COMPUTER VISION 2020/2021

Francesko Hyso
(ID: 1190475)

July 17, 2021

Computer vision project: boat detector

The boat detection problem has been tackled realizing a boat detector which employs a sliding window and performs a binary classification between boat and non-boat exploiting the bag of words (BoWs) method and an SVM classifier. In particular, the idea is that, given an input image, for each position taken by the sliding window we consider the SIFT features included in the window and we use them to compute the BoW of the window, and then, according to this BoW, we classify through an SVM the window as a boat or a non-boat. Moreover, in order to be able to detect boats of different scales, the search is performed over the images of a Gaussian pyramid generated from the input image. Finally, the whole boat detector is composed of two parts reflecting two different phases: an offline part, where essentially we build a codebook and we train the SVM classifier, and an online part, where we take in input an image and we analyse it through the sliding window.

Offline phase

Codebook generation and descriptor voting

The codebook has been generated fixing a size of 12 500 visual words where the latter have been obtained clustering, through k-means, the SIFT descriptors extracted from a dataset of 19 917 images having different sizes and among which 3100 were positive examples[1] and 16 817 where negative examples. In particular, the idea was that the codebook should be created using also negative examples such as water, sky, buildings, etc, in order to produce also some visual words that can be used to describe (at least) the typical negative windows that could be encountered in boat detection (such as only water), and this is helpful because of the following reasoning.

Let's suppose we are computing the BoW of a certain window in the online phase. Then, to each SIFT descriptor in the window we need to assign a certain visual word, namely, in other terms, each descriptor should vote a certain visual word. To do so, the first idea may be to let each descriptor vote for its closest visual word in ℓ_2-norm. However, doing so, if the codebook contains only visual words expressing boat features, then we force also a feature not related to a boat to vote for a boat feature, which is not a good choice. So, in order to avoid this problem one idea could be to keep the codebook only with "boat visual words" but to allow a descriptor to vote only if the distance from its closest visual word is less than a certain threshold because intuitively a non-boat descriptor should be far from the boat visual words, so we could just forbid such descriptor to vote. Unfortunately though, once tried, this approach in practice resulted impracticable due to a

[1]Note: the positive examples where mainly those provided by the Kaggle and the MAR dataset.

problem related to the SVM, namely since the ideal situation with this method is that a non-boat window in the online phase produces a BoW with all zeros, then, in order to separate the "zero" BoWs from the "non-zero" BoWs, the correct SVM type to use seemed the "ONE CLASS" type in OpenCV, however, on a practical level, with the possessed knowledge, the "ONE CLASS" type resulted too hard to tune because there was not an automatic training able to find good parameters and proceeding trough trial and error was not feasible due to the quite high number of parameters to fix and the non-negligible amount of time required for the training.

So, an alternative approach to manage the codebook generation and the descriptor voting is the following: we use a codebook having both boat visual words and some non-boat visual words, and then we let a descriptor voting for its closest visual word if its satisfies both a threshold-based matching and a nearest neighbour distance ratio matching, namely a descriptor $\mathbf{x}$ votes for its (first) closest visual word $\mathbf{w}_1$ if:

(1) $||\mathbf{x} - \mathbf{w}_1||_2 < t_1$ with t_1 arbitrary threshold;

(2) $\frac{||\mathbf{x} - \mathbf{w}_1||_2}{||\mathbf{x} - \mathbf{w}_2||_2} < t_2$ with t_2 arbitrary threshold and $\mathbf{w}_2$ second closest visual word to $\mathbf{x}$;

and in this way through (1) we are able to filter out some descriptors and through (2) we are able to let to vote only the descriptors less uncertain on their closest visual word. On a practical level two reasonable thresholds are $t_1 = 1000$ and $t_2 = 0.95$.
Finally, as a consequence of the codebook generation and the descriptor voting mechanism we will use as SVM an SVM of type "C_SVC" in OpenCV and we will train it with both positive and negative examples.

SVM Training

As already mentioned, in order to perform the window classification we use an SVM of type "C_SVC" which in particular employs an "RBF" kernel. On a practical level the SVM has been trained using the OpenCV function `trainAuto()` which has been fed taking as positive and negative examples the BoWs of the images of the same dataset used for generating the codebook.

Online phase

In the online phase the main action performed is the sliding window procedure and in order to realize this action the window size has been fixed to 100×100 pixels while the shift step has been fixed to 10 pixels.

The online phase pipeline

The online phase is essentially composed by the following steps:

(1) input image acquisition;

(2) preliminary resizing: said w and h respectively the width and the height of the input image, we resize the latter to be $N \times N$ pixels with N obtained as follows: if both w and h are greater than 600 then $N = 800$ in order to implement a 3-level Gaussian pyramid with images of predefined dimension $800 \times 800, 400 \times 400$ and 200×200, otherwise if both w and h are greater than 300 then $N = 400$ in order to implement a 2-level Gaussian pyramid with images of predefined dimension $400 \times 400, 200 \times 200$, otherwise $N = 200$;

(3) preliminary bilinear filtering: we compute a first time the SIFT features of the input image and, said n the number of features, we smooth the input image with a bilinear filter with diameter $= \sigma_r = \sigma_s = \frac{n}{43}$ if $n <= 7000$ or diameter $= \sigma_r = \sigma_s = \frac{n}{32}$ if $n > 7000$ because

in this way we are able to remove some "weak" features among which the majority often belongs to regions containing for instance water;

At this point the next step is to perform the sliding window procedure, and since it is know that the sliding window is intrinsically not efficient, in order to reduce the computational burned one first implementation could be the following: we observe that we have three types of shifts:

(1) right shifts of the first row: considering Figure 1, in these shifts we just need to discard the features of the rectangle 1, to keep those of the rectangle 2 and to compute the features in the rectangle 3;
(2) down shifts: referring to Figure 2, we have a situation similar to the previous point;
(3) right shifts of the rows after the first: considering Figure 3, in these cases we have to discard the features of rectangle 1, to keep those of rectangle 2, to recover from memory those of rectangle 3 (which have been already computed in the right shifts of the previous row) and to compute the ones of rectangle 4.

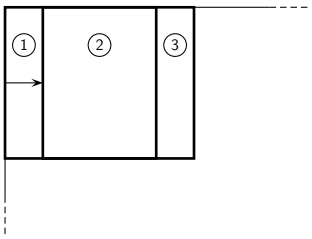

Figure 1

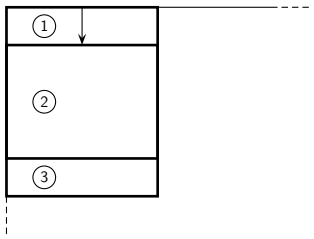

Figure 2

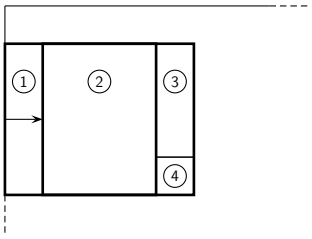

Figure 3

Now, although this implementation makes a tentative of optimization, actually it is not really effective because once obtained all the features of a window we still have to compute the BoW of the window online, namely during the sliding of the window, and in regions with many features this slows down the sliding significantly. So, an alternative better implementation, which is the one realized, approaches the problem from another viewpoint. Considering Figure 4, we continue the online pipeline as follows:

(4) SIFT feature extraction: we extract the definitive SIFT features from the smoothed image;
(5) image division in blocks of size 10×10: we take the smoothed image and we divide it in blocks of size 10×10 pixels, namely in practice we create a vector $\mathbf{v}$ whose elements refer each to one image block considering the blocks from the top to the bottom from left to right;
(6) feature assignment to the block in which they lie: for each feature extracted in (4), we assign it to the block in which the feature lies;
(7) computation of the BoW of each block: for each block of index k in $\mathbf{v}$ we compute its BoW $\mathbf{b}_k$, with $k = 0, 1, 2, ..., n(n-1) + n - 1 = n^2 - 1$;
(8) we define as a sliding window the list W of the indexes of the 100 image blocks currently covered by the window and taken in $\mathbf{v}$. For instance, the first window, namely the one with top-left corner in block $(i, j) = (0, 0)$, is

$$W_{0,0} = \{ \underbrace{0, n, 2n, ..., 9n}_{\substack{\text{indexes in } \mathbf{v} \text{ of the} \\ \text{blocks covered by the} \\ \text{1st column of the window}}} , \underbrace{1, n+1, 2n+1, ..., 9n+1}_{\substack{\text{indexes in } \mathbf{v} \text{ of the} \\ \text{blocks covered by the} \\ \text{2nd column of the window}}}, ..., \underbrace{9, n+9, 2n+9, ..., 9n+9}_{\substack{\text{indexes in } \mathbf{v} \text{ of the} \\ \text{blocks covered by the} \\ \text{10th column of the window}}} \},$$

where notice the indexes are taken considering the blocks of the window from left to right from the top to the bottom. This organization is more natural for the updating of the list W because since the majority of the shifts are right shifts, with this organization in such shifts we can just discard the first column of the window and add a new column.

(9) at this point we observe that the BoW $\mathbf{B}_{i,j}$ of a window $W_{i,j}$ is nothing but the sum of the BoWs of the blocks of $W_{i,j}$, namely

$$\mathbf{B}_{i,j} = \sum_{k \in W_{i,j}} \mathbf{b}_k,$$

and exploiting this fact we can avoid to first recover all the SIFT features and then to compute the BoWs starting from the descriptors because the majority of the computations can be performed preliminarily.

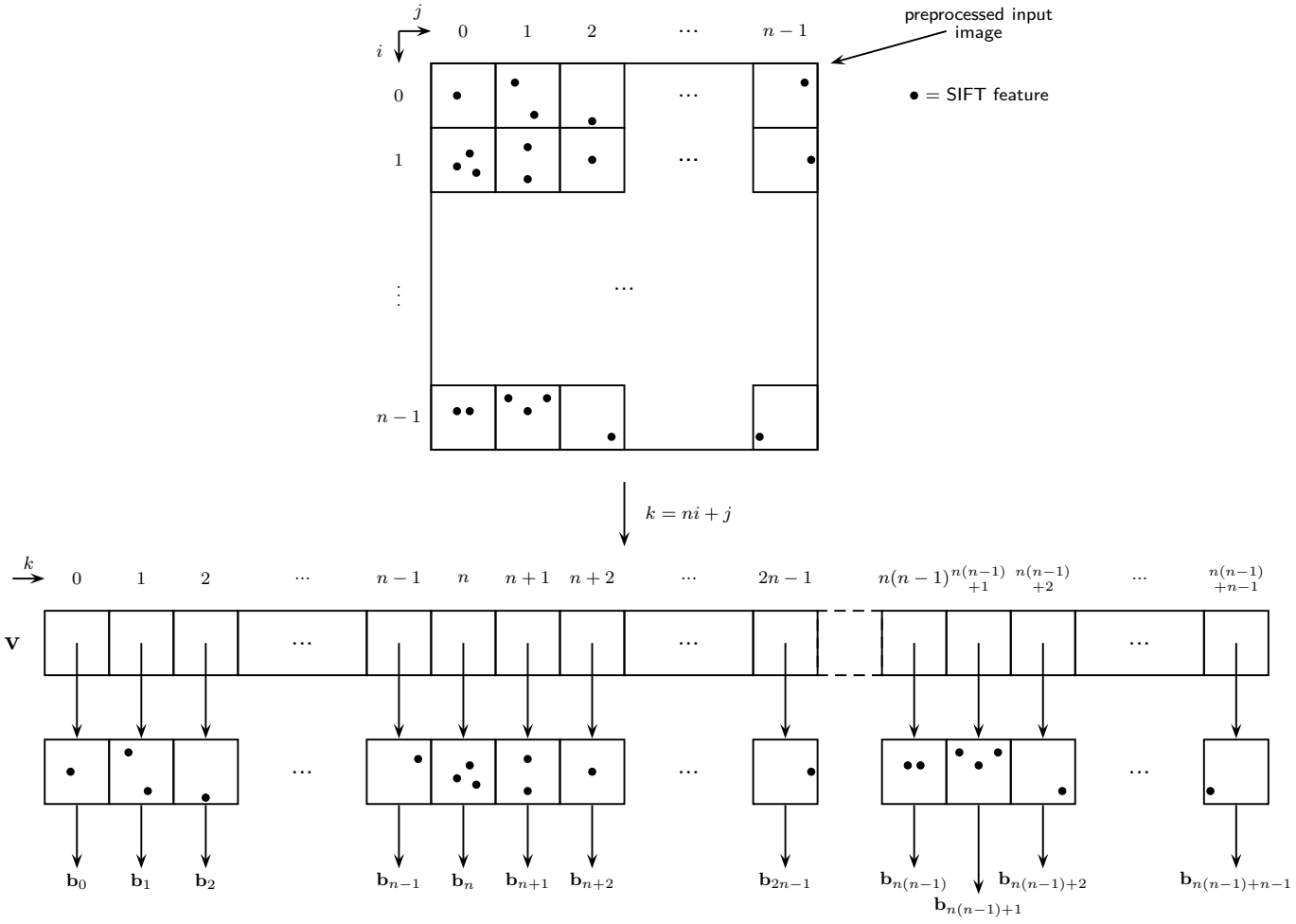

Figure 4: Preliminary operations for the sliding window procedure.

