other functions (like afterBB and the testing function calculations) that helped to complete the work.

Day - working hours correspondences (approximative):

24 May	5h
25 May	5h
26 May	2h
27 May	7h
29 May	5h
30 May	7h
31 May	5h

01 June	4h
05 June	5h
06 June	7h
07 June	5h
08 June	7h
13 June	3h
14 June	5h

15 June	5h
25 June	4h
04 July	7h
06 July	2h
07 July	3h
08 July	5h
11 July	3h

12 July	3h
14 July	2h
16 July	1h
17 July	7h
18 July	7h
19 July	8h
TOTAL	**129h**

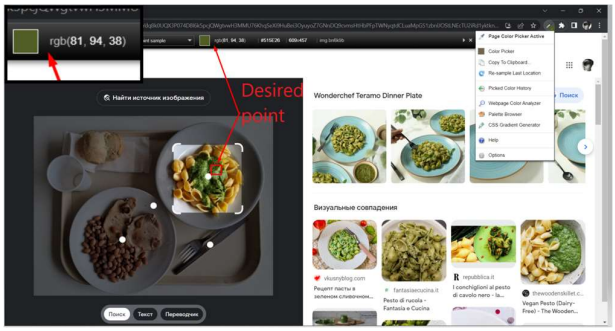
Experience of Arsen

Together with the team, we have allocated tasks, and I have been assigned the task of food detection in images, specifically before it is consumed, as provided by Andrea after processing. After studying various resources on object detection, I have discovered that most of such work is accomplished through machine learning. I have decided to begin by searching for images of each type of food that may be present on a tray. Over the course of several days, I have simultaneously explored neural networks capable of detecting various types of objects. I have chosen to focus on convolutional neural networks, as they are widely used in computer vision tasks. Throughout the week, my skills in neural networks have improved. However, the process of finding a suitable dataset of food photographs has proven to be challenging and unenjoyable. After a week, I decided to delve into pixel-level analysis, as daily contemplation of the task gradually led me to the right ideas and insights

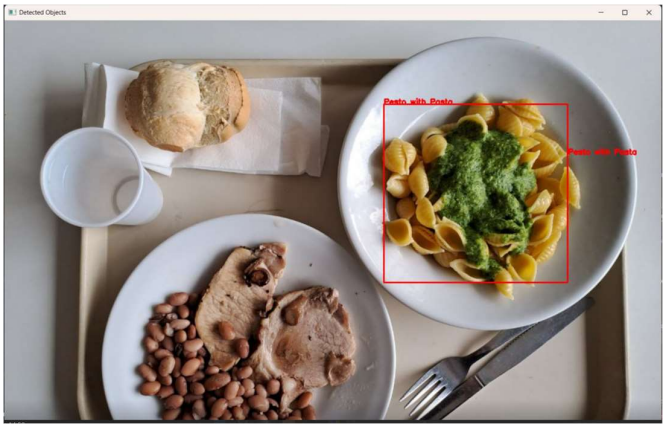
regarding my objective. I resolved to work with pixels, specifically with the range of colors associated with each object.

First ideas:

Initially, I would take an image and upload it to Google, where I used the ColorZilla application to determine the RGB color for a specific type of food. For example, for pasta with pesto, the dominant color was green. I needed to select the lightest and darkest shades based on my perception to determine the most appropriate range for that particular food.

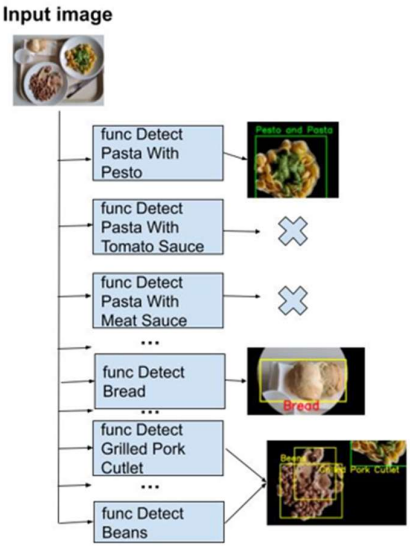


Therefore, after considerable effort and time, I managed to determine the ranges for some types of food that were predominantly characterized by a single color. The initial result appeared as follows:



In the process of detecting food on a tray, I utilized various image processing techniques to improve the accuracy and reliability of the food detection algorithm. One essential aspect was the application of blur, including techniques like median blur, dilation, and erosion. Blur played a crucial role in reducing noise and unwanted details, thereby enhancing the quality of the input images. This improved image quality was instrumental in achieving more accurate food detection results.

Additionally, dilation played a significant role by expanding the boundaries of food objects, bridging any small gaps that may have existed between them. This operation helped create more complete and connected representations of food items on the tray. On the other hand, erosion was employed to refine the object boundaries and separate



connected food objects. By selectively removing pixels from the object edges, erosion eliminated small details, noise, and spurious elements, thereby improving object separation.

However, there were significant challenges in detecting dishes that fell within the same color range. My initial idea was to create separate functions for each of the thirteen dishes we have, which would operate independently from one another.

This led to the fact that one object could be detected as two or three different foods.