(10) launch of the sliding window procedure: we perform the window sliding and the classification;

(11) at the end of the window sliding, among all the detected boxes, we merge those with overlap greater or equal than the 20% of their area computing the smallest box containing all of them. To do so we exploit the following idea: considering Figure 5 we observe that a set of overlapping boxes (of same size) is equivalent to a (an undirected) graph where each node represents a box and each connection represents an overlapping $\geq 20\%$, so, given for each box the indexes (in $\mathbf{v}$) of the boxes with which the considered box has overlap $\geq 20\%$, the computation of the groups of overlapping boxes is a set of boxes boils down to the computation of the connected component of the graph associated to the set boxes;

(12) we refine the boxes resulting from (11) shrinking them in order to obtain the smallest boxes containing the same SIFT features;

(13) if there are levels of the image pyramids not analysed yet we downsample the current image and we repeat the procedure from step (4);
(14) we upsample all the final boxes extracted from each level;
(15) we merge the final boxes of a same level that have an overlap greater or equal than the 20%. Note: now the boxes may have different sizes but the idea of (11) can be generalized: we use now a directed graph where node i points to node j if the overlap between box i and j for box i is $\geq 20\%$ and where the aim is to find the weakly connected components.
(16) for the final boxes of the levels 2 and 3 we discard them if there are boxes of the lower levels which overlap with them;
(17) we merge the remaining boxes that have overlap $\geq 75\%$ for at least one of the involved boxes taking as result the smallest box containing all of them;
(18) we compute the union between the set of final detected boxes and the set of ground truth boxes, and then we compute the area of this set;
(19) we compute the intersection between the set of final detected boxes and the set of ground truth boxes, and then we compute the area of this set;
(20) we compute the IoU value;

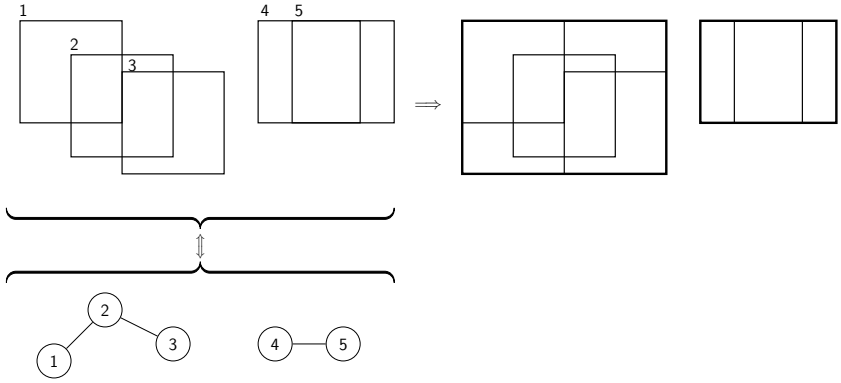

Figure 5

Summarizing, the whole boat detector is described in the scheme of Figures 6 and 7.

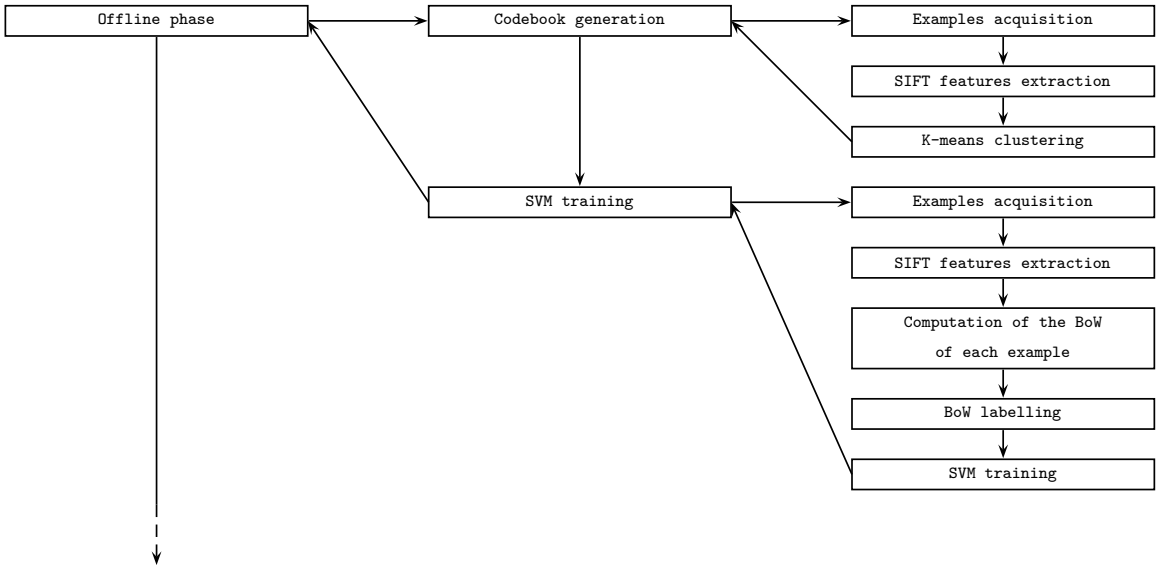

Figure 6: Boat detector scheme pt.1.

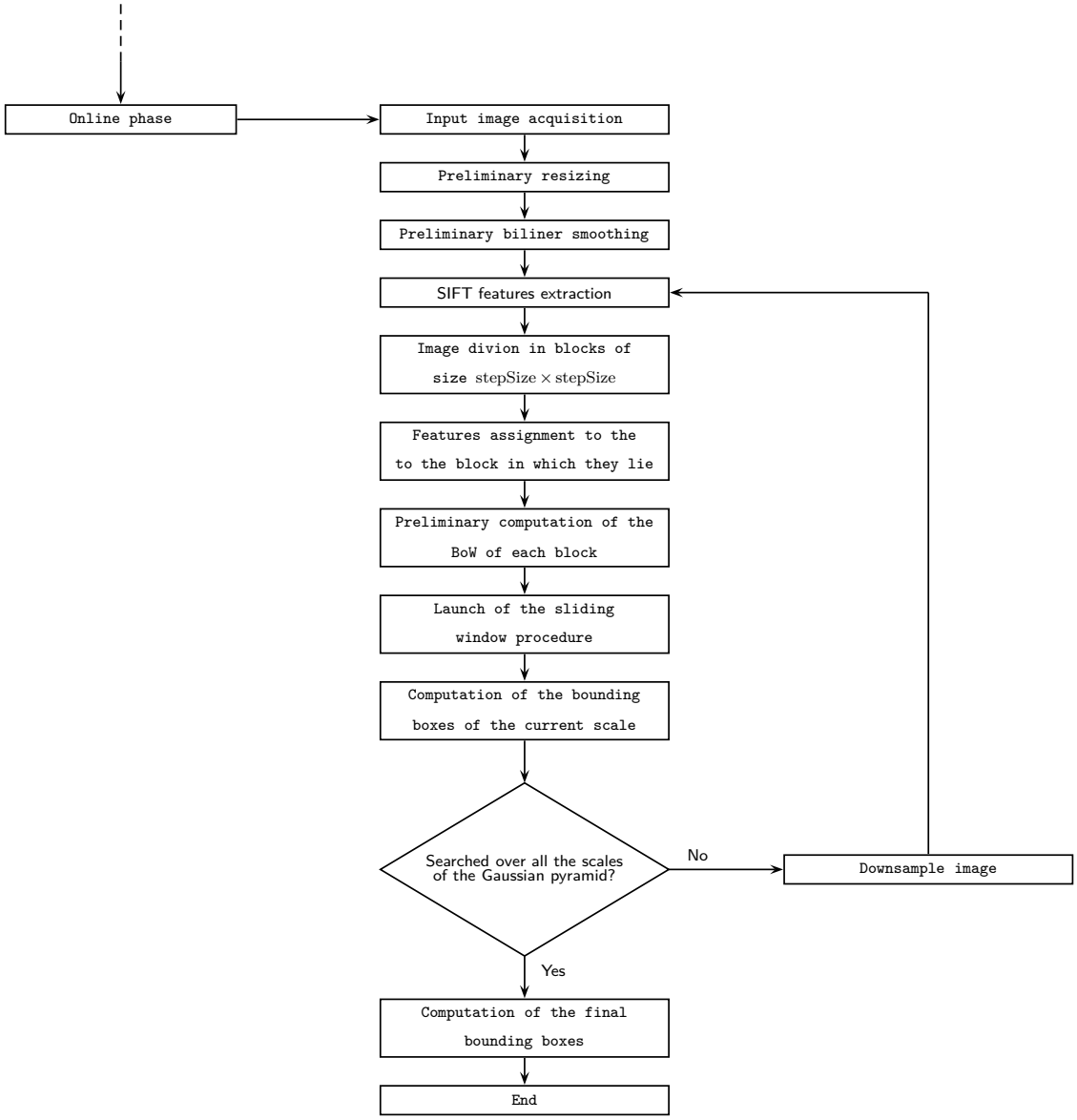

Figure 7: Boat detector scheme pt.2.

The code structure

The code is contained in the folder `Code` and is organized as follows:

(1) the file containing the main is `boatDetector.cpp`;

(2) there are 2 classes contained in the folder `Code\supportClasses`:

(2.1) `Codebook.cpp\.h`: this class is used for creating and storing the codebook;

(2.2) `Graph.cpp\.h`: this class is used for computing the connected components of the graph associated to a set of boxes, namely for computing the groups of overlapping boxes in the set of boxes;

In order to compile the code the idea is to run the command `cmake . -G "MinGW Makefiles"` from the folder `Code` so to produce here the `Makefile`, and then, always from the folder `Code`, to run also the command `mingw32-make` in order to produce the executable `boatDetector.exe`.

The ground truth convention

Both in the offline phase and in the online phase the boat detector requires some ground truth coordinates. The convention on the latter consists of writing the coordinates of the actual boxes of an image as follows:

imageName n x_1^{tl} y_1^{tl} w_1 h_1 x_2^{tl} y_2^{tl} w_2 h_2 ... x_n^{tl} y_n^{tl} w_n h_n

where:

(1) imageName: is the image name;
(2) n: is the number of ground truth boxes in the image;
(3) x_i^{tl} : is the x-coordinate of the top left corner of the box i;
(3) y_i^{tl} : is the y-coordinate of the top left corner of the box i;
(3) w_i : is the width of the box i;
(3) h_i : is the height of the box i;

Finally, the `.txt` file containing the ground truth coordinates should end with a single carriage return character after the boxes' coordinates of the last image.
Note: since for building the codebook we provide a dataset which has also negative examples, then the ground truth coordinates of these images are of the type:

imageName 1 0 0 w h

where:

(2) n is formally equal to 1 even if there is no boat;
(3) $x_1^{tl} = y_1^{tl} = 0$ in order to take as top left corner $(0, 0)$;
(3) $w_1 = w$ where w is the image width;
(3) $h_1 = h$ where h is the image height;

and in this way we just load the image without any extraction of a subimage representing a boat.
Example: we have the following examples:

(1) the ground truth coordinates of a dataset of m positive examples and l negative examples for generating the codebook should be as follows:

posImg_1 n_1 $x_{1,1}^{tl}$ $y_{1,1}^{tl}$ $w_{1,1}$ $h_{1,1}$ $x_{1,2}^{tl}$ $y_{1,2}^{tl}$ $w_{1,2}$ $h_{1,2}$ ... x_{1,n_1}^{tl} y_{1,n_1}^{tl} w_{1,n_1} h_{1,n_1}
posImg_2 n_2 $x_{2,1}^{tl}$ $y_{2,1}^{tl}$ $w_{2,1}$ $h_{2,1}$ $x_{2,2}^{tl}$ $y_{2,2}^{tl}$ $w_{2,2}$ $h_{2,2}$ ... x_{2,n_2}^{tl} y_{2,n_2}^{tl} w_{2,n_2} h_{2,n_2}
$\vdots$
posImg_m n_m $x_{m,1}^{tl}$ $y_{m,1}^{tl}$ $w_{m,1}$ $h_{m,1}$ $x_{m,2}^{tl}$ $y_{m,2}^{tl}$ $w_{m,2}$ $h_{m,2}$ ... x_{m,n_m}^{tl} y_{m,n_m}^{tl} w_{m,n_m} h_{m,n_m}
negImg_1 1 0 0 w_1 h_1
negImg_2 1 0 0 w_2 h_2
$\vdots$
negImg_l 1 0 0 w_l h_l

(2) the ground truth coordinates of a single image for the online phase in general consists of only one row as follows:

img n x_1^{tl} y_1^{tl} w_1 h_1 x_2^{tl} y_2^{tl} w_2 h_2 ... x_n^{tl} y_n^{tl} w_n h_n

and if the image doesn't contain any boat then the ground truth coordinates become just:

img 0.

Note: notice that the ground truth convention for the negative images is different between offline and online phase because in the offline phase even if there is no boat we still want to use the image while in the online phase we have no boxes to draw.

Example of execution of the online phase

In Figures 8-11 is reported an example of execution of the boat detector:

(1) Figure 8: in this figure we can see that when there are no features in the current window the latter is drawn in green;
(2) Figure 9: here we see that when there are features in the current window the latter in drawn in blue and the current features in cyan;
(3) Figure 10: in this figure we see that if a window is classified as boat then it is drawn permanently in dark red and its features in light blue;
(4) Figure 11: here is reported the final situation of the level-1 image of the Gaussian pyramid where the orange boxes are the final boxes of this level;