Second step:

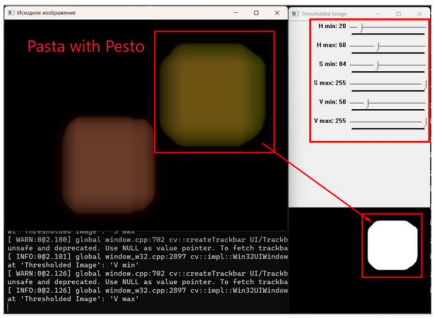
Furthermore, during the course of my work, I made the decision to transition from the RGB color space to HSV (Hue, Saturation, Value) color space. This choice was motivated by the realization that working with HSV would provide greater convenience and suitability for my specific task of food detection.

By switching to HSV, I gained several advantages. Firstly, the Hue component captures the dominant color information, allowing for more robust and reliable color-based food detection. Saturation provides valuable information about the intensity or purity of the color, helping to distinguish between different shades of the same color. Lastly, the Value component represents the brightness or lightness of the color, which aids in further discriminating food objects from the background.

This shift to HSV facilitated a more intuitive and effective way of analyzing and processing color information for food detection. By leveraging the unique properties of the HSV color space, I was able to refine my approach and improve the accuracy of identifying and distinguishing various types of food on the tray.

However, my mistake was that at the initial stage I used ColorZilla to get the RGB format, and then converted it to HSV. It was a very long process, which, of course, gave results, but not so quickly.

Following one of our discussions, I recognized the need to incorporate a TrackBar into the system. This interactive tool enables individual adjustments and threshold selections for each type of food product independently.

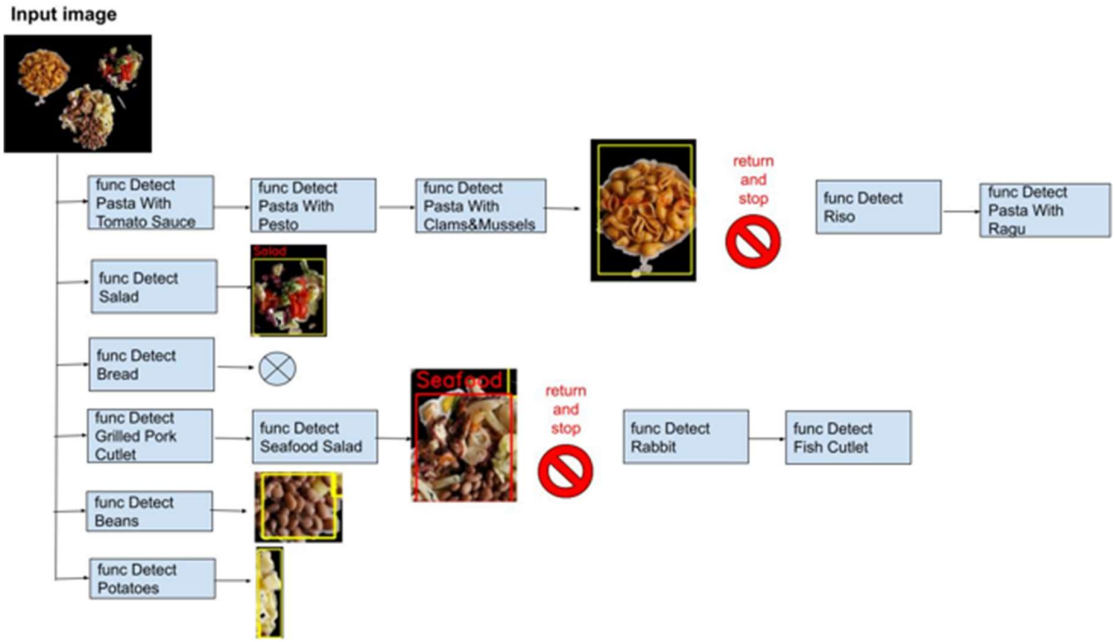


The challenge posed by specific food items was effectively addressed, particularly following the receipt of the processed image from Andrea, in which only the contents within the plates and the bread were retained. Nonetheless, certain products exhibited overlapping threshold values, while food items such as salads presented a diverse spectrum of colors that necessitated simultaneous detection.

While adjusting the values for each food item, I had to neglect certain HSV values due to color intersections, particularly in the brown color range, observed in products like beans,

rabbit, and grilled pork cutlet. Fine-tuning the threshold values for each item was time-consuming. Changing the object extraction methods also led to mixed detections. My mistake was initially working with screenshots, as I discovered that the color values in the screenshots didn't always match the true color values in the automatically captured images. Therefore, when I transitioned to using the accurate processed values, I encountered an error and had to readjust everything once more. In the case of salad, I used a combination of masks. Since this product has a large number of colors and if they are described by one threshold value, then we can capture too large an interval of colors.