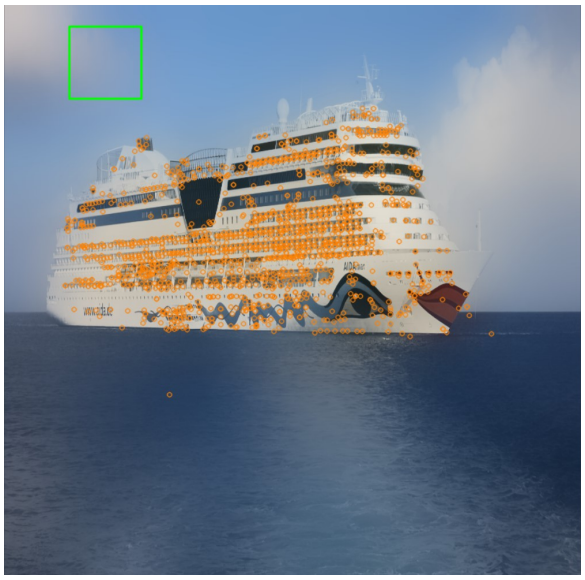

Figure 8

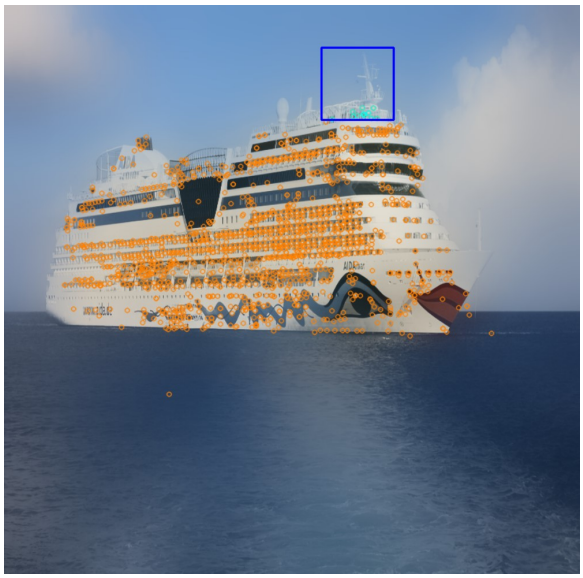

Figure 9

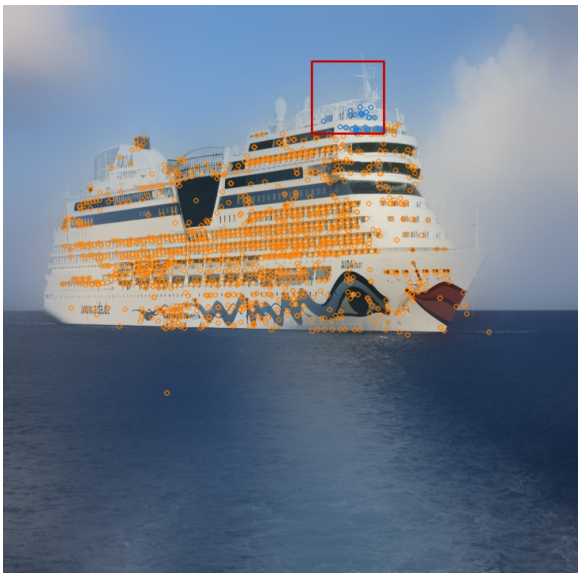

Figure 10

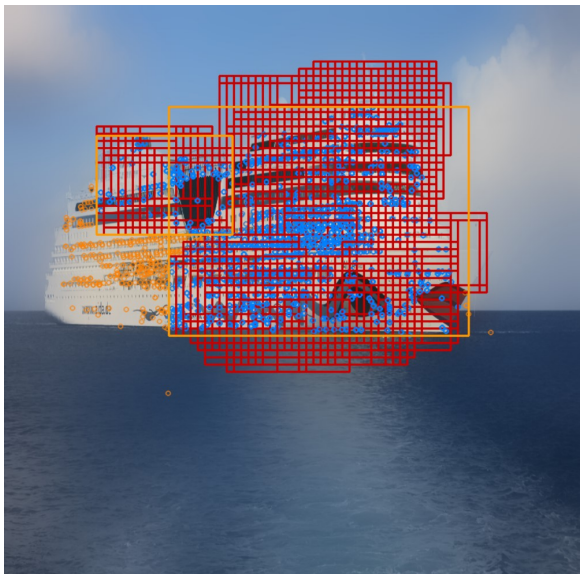

Figure 11

Finally, the boat detector produces as output four images (that are written in some .jpg files) such as those in Figures 12-15:

(1) Figure 12: in this figure are shown the final boxes of the three levels of the Gaussian pyramid: the orange are level-1 boxes, the purple is a level-2 box and the turquoise is a level-3 box;
(2) Figure 13: in this figure are reported the results of the mergers of the boxes of a same level with overlap $\geq 20\%$ for at least one of the involved boxes; here, for instance, since for the small orange box the overlap with the big orange box is $\geq 20\%$ of the small box area, then the two boxes are merged taking as result the smallest box containing both of them;
(3) Figure 14: here we have in green the ground truth box and in blue the final detected box.
(4) Figure 15: in this figure are shown respectively in red the union between the set of the detected boxes and the ground truth boxes and in yellow the intersection between the set of detected boxes and the set of the ground truth boxes.

Figure 12

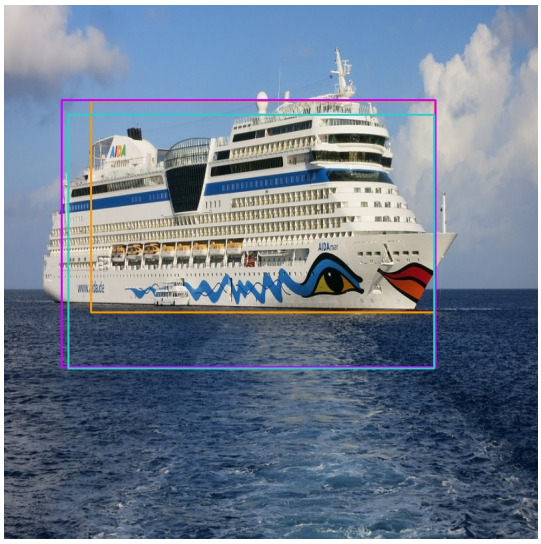

Figure 13

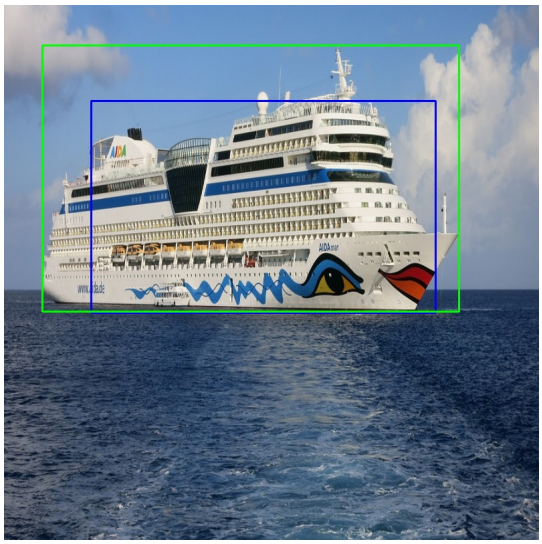

Figure 14

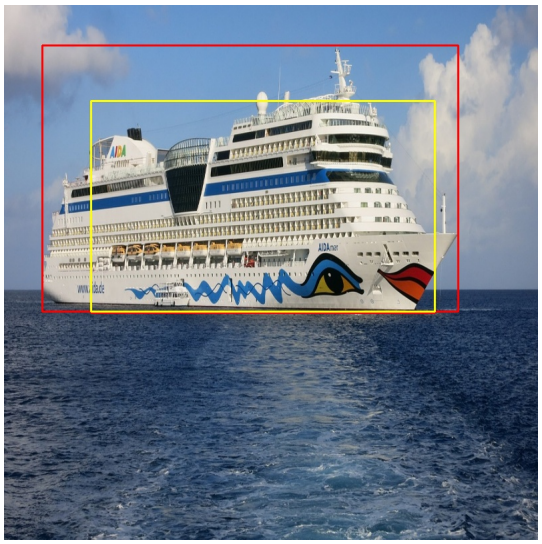

Figure 15

Test datasets result

Kaggle dataset results

We have the results shown from Figure 16 to Figure 35.

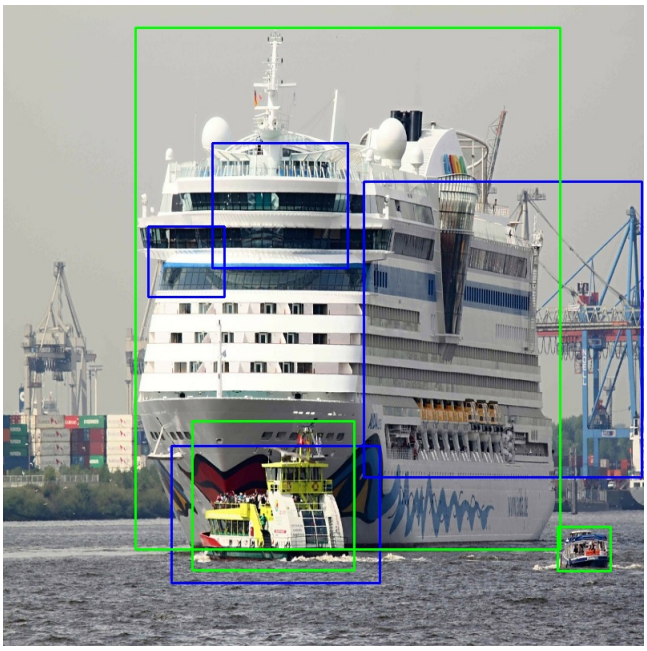

Figure 16: kaggleImg01.

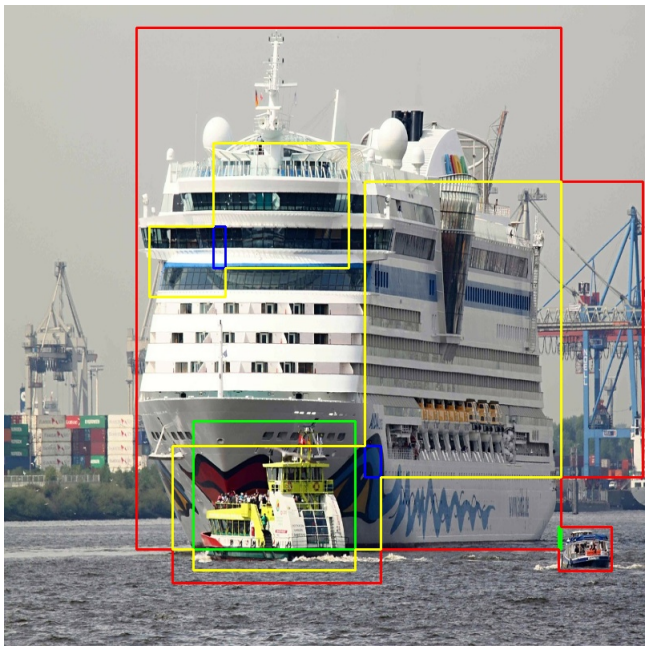

Figure 17: IoU = 0.409 417.

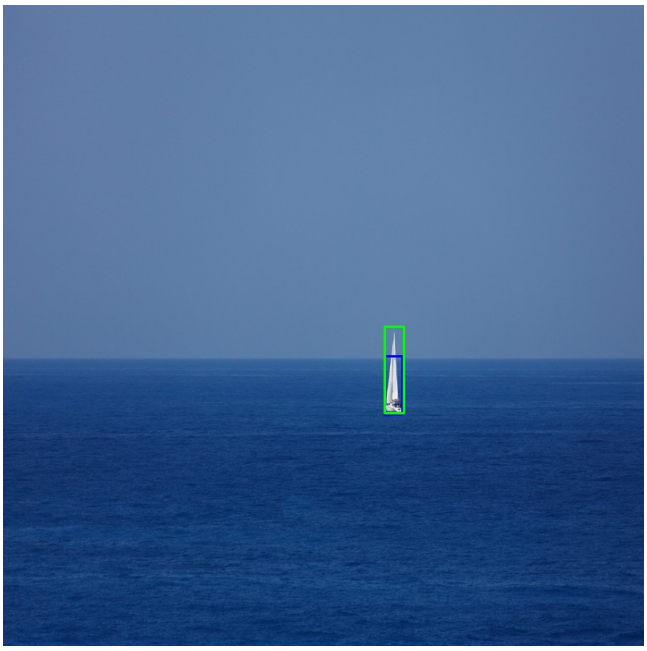

Figure 18: kaggleImg02.

Figure 19: IoU = 0.577 072.

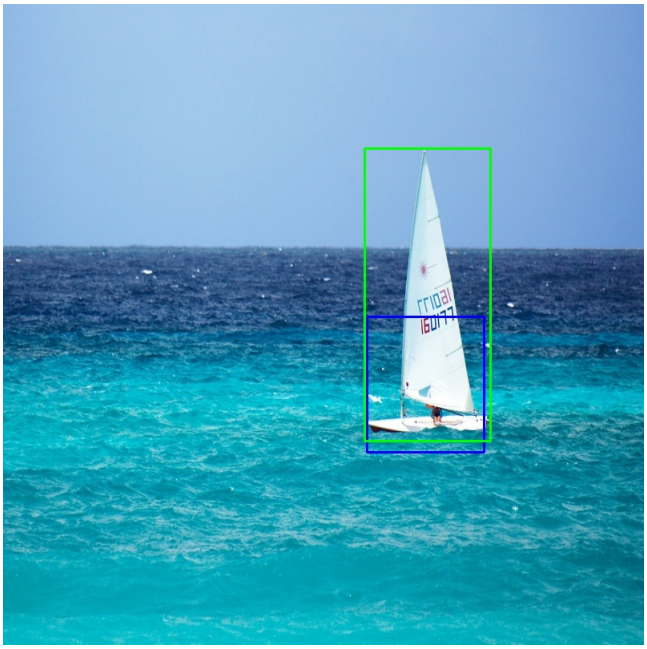

Figure 20: kaggleImg03.

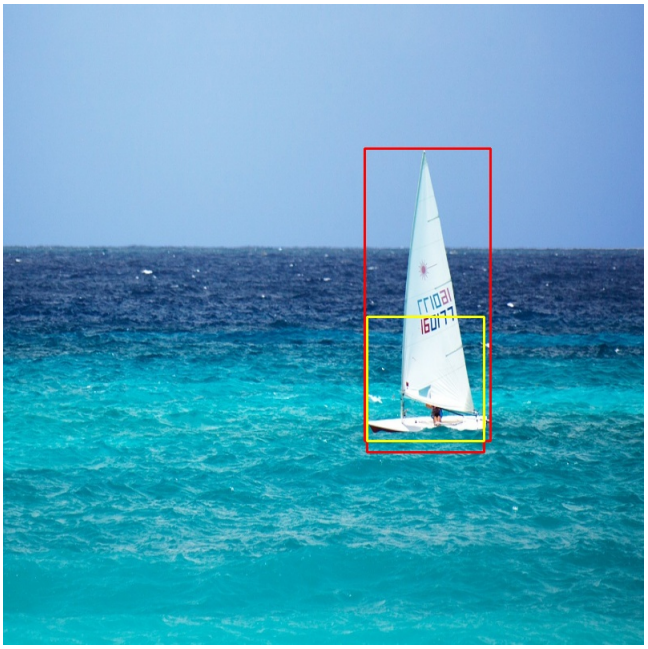

Figure 21: IoU = 0.383 81.