The problem of incorrect detection was still relevant, I tried to find threshold values for each dish, however, I had to sacrifice the quality of detection. I had to change the structure of the program and then I decided to make the following search structure, where the functions worked sequentially. For example, a plate of pasta and pesto will only be searched if we don't find pasta with tomato sauce, and so on. You can see a more detailed diagram in the picture below:



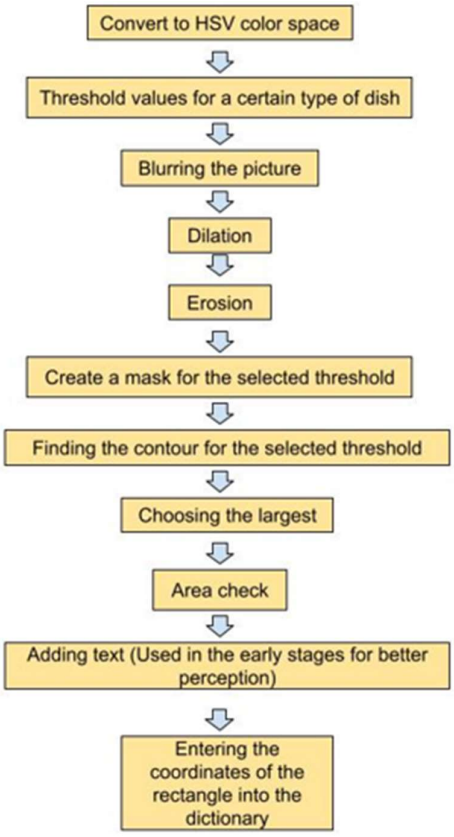
Inside, the function for detecting a dish in the image was approximately the same for everyone, and you can see it in the picture on the right:

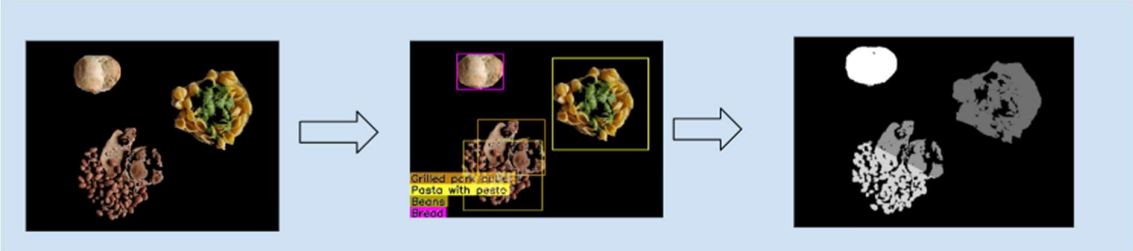
Regarding bread detection, we managed to eliminate the need for its specific identification, as the mask designed for this particular product turned out to be the smallest among all dishes. This refinement allowed us to streamline the detection process, leading to an overall

improvement in the program's quality. Since bread exhibits a variety of yellow and brown shades, this optimization significantly contributed to accurate dish recognition.

After executing all the functions, post-processing of the obtained image was required, involving the preparation of masks with corresponding results and dictionaries containing the pixel counts for each dish. This processing was carried out within the "highlightObject" function. This function takes the input image and a dictionary of objects with bounding rectangles as its parameters. It generates binary masks for each rectangle and merges overlapping masks.

Next, work was carried out to process the overlap between the rectangles in the image. The task was to determine which rectangular areas overlapped with each other, and update the masks of these areas accordingly. The function sequentially checks all the rectangles, and if there is an intersection, then the overlap mask is calculated. The number of non-zero pixels in this mask is then analyzed to determine the degree of overlap. As a result, we get masks of rectangular areas without overlap, which allows us to analyze the image more accurately.





As a result, a dictionary with values and their area was ready for output, as well as masks with food, which were needed for the final check with the mask prepared by Alessandro.

Time spent for the project:

01.06-6h	12.06-5h	23.06-5h	03.07-6h	13.07-6h
02.06-3h	14.06-3h	25.06-7h	07.07-8h	15.07-7h
03.06-7h	15.06-7h	26.06-5h	08.07-3h	16.07-9h
05.06-4h	16.06-5h	27.06-7h	09.07-5h	17.07-5h
07.06-7h	17.06-7h	28.06-3h	10.07-4h	18.07-4h
08.06-4h	18.06-6h	29.06-9h	11.07-5h	
10.06-3h	18.06-3h	30.06-3h	12.07-5h	
11.06-4h	19.06-3h	04.07-7h	14.07-3h	**Total: 193h**

Experience of Alessandro

My experience begins with a brainstorming session with my colleagues on how to find a very efficient way to detect parts of foods that can be very small (1x1 pixel too), considering that I can't have any regularity in the image. I have thought initially of using some of my previous experience in machine learning and statistical analysis to find how much similar can be a portion of image with another in terms of RGB color spectrum, without using other more sophisticated ways, because the color similarity can be adapted in any possible situation also for a pad 1x1 pixels.

So I was thinking about how to calculate the RGB color spectrum similarity between different parts of image (food) and manage it in the best reusable and maintainable way.
I have found that object orientation in c++, with classes and methods, can be the best way to manage this and make the code more easy to read and modify.

So initially I was thinking of adapting all the code in order to use object orientation, but this was not useful for the before classification and segmentation because this didn't give any advantage and would have made the time of the development of the program overly longer.

So I have created some functions to read the RGB colors pixel by pixel from a mask and the relative image, in this case from the "before" image.

After that I started to create all the classes.
I started from the Color_spectrum_layer class. I have written the declaration and the definition of some methods, starting from the constructor, copy constructor and then I added other methods for acquiring pixel by pixel the RGB color spectrum while I was scanning any pixel from the mask and the image.
I have revised the concept of pointer and allocation on the heap and find that in C++ it is very complicated to manage the memory in efficient way so I need deallocate manually all the memory that is allocated in the heap because there is no garbage collector in C++ (while Java has it).
I tested very often with some testing code in the main and with the debugger to assure the correctness of each method.

So I have started to write the Food class to manage the food ID and I was thinking of creating a RGB color spectrum for any food so I would be able to calculate the color spectrum when I scan the image with the mask, in particular for ‘before’ image.
My idea was to create an empty RGB color spectrum and fill it with the data when I scan the image; indeed I have a method to return a pointer to the RGB color spectrum for all the foods that I find in the image.
I have written the declaration and then the definitions, then I tested the correctness with some test code in the main.

When the class Food was completed, I wrote the class Plate to manage the whole plate that is composed from some foods.
Initially I was thinking of managing a global RGB color spectrum differently from any food and fill during the reading from the image and the mask but then I thought that is a redundant approach, so I created a method that updates the global RGB color spectrum from any RGB color spectrum of foods.
In this case I can easily calculate the number of pixels and the global RGB color spectrum for the whole food in the plate, which is a very important step of the after classification algorithm.

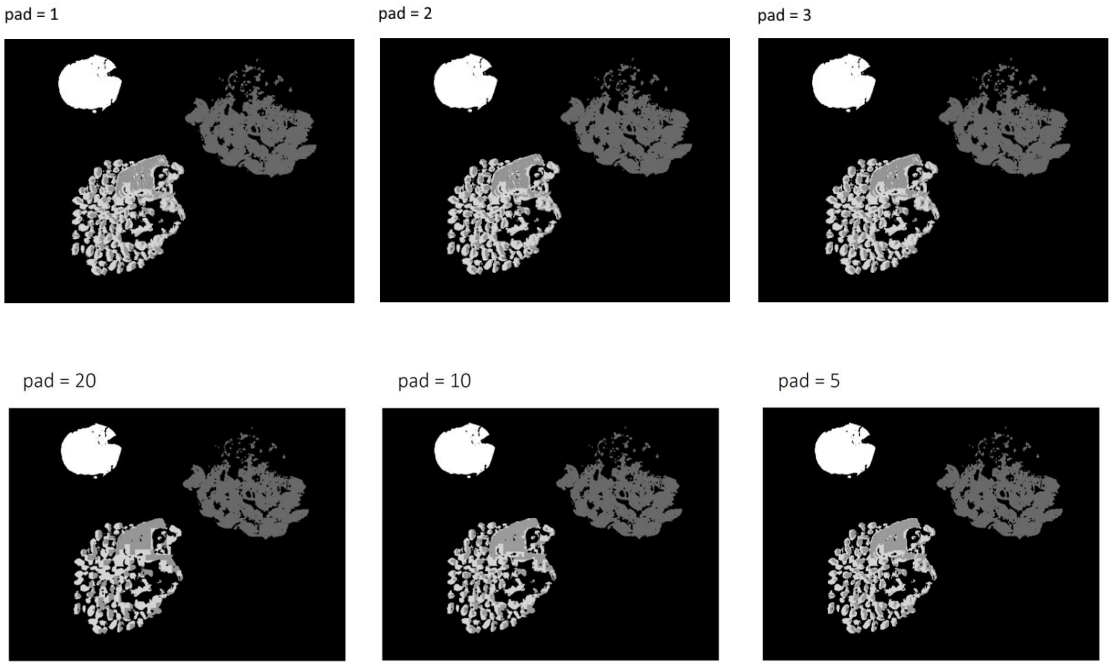
In the end I created the last class Tray that manages all the plates in the tray.
This was the most important class and was very challenging because this class offers the functionality to make the after classification actually work.
When I created this class I declared and defined some methods, first of all the method to acquire data from a mask, an image and all plates and relative foods within a specific RGB color spectrum.
So I started to create the function to detect foods. Initially I was thinking about a static method of class that takes 2 trays and modifies the “after” mask with the foods detected, but after I chose a method that takes another tray and calculates the similarity score.

Meanwhile I thought about the best formula to manage the similarity score; initially I was thinking to sum all the values for the tray and subtract the values with the other tray and power to square, but I was thinking that we need check component by component for R, G and B separately so I used the concept of variance for any component so I can find a more correct similarity score in this way.
So I found the similarity score between 2 RGB color spectrum and tested it; but at this point I had another problem to solve: I can find a score between plates but a plate can have more than one food, so I need to check similarity about foods inside the plates.
Now I have a correspondence between “after” plate and “before” plate and for ‘before’ plate I have all the foods with the relative areas. Using a similar approach about similarity score I created a similarity score between two pixels in the “after foods“ plate with any RGB color spectrum of any food in the ‘before’ plate.
But I found the first problem about this technique: the efficiency. Because of the high level of allocated memory in the heap, the high number of pixels and using low efficiency data structures, the technique resulted very slow.