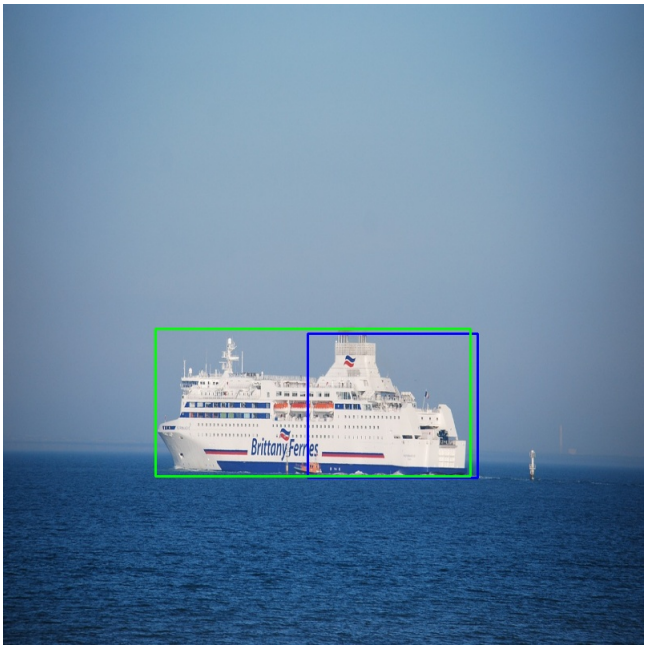

Figure 22: kaggleImg04.

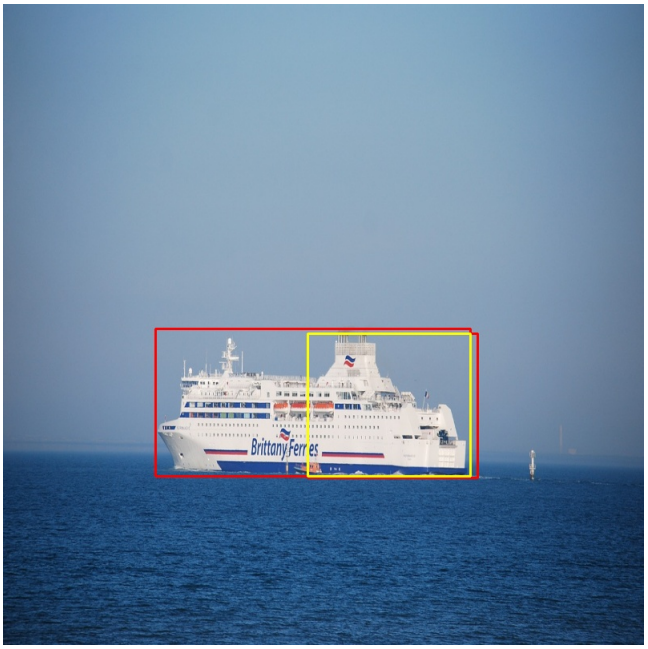

Figure 23: IoU = 0.489 083.

Figure 24: kaggleImg05.

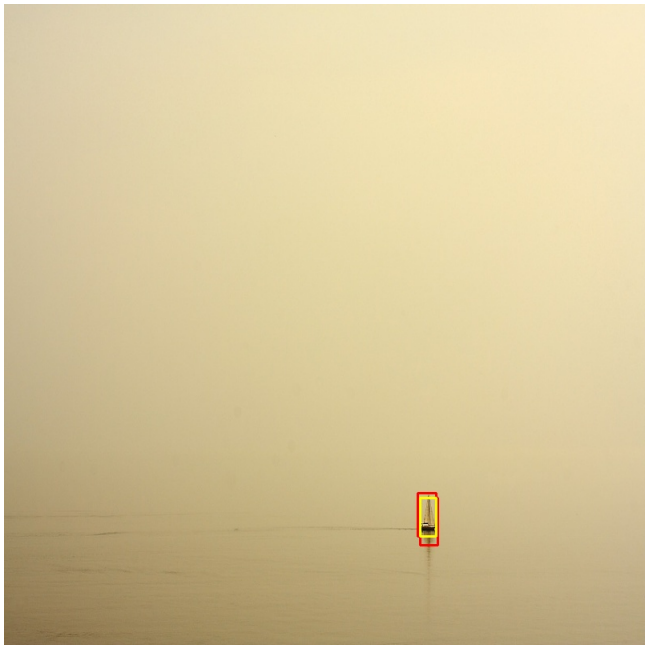

Figure 25: IoU = 0.635 05.

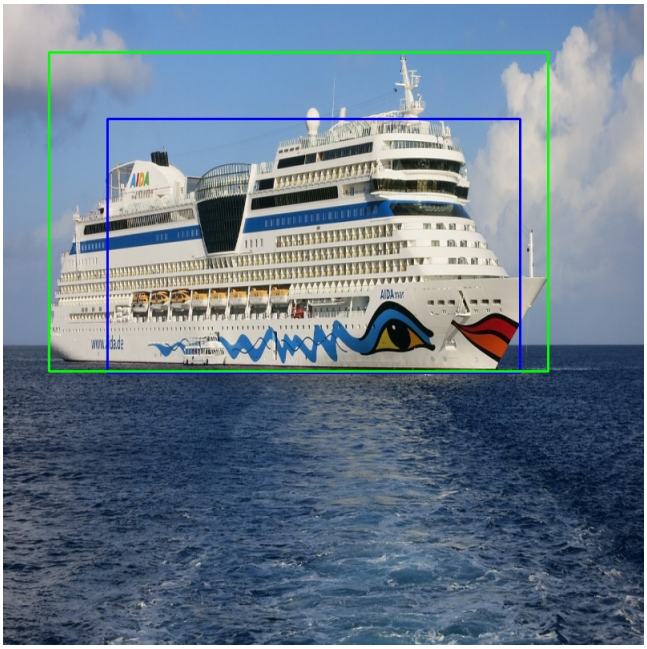

Figure 26: kaggleImg06.

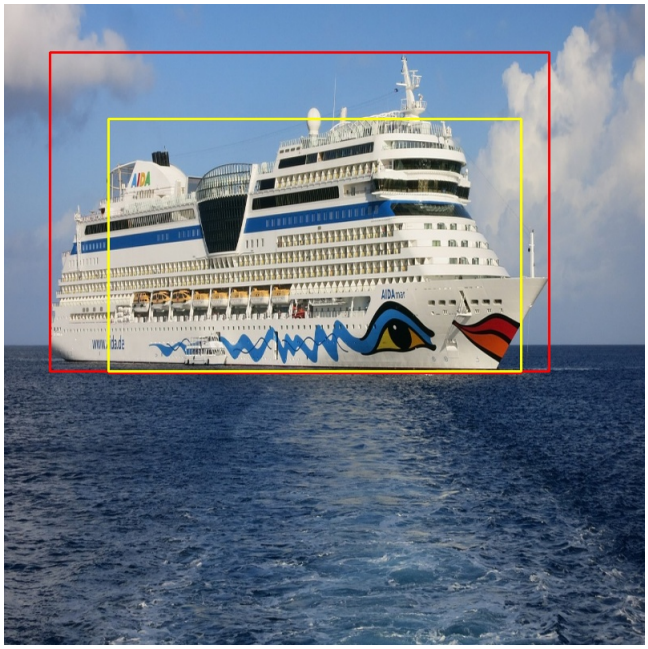

Figure 27: IoU = 0.653 004.

Figure 28: kaggleImg07.

Figure 29: IoU = 0.704 196.

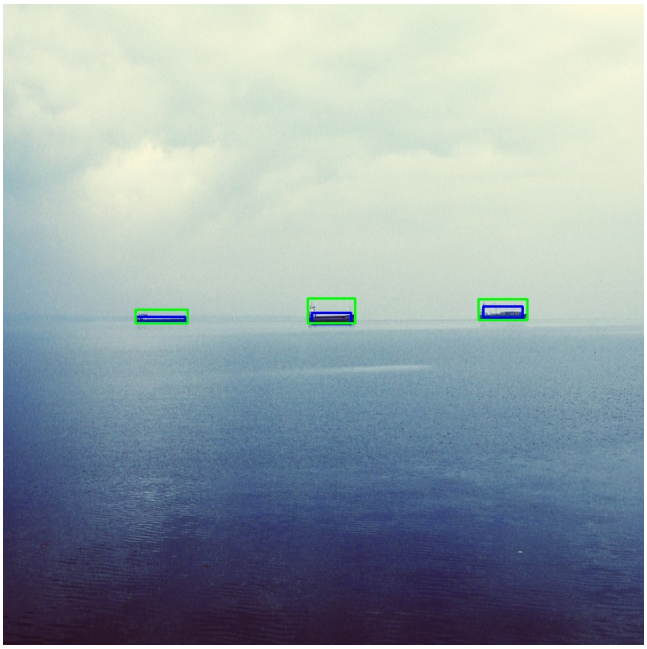

Figure 30: kaggleImg08.

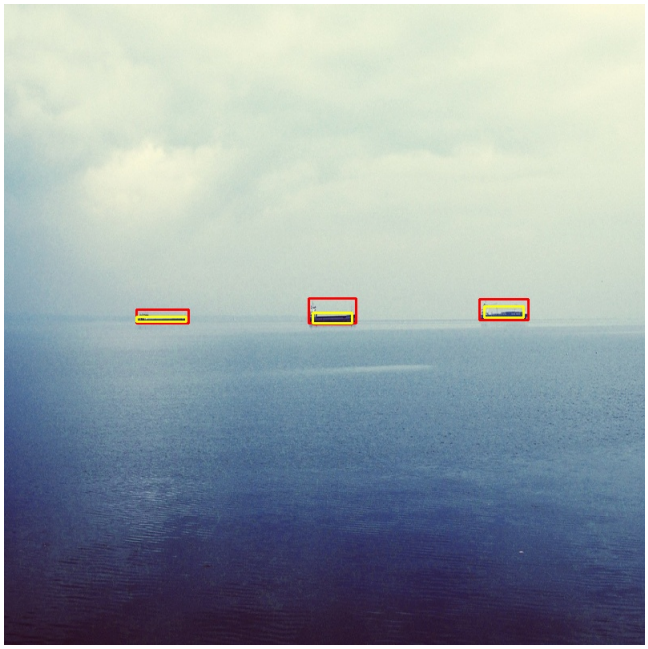

Figure 31: IoU = 0.370 114.

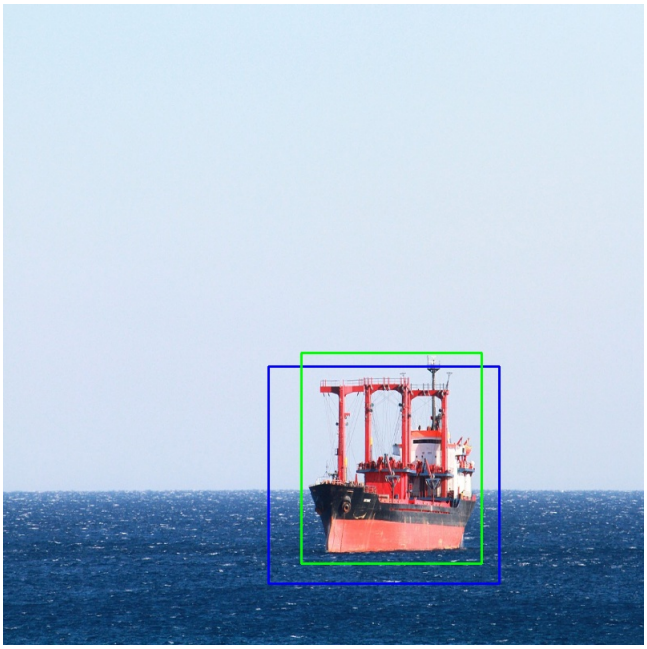

Figure 32: kaggleImg09.

Figure 33: IoU = 0.677 413.

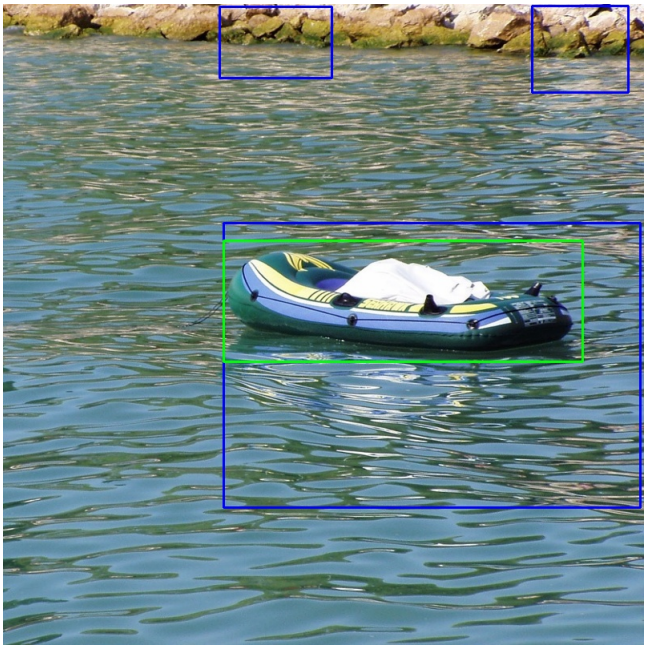

Figure 34: kaggleImg10.

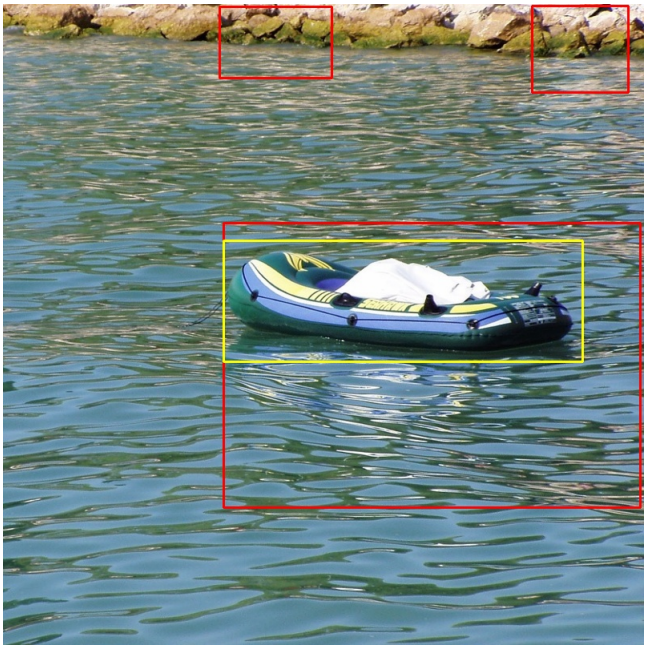

Figure 35: IoU = 0.323 58.

MAR dataset results

We have the results shown from Figure 36 to Figure 59.