So I thought of a new solution to improve the efficiency and by brainstorming between all members of the group we thought about the concept of pad. The pad is a group of pixels that are considered as a single block and the similarity score is considered between the pad and the full ‘before’ food.

I adapted the code to work with the pad, also checking and not taking in consideration eventual background pixels in the pad. The similarity score is created “on the fly” while dividing the image in pads.
I also worked to optimize the code, deallocating (when it is possible) the allocated memory in the heap, and changing some data structure to improve the efficiency (like using an array of int with dimension equal to 256 and not the map structure from unsigned char to int).
I reduced the time of elaboration and improved the accuracy a lot with this code update.
However I had some false positive and inconsistent classification, like to classify 2 “after” food with the same “before” food.
So I ordered the foods in the tray array putting the already-detected foods before the still-to-be-classified foods.
And when I match an “after” plate with a “before” plate, I exclude the “before” plate in the next computations.
With this last update in the code, and after finding the optimal dimension for a pad (3*3), the results and the efficiency resulted really good.

Here I show some examples of “after” masks with different pads and we can see the result for pad =1 , pad = 2 and pad = 3 are similar but the best tradeoff for time consumed for the elaboration and the accuracy of result is for pad =3.



Test a new methodology for before classification

Because the result was really good in general I was wondering if it is possible to use the same approach to the “before” classification.
So I had an idea to calculate for all the parts of the image, for any food, an average RGB color spectrum and store this data that can be serialized very easily in a JSON format.
So we could have the RGB color spectrum that defines any kind of food.
After that, using the similarity score, we could find, with the same approach like “after” classification, the best score. In the case of multi-food we could reconstruct the mask with the similarity score for any pad.

If that was done, we could then make a comparison, with some metrics, of the two different techniques, and find the best one.
However this is only an alternative idea to make before classification works, this is a possible future test that could be realized.

Personal Conclusion

I have revised my c++ knowledge and learned how to work with the OpenCV library for a challenging project.
I have learnt many things about Computer Vision that I would like to use in my future work career or for other academic projects.

Time spent on the project

25 June	2h	1 July	2h	7 July	4h	13 July	4h	19 July	6h
26 June	3h	2 July	3h	8 July	3h	14 July	6h	20 July	8h
27 June	3h	3 July	3h	9 July	4h	15 July	3h		
28 June	2h	4 July	2h	10 July	4h	16 July	4h		
29 June	2h	5 July	3h	11 July	6h	17 July	4h		
30 June	3h	6 July	2h	12 July	4h	18 July	5h	TOTAL	95h

Testing results

Testing on the full dataset gave us the following result:

- mAP: 0.88
- mIoU
    - Pasta with pesto: 0.693
    - Pasta with tomato sauce: 0.615
    - Pasta with meat sauce: 0.720
    - Pasta with clams and mussels: 0.695
    - Pilaw rice with peppers and peas: 0.724
    - Grilled pork cutlet: 0.360
    - Fish cutlet: 0.475
    - Rabbit: 0.359
    - Seafood salad: 0.273
    - Beans: 0.337
    - Basil potatoes: 0.424
    - Salad: 0.598
    - Bread: 0.822

Food scores (#pixels in after / #pixels in before):

Predicted:

- Tray 1
  - Leftover 1
    - pasta with pesto: 0.898
    - grilled pork cutlet: 1.129
    - beans: 0.964
    - bread: 0.944
  - Leftover 2
    - pasta with pesto: 0.725
    - grilled pork cutlet: 0.951
    - beans: 0.496
    - bread: 0.762
  - Leftover 3
    - pasta with pesto: 0.491
    - grilled pork cutlet: 0.656
    - beans: 0.261
    - bread: 0.676
- Tray 2
  - Leftover 1
    - pasta with tom s: 0.996
    - fish cutlet: 1.300
    - basil potatoes: 0.811
    - salad: 1.005
  - Leftover 2
    - pasta with tom s: 0.645
    - fish cutlet: 0.776
    - basil potatoes: 0.548
    - salad: 0.758
  - Leftover 3
    - pasta with tom s: 0.835
    - fish cutlet: 0.382
    - basil potatoes: 0.329
    - salad: 0.425
- Tray 3
  - Leftover 1
    - pasta with tom s: 0.775
    - rabbit: 0.671
    - salad: 0.892
  - Leftover 2
    - pasta with tom s: 0.670
    - rabbit: 0.567
    - salad: 0.746
  - Leftover 3
    - pasta with tom s: 0.815
    - rabbit: 0.866
    - salad: 0.552
- Tray 4
  - Leftover 1
    - pilaw rice: 1.072
    - fish cutlet: 0.944
    - basil potatoes: 0.366

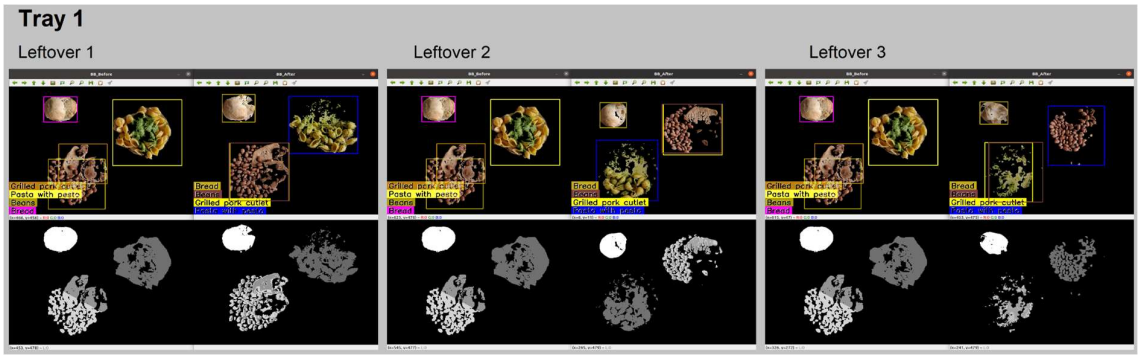
Ground truth:

- Tray 1
  - Leftover 1
    - pasta with pesto: 0.74
    - grilled pork cutlet: 0.817
    - beans: 1.285
    - bread: 0.870
  - Leftover 2
    - pasta with pesto: 0.487
    - grilled pork cutlet: 0.431
    - beans: 0.845
    - bread: 0.815
  - Leftover 3
    - pasta with pesto: 0
    - grilled pork cutlet: 0
    - beans: 1.068
    - bread: 0.561
- Tray 2
  - Leftover 1
    - pasta with tom s: 1.005
    - fish cutlet: 1.067
    - basil potatoes: 1.076
    - salad: 1.236
  - Leftover 2
    - pasta with tom s: 0.342
    - fish cutlet: 0.747
    - basil potatoes: 0.732
    - salad: 0.707
  - Leftover 3
    - pasta with tom s: 0
    - fish cutlet: 0.357
    - basil potatoes: 0.247
    - salad: 0
- Tray 3
  - Leftover 1
    - pasta with tom s: 0.344
    - rabbit: 0.770
    - salad: 0.748
  - Leftover 2
    - pasta with tom s: 0.318
    - rabbit: 0.843
    - salad: 0.795
  - Leftover 3
    - pasta with tom s: 0
    - rabbit: 0.893
    - salad: 0
- Tray 4
  - Leftover 1
    - pilaw rice: 1.087
    - fish cutlet: 0.598
    - basil potatoes: 0.590

- salad: 0.775 - bread: 1.302 - Leftover 2 - pilaw rice: 0.984 - fish cutlet: 0.751 - basil potatoes: 0.422 - salad: 0.004 - bread: 1.152 - Leftover 3 - pilaw rice: 0.290 - fish cutlet: 0.095 - basil potatoes: 0.0709 - salad: 0.640 - bread: 0.177 - Tray 5 - Leftover 1 - pasta with meat s: 0.751 - rabbit: 0.540 - beans: 1.317 - bread: 0.717 - Leftover 2 - pasta with meat s: 0 - rabbit: 0.402 - beans: 0.768 - bread: 0.967 - Leftover 3 - pasta with meat s: 0.456 - rabbit: 0.317 - beans: 0.592 - bread: 0.765 - Tray 6 - Leftover 1 - pasta + clams&mus: 0.765 - grilled pork cutlet: 0.521 - beans: 1.337 - salad: 0.875 - Leftover 2 - pasta + clams&mus: 0.805 - grilled pork cutlet: 0.463 - beans: 1.409 - salad: 0.896 - Leftover 3 - pasta + clams&mus: 0.479 - grilled pork cutlet: 0.065 - beans: 0.087 - salad: 0.575 - Tray 7 - Leftover 1 - pasta + clams&mus: 0.674 - fish cutlet: 0.343 - basil potatoes: 0.597 - salad: 0.912	- salad: 1.165 - bread: 1.456 - Leftover 2 - pilaw rice: 0.868 - fish cutlet: 0.583 - basil potatoes: 0.556 - salad: 0 - bread: 1.289 - Leftover 3 - pilaw rice: 0.203 - fish cutlet: 0 - basil potatoes: 0.156 - salad: 0.480 - bread: 0.368 - Tray 5 - Leftover 1 - pasta with meat s: 0.477 - rabbit: 0.698 - beans: 0.662 - bread: 0.709 - Leftover 2 - pasta with meat s: 0 - rabbit: 0.679 - beans: 0.254 - bread: 0.963 - Leftover 3 - pasta with meat s: 0 - rabbit: 0.620 - beans: 0.266 - bread: 0.836 - Tray 6 - Leftover 1 - pasta + clams&mus: 0.606 - grilled pork cutlet: 0.278 - beans: 1.237 - salad: 1.084 - Leftover 2 - pasta + clams&mus: 0.617 - grilled pork cutlet: 0.235 - beans: 1.241 - salad: 1.008 - Leftover 3 - pasta + clams&mus: 0.266 - grilled pork cutlet: 0 - beans: 0 - salad: 0.445 - Tray 7 - Leftover 1 - pasta + clams&mus: 0.512 - fish cutlet: 0.416 - basil potatoes: 0.494 - salad: 0.912