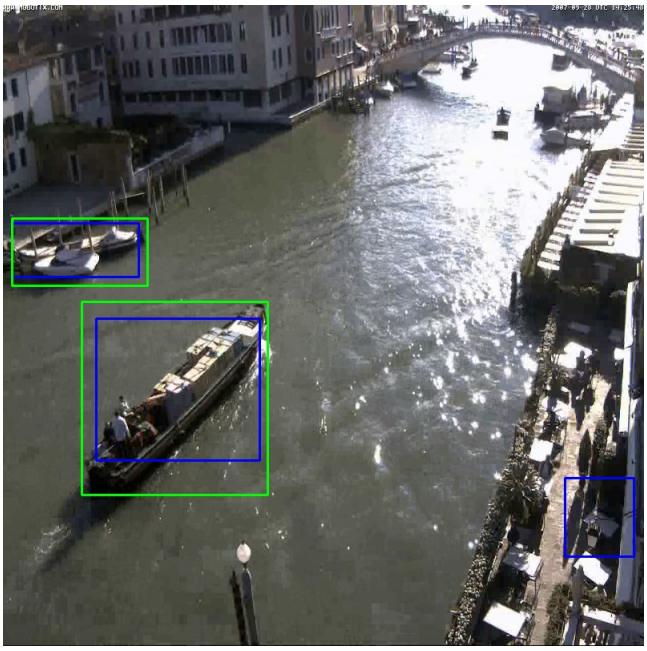

Figure 36: veniceImg01.

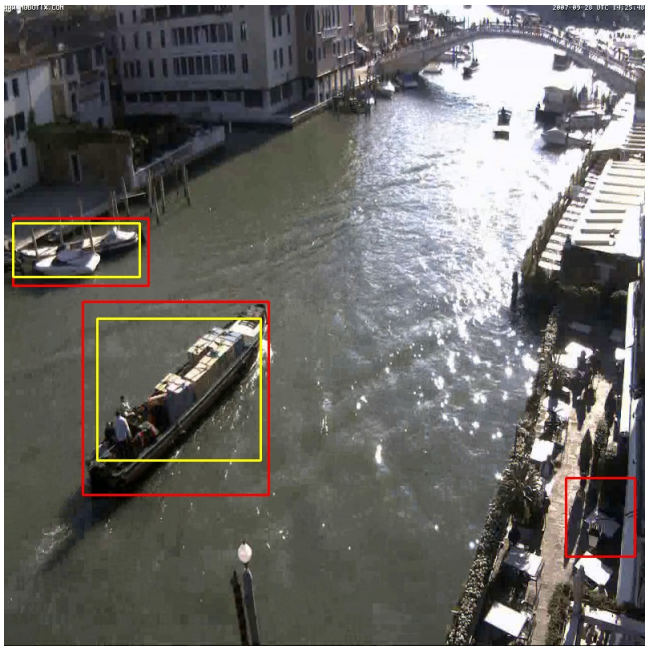

Figure 37: IoU = 0.592 751.

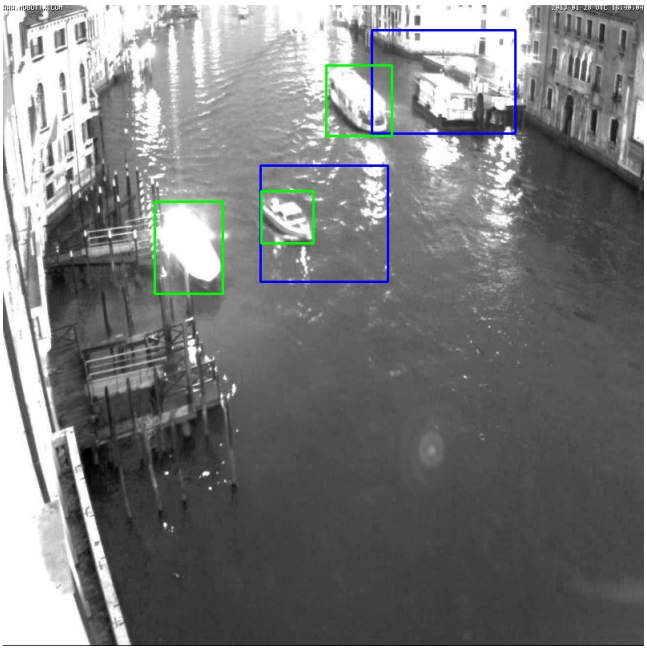

Figure 38: veniceImg02.

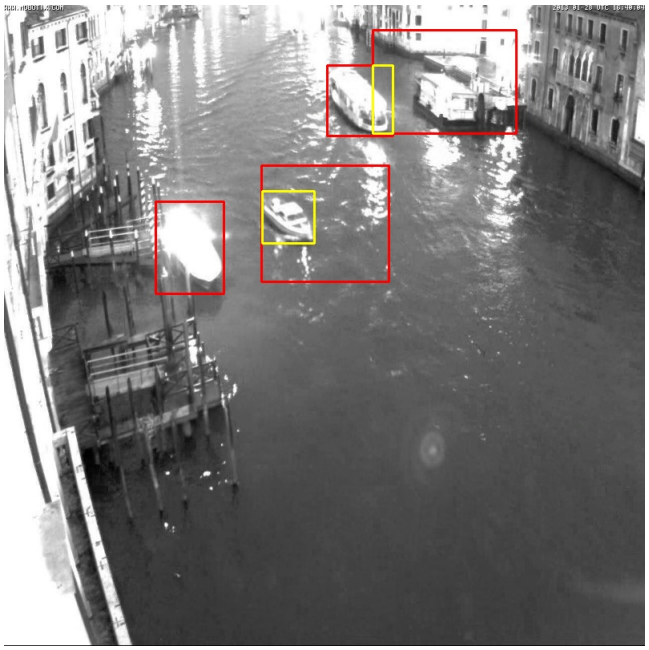

Figure 39: IoU = 0.104 05.

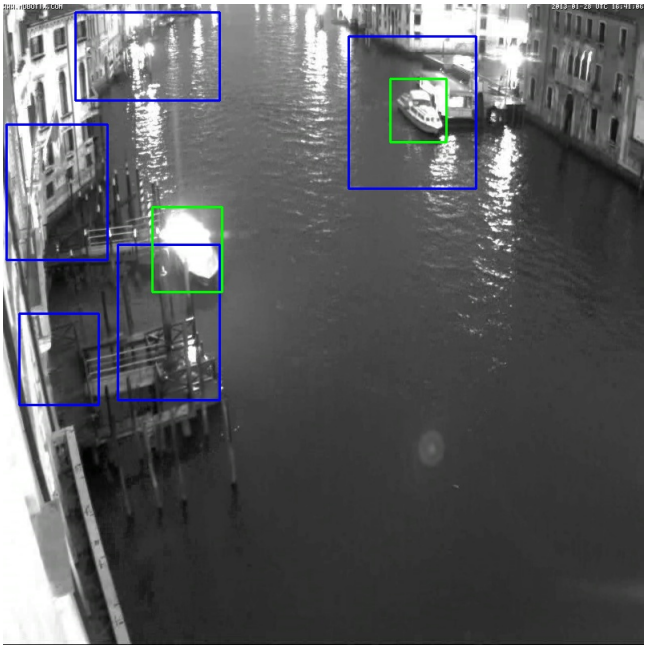

Figure 40: veniceImg03.

Figure 41: IoU = 0.093 469 6.

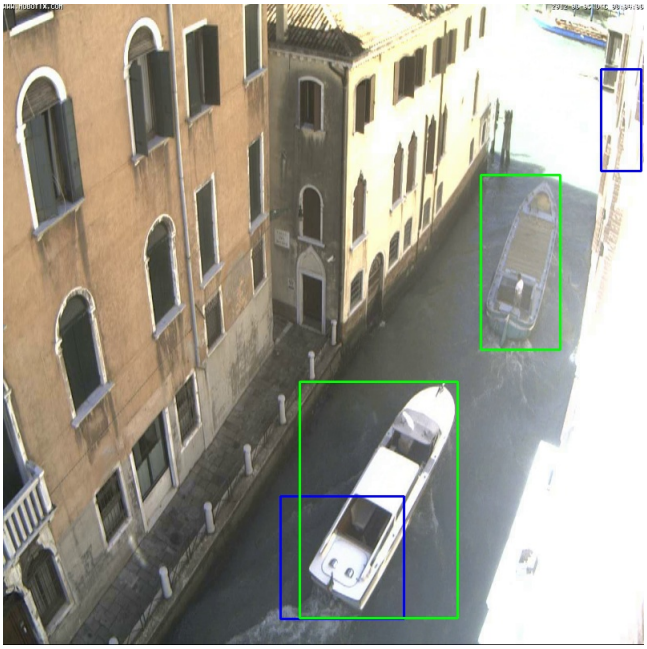

Figure 42: veniceImg04.

Figure 43: IoU = 0.220 793.

Figure 44: veniceImg05.

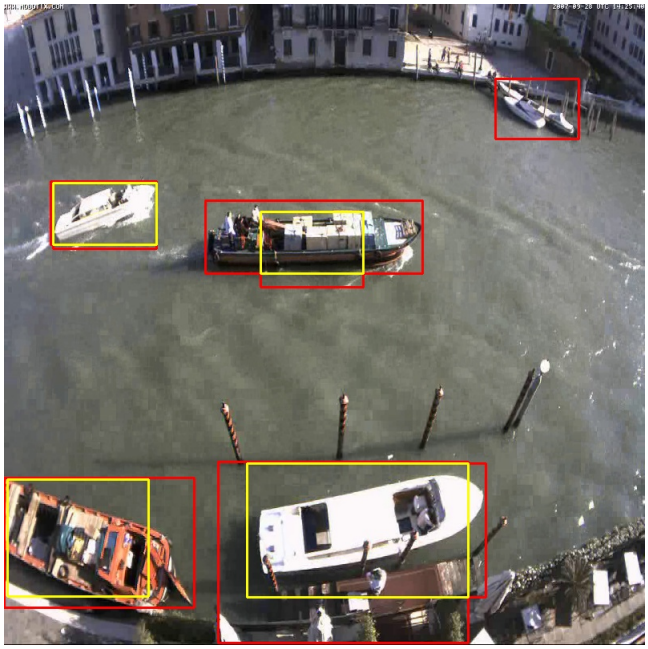

Figure 45: IoU = 0.577 753.

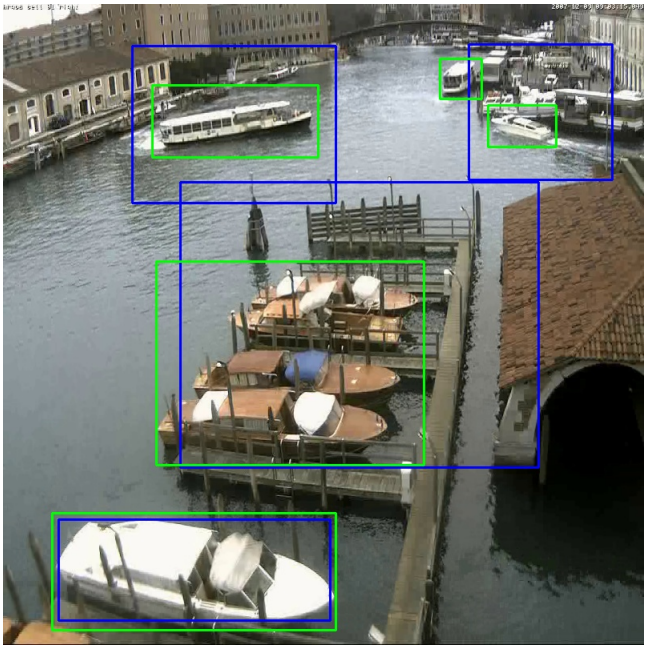

Figure 46: veniceImg06.

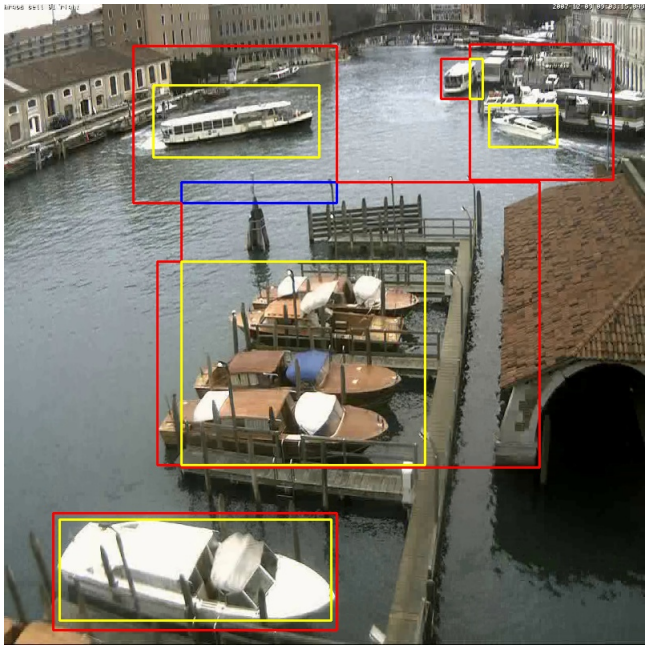

Figure 47: IoU = 0.486 279.

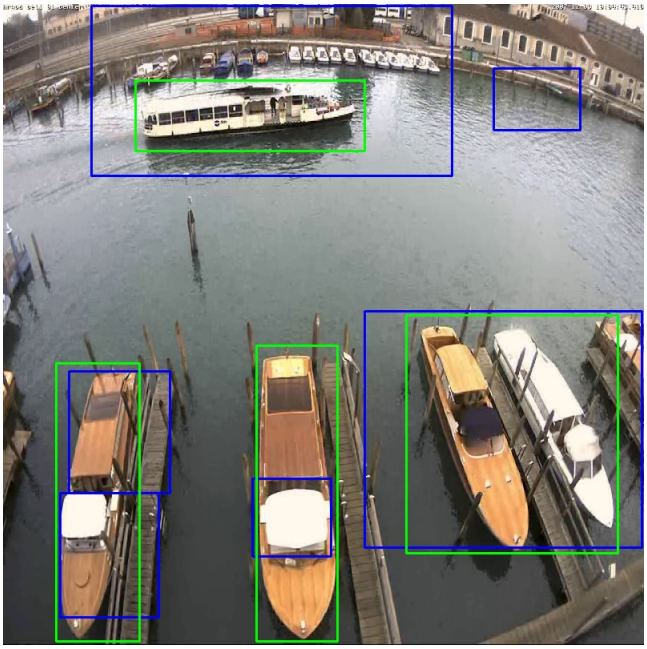

Figure 48: veniceImg07.