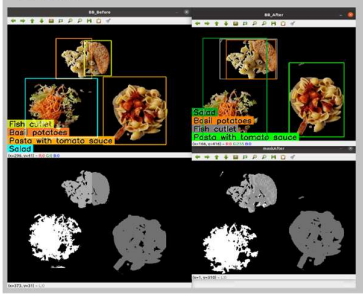
- Leftover 2
  - pasta + clams&mus: 0.738
  - fish cutlet: 0.349
  - basil potatoes: 0.624
  - salad: 0.941
- Leftover 3
  - pasta + clams&mus: 0
  - fish cutlet: 0.050
  - basil potatoes: 0.085
  - salad: 0.753
- Tray 8
- Leftover 1
  - pasta + clams&mus: 0.473
  - seafood salad: 0.252
  - beans: 0.296
  - basil potatoes: 0.317
  - salad: 0.652
- Leftover 2
  - pasta + clams&mus: 0.738
  - seafood salad: 0.799
  - beans: 1.197
  - basil potatoes: 0.743
  - salad: 1.171
- Leftover 3
  - pasta + clams&mus: 0.481
  - seafood salad: 0.073
  - beans: 0.013
  - basil potatoes: 0.019
  - salad: 0.234
	- Leftover 2
	  - pasta + clams&mus: 0.561
	  - fish cutlet: 0.488
	  - basil potatoes: 0.537
	  - salad: 0.978
	- Leftover 3
	  - pasta + clams&mus: 0.269
	  - fish cutlet: 0
	  - basil potatoes: 0.163
	  - salad: 0
	- Tray 8
	- Leftover 1
	  - pasta + clams&mus: 0.227
	  - seafood salad: 0.317
	  - beans: 0.259
	  - basil potatoes: 0
	  - salad: 0.417
	- Leftover 2
	  - pasta + clams&mus: 0.554
	  - seafood salad: 1.311
	  - beans: 1.275
	  - basil potatoes: 0
	  - salad: 1.120
	- Leftover 3
	  - pasta + clams&mus: 0.260
	  - seafood salad: 0
	  - beans: 0
	  - basil potatoes: 0
	  - salad: 0

Here there are all the graphical results for all the images in the given dataset (masks are histogram-equalized to be more visible), note that sometimes some plates in the ‘before’ images gets “dirty” from other near food bounding boxes, so in the ‘after’ image also there are some wrong pixels (since the tray format and its plate compositions are respected) and those bounding boxes results bigger that they should be:



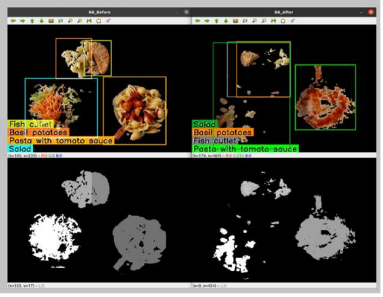
Tray 2

Leftover 1



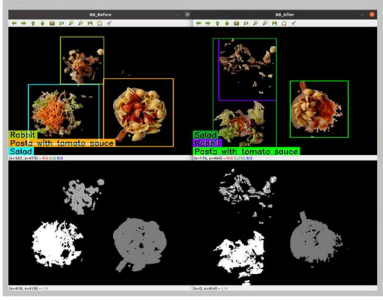
Leftover 2

Leftover 3

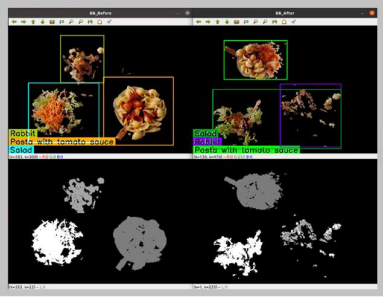


Tray 3

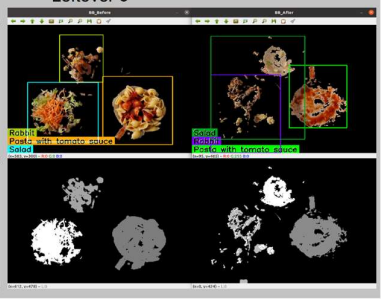
Leftover 1



Leftover 2

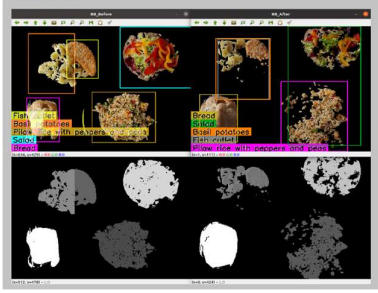


Leftover 3

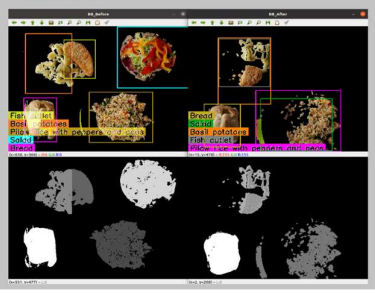


Tray 4

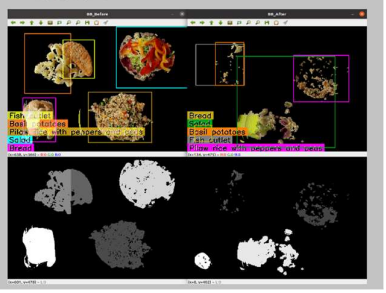
Leftover 1



Leftover 2

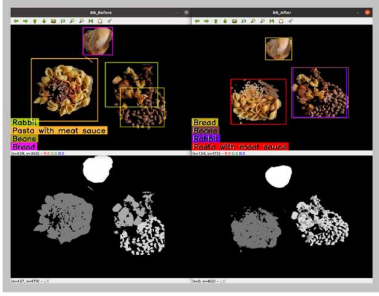


Leftover 3

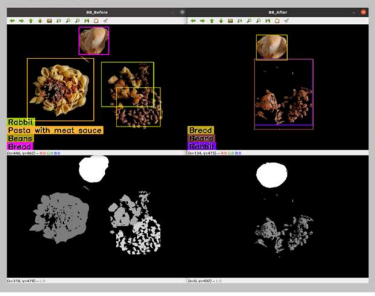


Tray 5

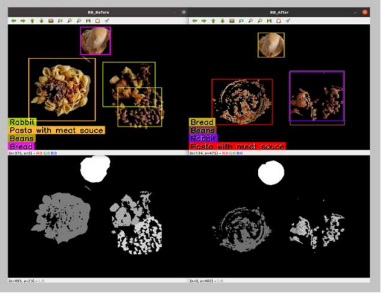
Leftover 1



Leftover 2

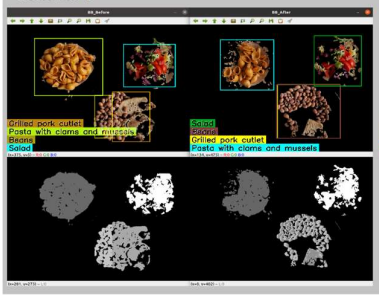


Leftover 3

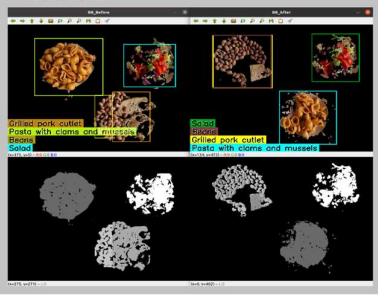


Tray 6

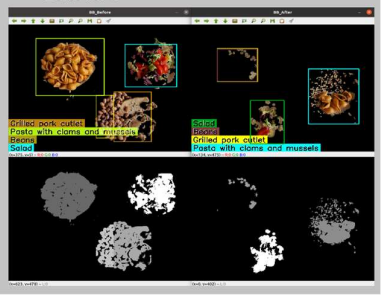
Leftover 1



Leftover 2

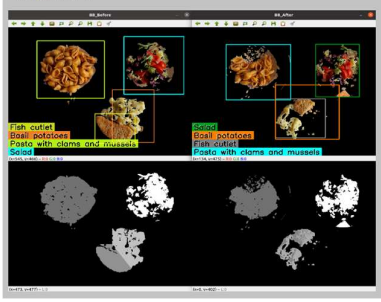


Leftover 3

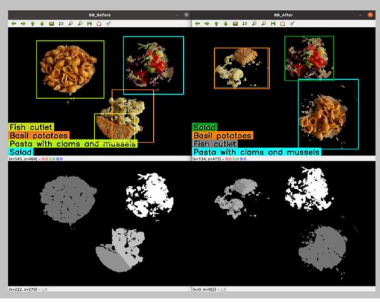


Tray 7

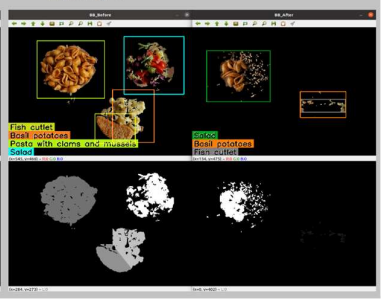
Leftover 1



Leftover 2

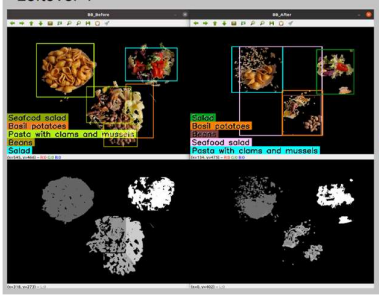


Leftover 3

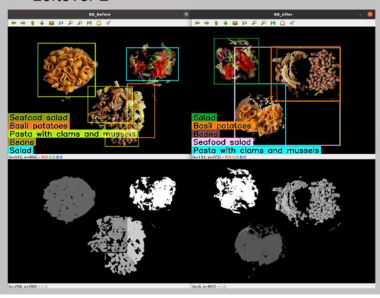


Tray 8

Leftover 1



Leftover 2



Leftover 3