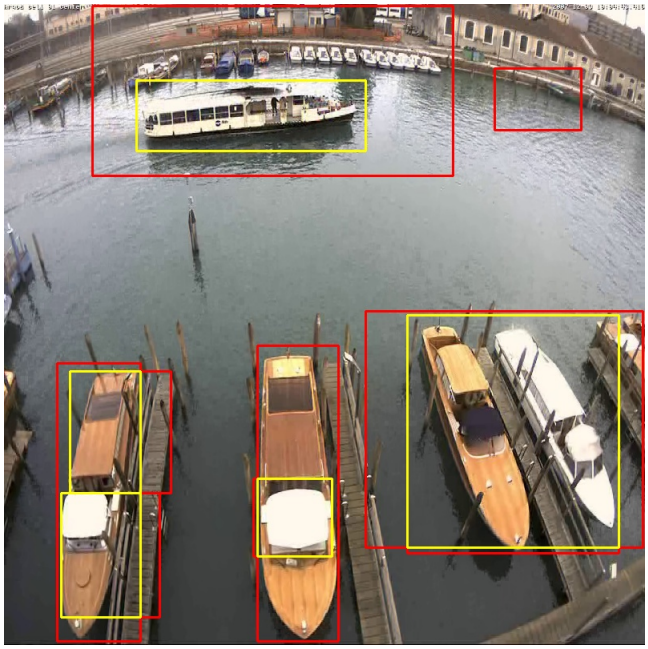

Figure 49: IoU = 0.479 083.

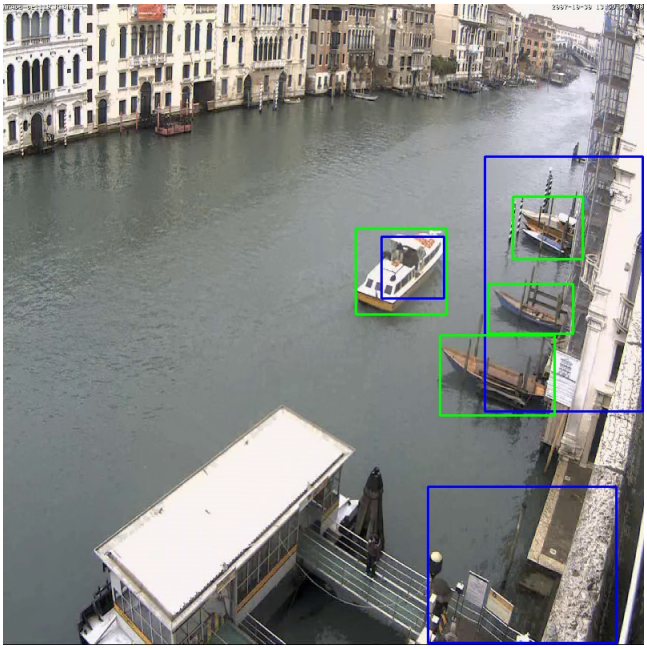

Figure 50: veniceImg08.

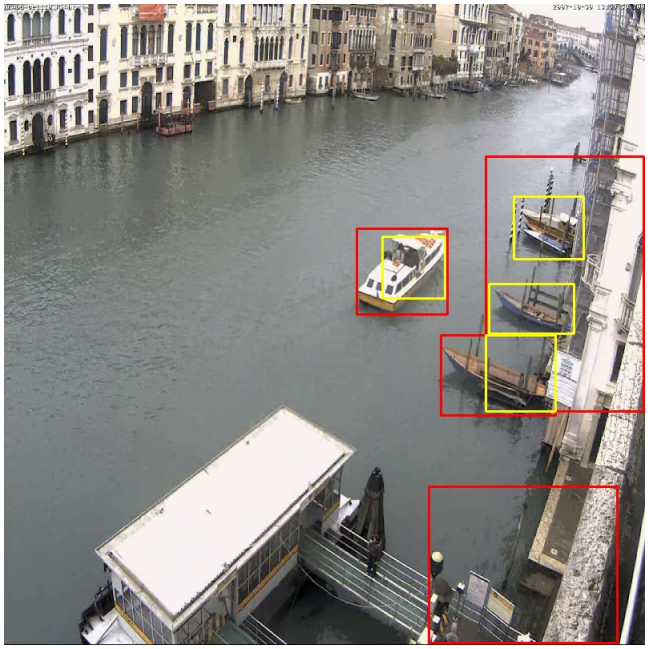

Figure 51: IoU = 0.219 363.

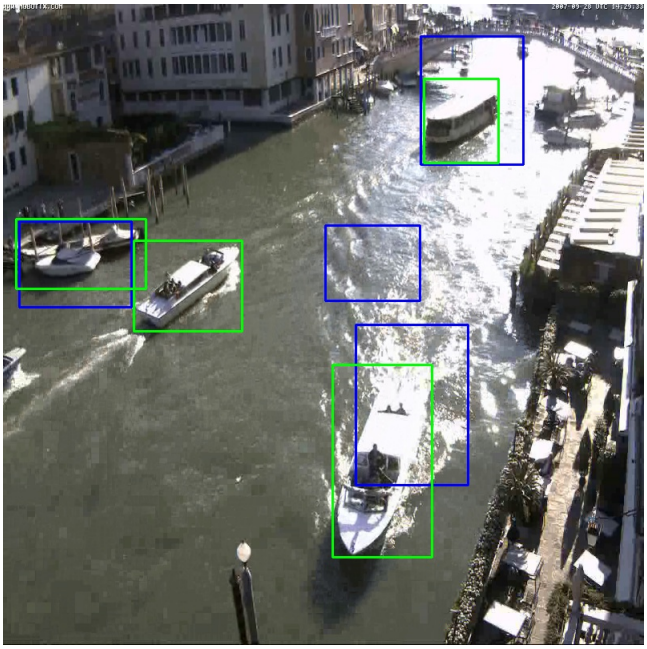

Figure 52: veniceImg09.

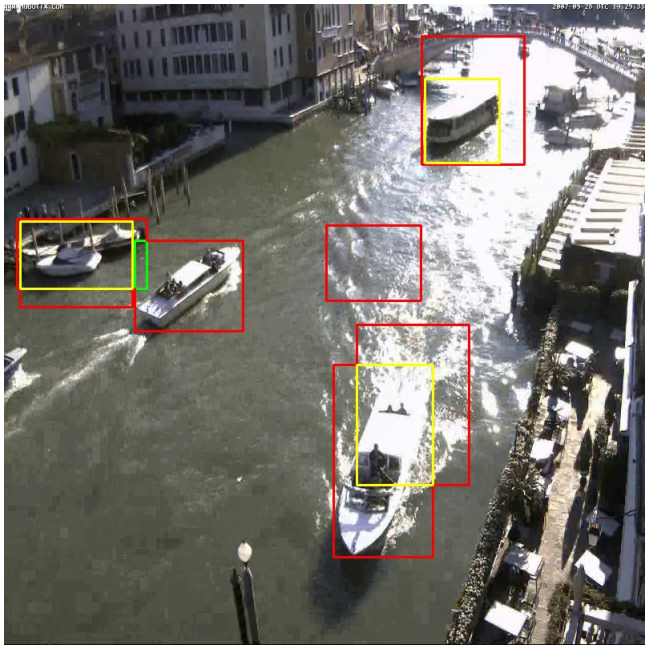

Figure 53: IoU = 0.335 638.

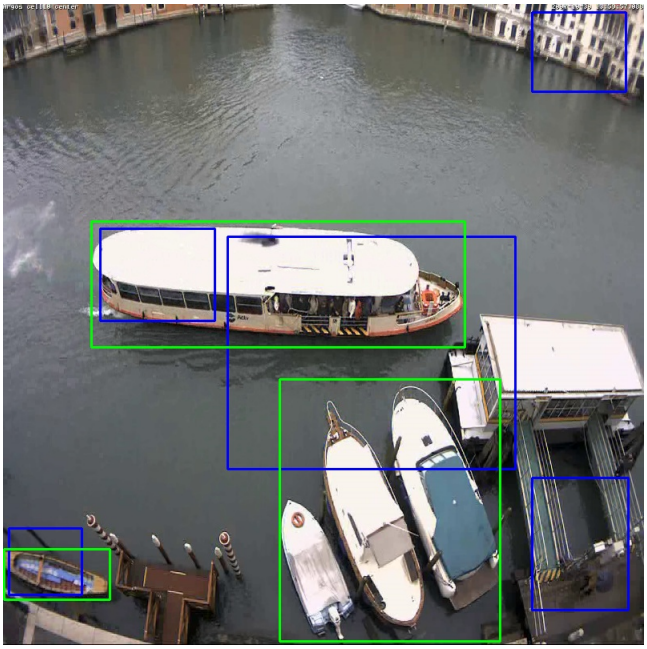

Figure 54: veniceImg10.

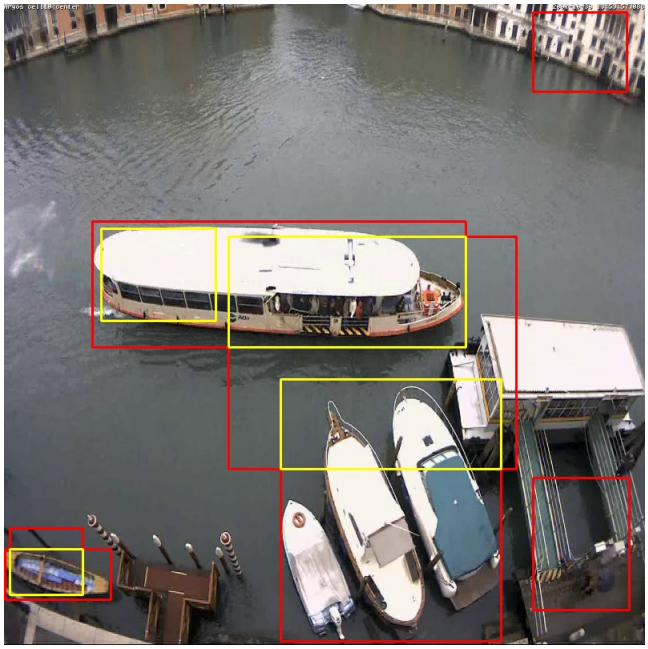

Figure 55: IoU = 0.393 192.

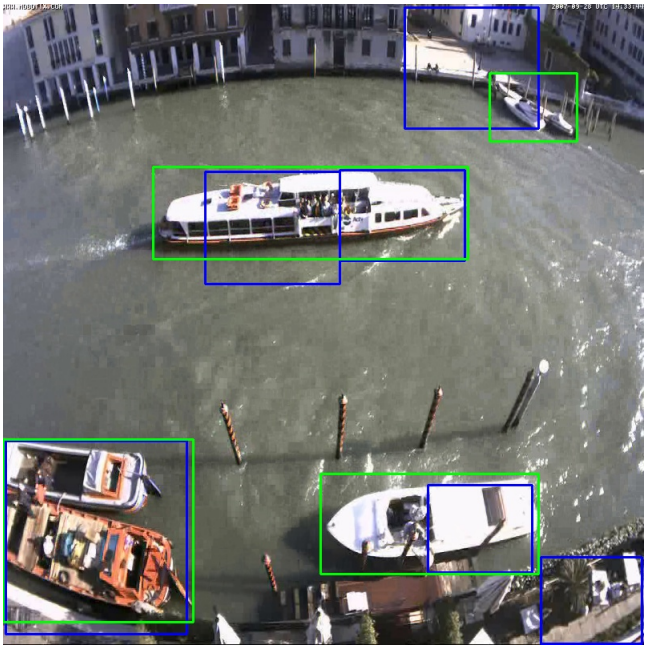

Figure 56: veniceImg11.

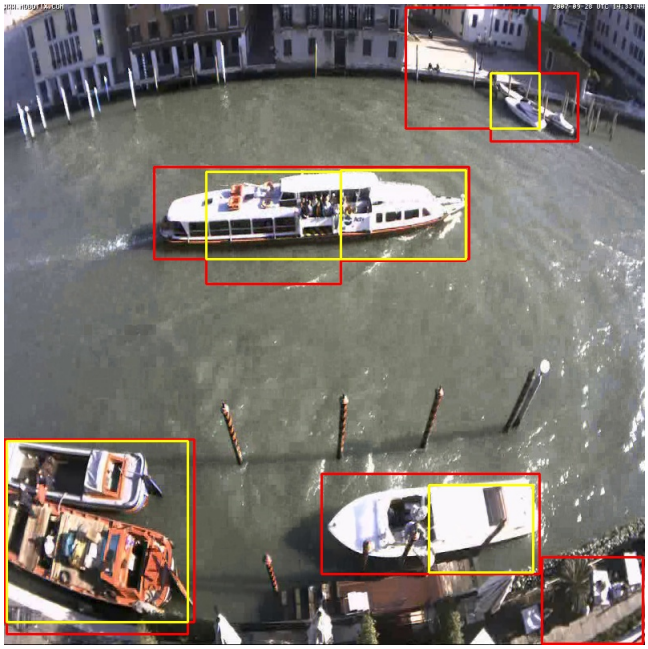

Figure 57: IoU = 0.564 906.

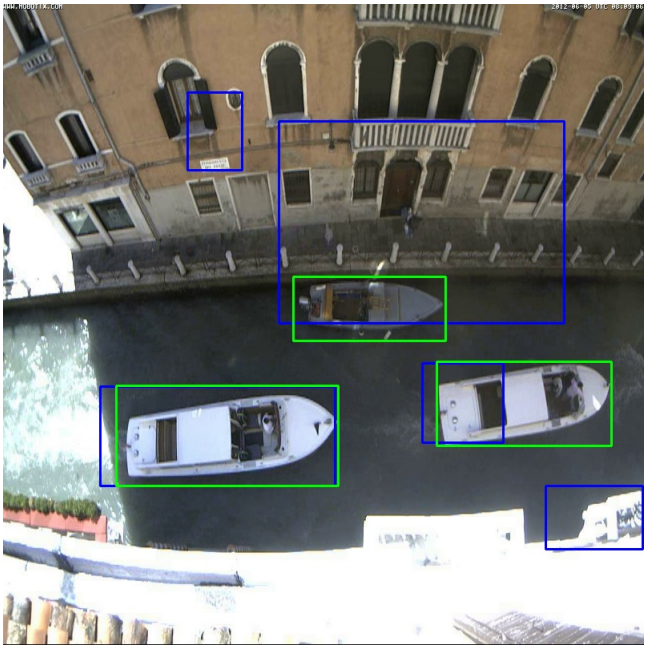

Figure 58: veniceImg12.

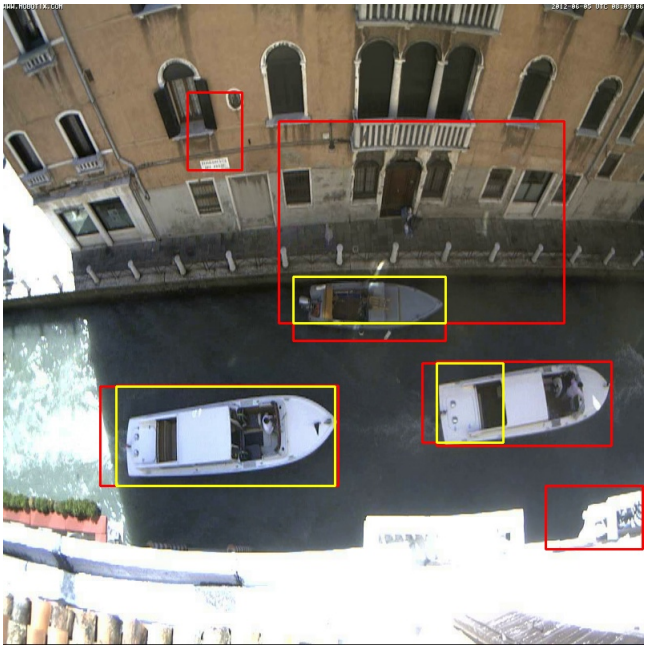

Figure 59: IoU = 0.308 254.

Conclusions

In conclusion the average IoU that we get in the two datasets is:
(1) kaggle dataset: 0.52;
(2) MAR dataset: 0.37.
Unfortunately the results are not very satisfactory but they are the best that has been possible to obtain. This may due to the fact that since there are many parameters to be fixed arbitrarily such as the codebook size, the parameters of the bilinear filter, the descriptor voting thresholds, the composition of the dataset for the codebook generation and for the SVM training, some SVM parameters, the thresholds for the mergers of the boxes, etc, is likely that the current tuning is not optimal. Moreover, is worth to notice that the adopted solution has also some intrinsic limits because, for instance, the preliminary bilinear smoothing often helps in removing non-boat features but at the same times it removes also some boat features, moreover the step (12) of the online pipeline where we shrink the final boxes of a level in order to obtain the smallest boxes containing the same features most of the times is necessary because allows to refine a box, but at the same time it has the drawback that since a boat may have regions without any feature, then these regions are cut off from the detected box. For instance in Figure 60 thanks to the shrinking we pass from the initial red box to the final orange box, but at the same time since there are no features in the tip of the sail, then the orange box cuts off that region of the boat and we end up having as final box the blue one in Figure 61.

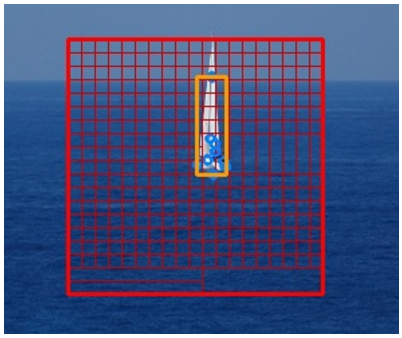

Figure 60

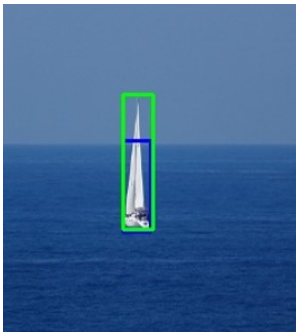

Figure 61

Other examples

Finally, we report some other examples whose original images are contained in the folder `OtherExamples`.

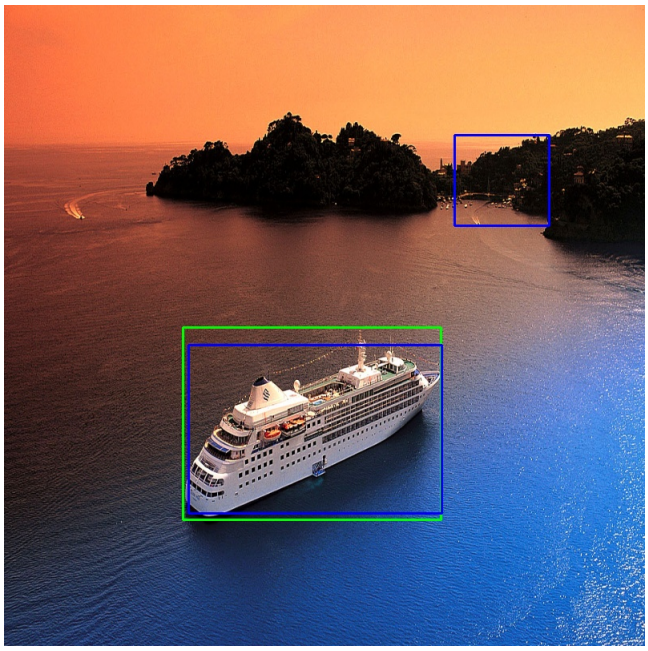

Figure 62: image 1.

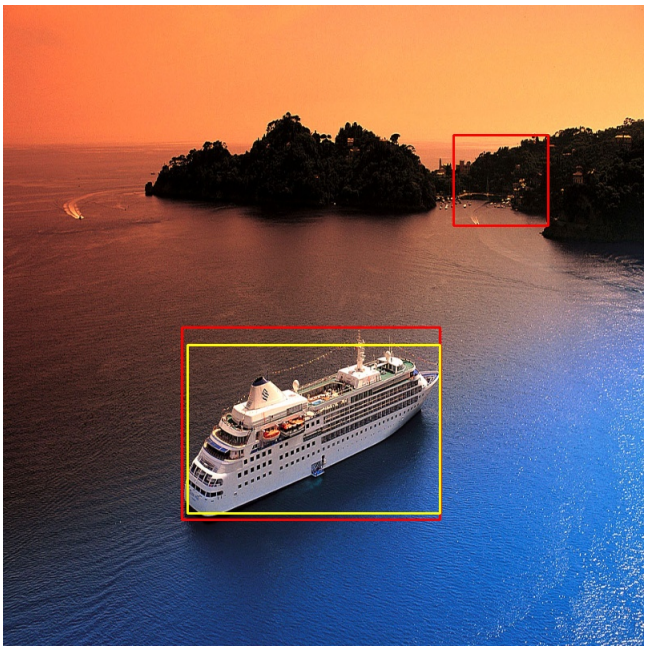

Figure 63: IoU = 0.728 694.

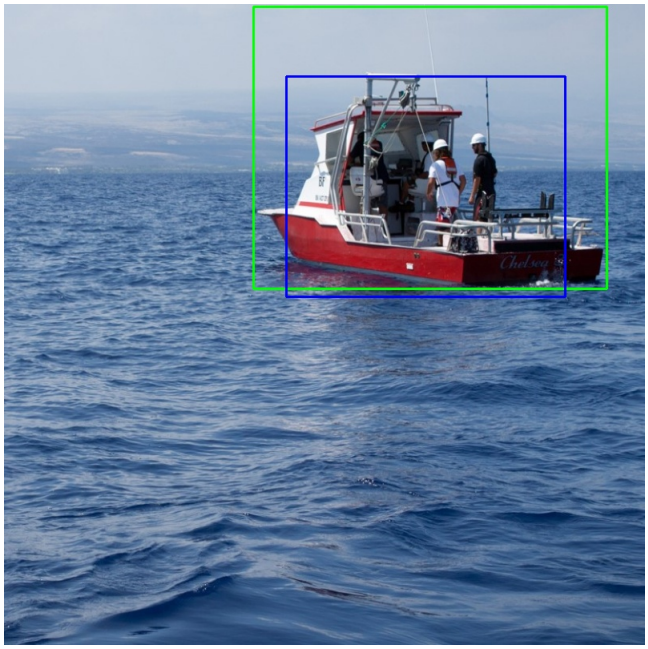

Figure 64: image 2.

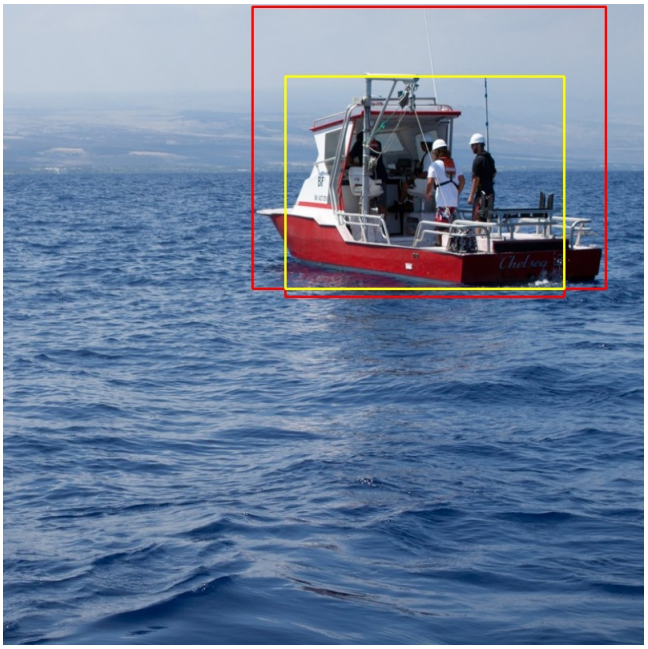

Figure 65: IoU = 0.581 487.

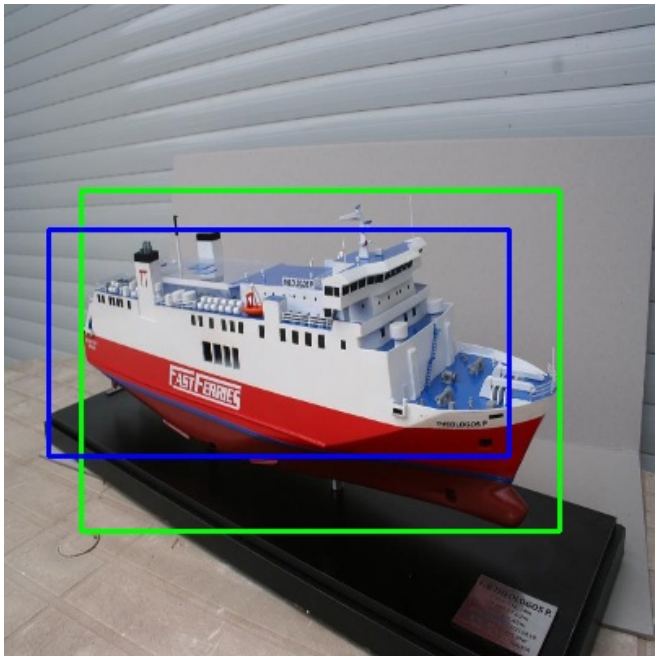

Figure 66: image 3.

Figure 67: IoU = 0.565 329.

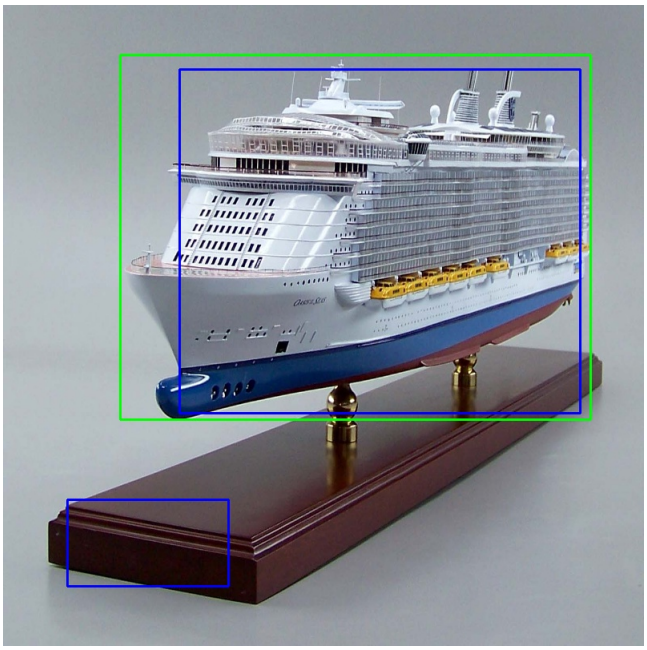

Figure 68: image 4.

Figure 69: IoU = 0.742 97.

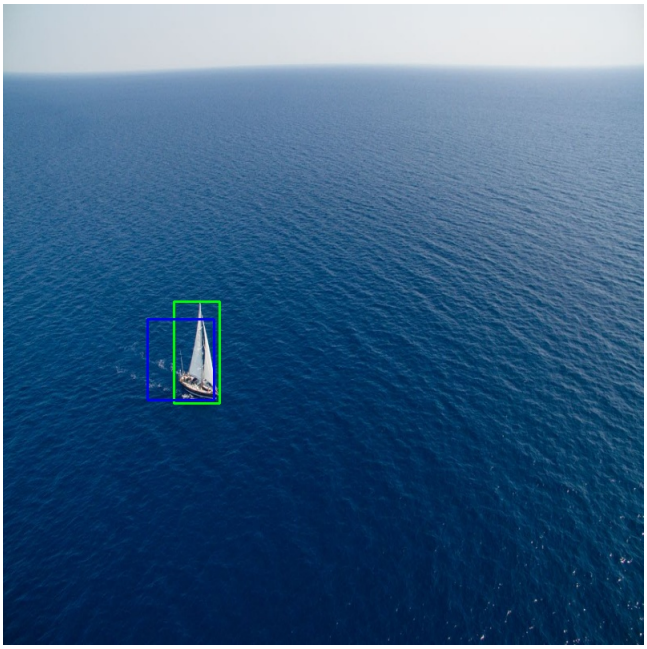

Figure 70: image 5.

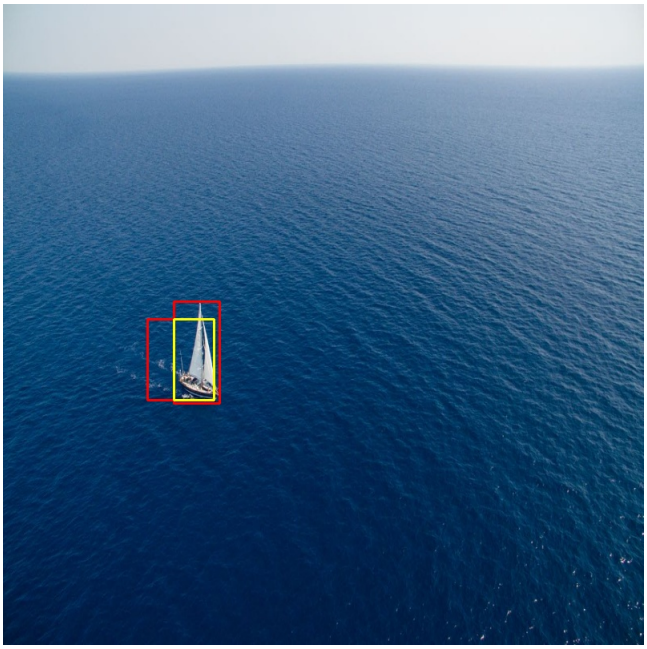

Figure 71: IoU = 0.477 627.

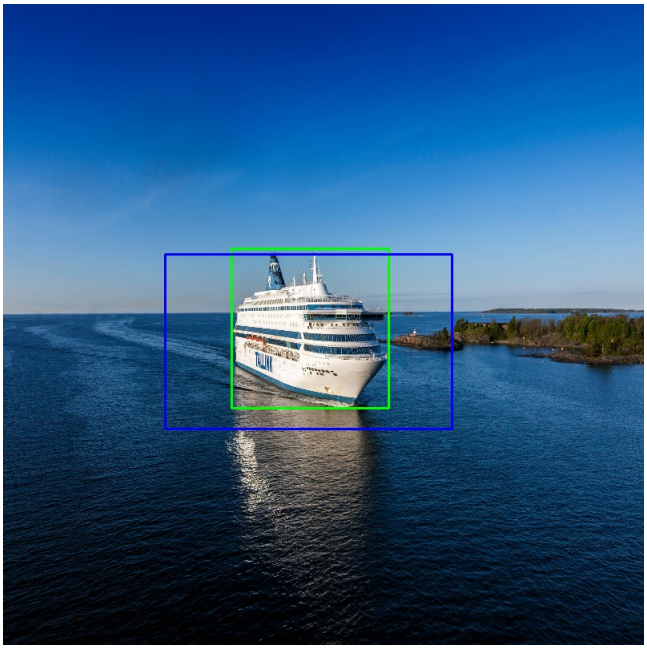

Figure 72: image 6.

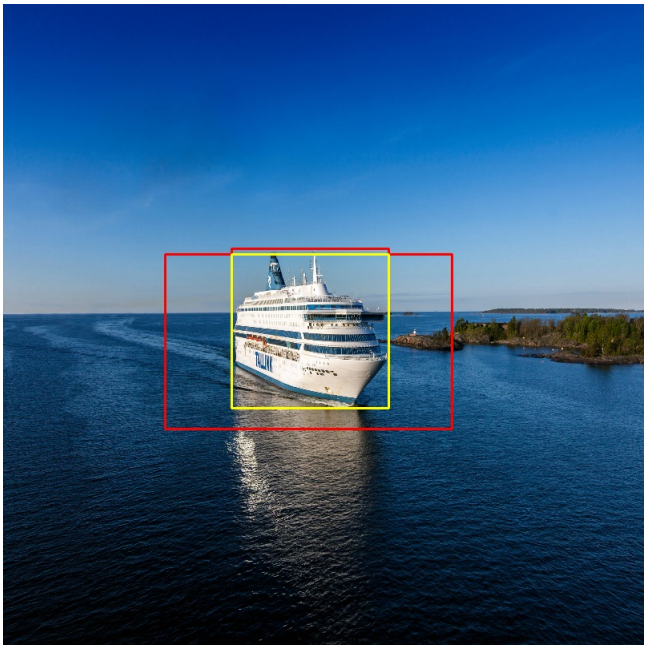

Figure 73: IoU = 0.475 06.

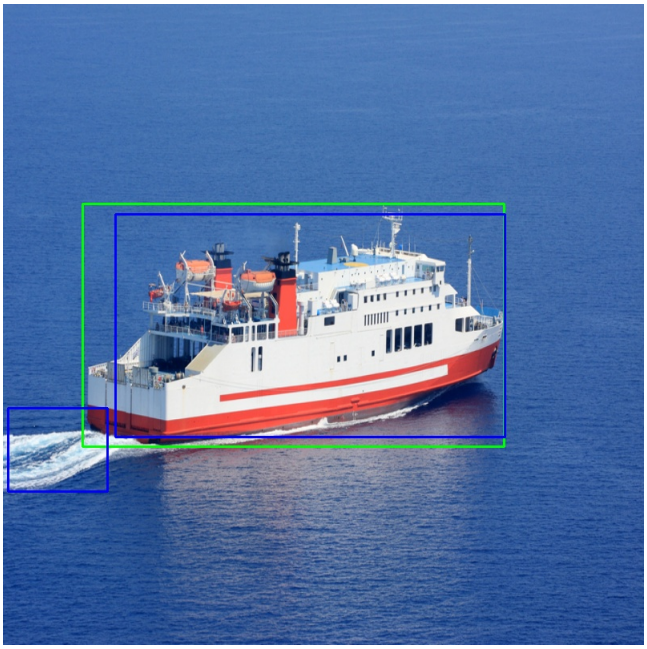

Figure 74: image 7.

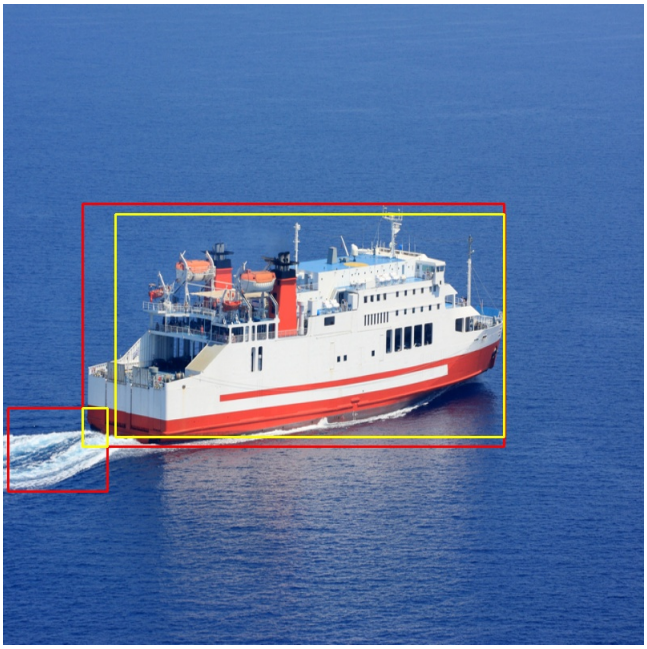

Figure 75: IoU = 0.797 169.

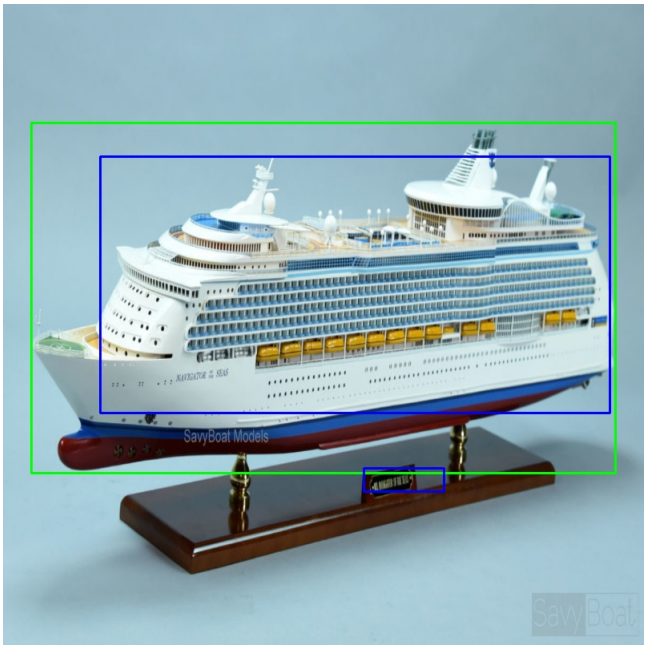

Figure 76: image 8.

Figure 77: IoU = 0.715 007.

Figure 78: image 7.

Figure 79: IoU = 0.797 169.

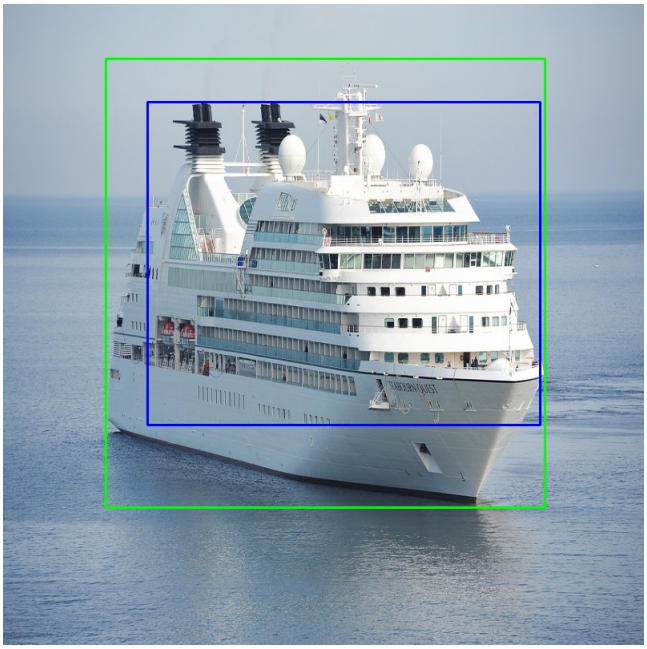

Figure 80: image 8.

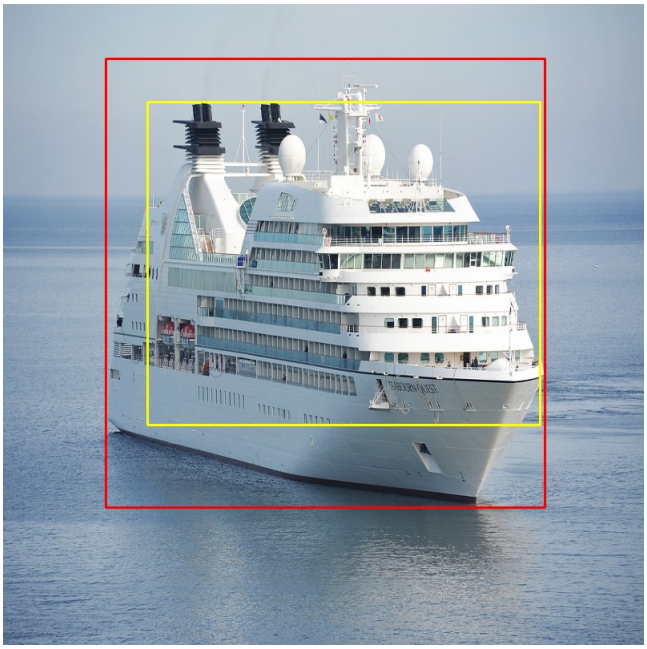

Figure 81: IoU = 0.643 006.

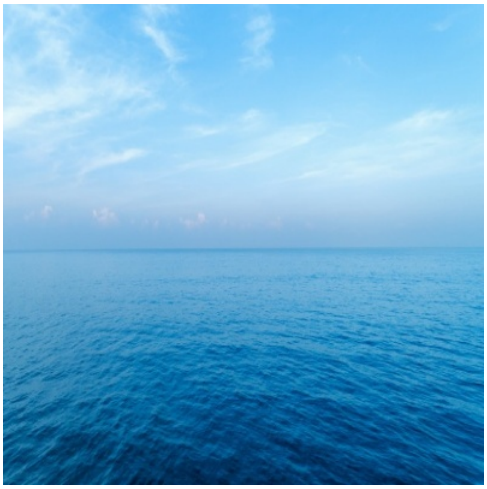

Figure 82: no false positive detected.

Figure 83: no false positive detected.

Figure 84: no false positive detected.

Figure 85: no false positive detected.

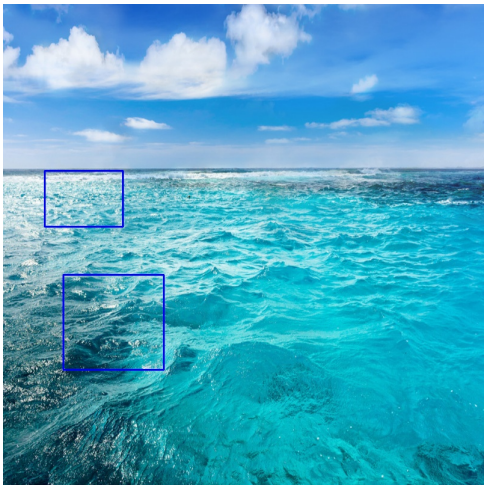

Figure 86: two false positive detected.

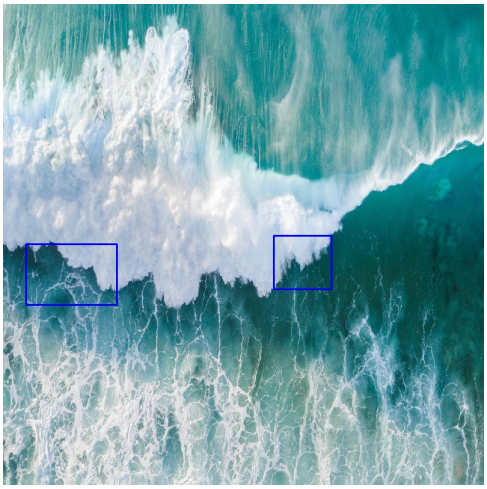

Figure 87: two false positive detected.