Boat detection

Nicola Gulmini*
University of Padua
Department of Information Engineering

Abstract

The purpose of this project is to develop a program that is capable of detecting one or more boats in an image and also to distinguish the case in which there is not any boat. To do so, the chosen method is a cascade of weak classifiers, trained on haar features. Once trained different models, the obtained results were compared.

Keywords: object detection, boat detection, haar features, cascade of weak classifier

1 Introduction

The task of object detection is one of the most important and interesting tasks of computer vision, it is sufficient to think about video surveillance, self-driving cars, or our smartphone that unlocks when we look at it (without masks...). Over the years, several approaches have been taken into consideration until arriving at deep learning based approaches that achieve very competitive performances. Many modern libraries allow to train a model quickly and easily, getting results at the risk of not fully understanding what's going on. Adopting different and more classical approaches, instead, can force to look more on what is being analyzed, giving more value to the part of data collection, processing and elaboration of the result.

This project is about boat detection, and it is inspired by the fact that traffic analysis is a mainstream project, while maritime traffic analysis can hide interesting pitfalls: since it is not a common task, it is difficult to find datasets, algorithms or models ready to solve it.

The goal is to be able to identify one or more boats present in a photo, drawing a rectangle that circumscribes each one, taking into account that the boats can appear at different angles, scales, brightness, or even not appear at all. In the latter case it is good to be able to distinguish images with boats from images that do not contain them.

Since this is an unusual request, we are faced with different types of problems: the wake left by the boats that makes it difficult to clearly define the boundaries of the boat, the sea that is often faded with the sky, boats that are not necessarily in the water, large cruise ships framed up close that look more like buildings than boats, oars, antennas, flags, sails that stick out and make it difficult to draw the rectangle with precision... even the real definition of boat! We can consider boats all water vehicles that accommodate men, so one might wonder if it is necessary to distinguish real boats and models, paper boats, platforms... This is not an easy task.

*nicola.gulmini@studenti.unipd.it

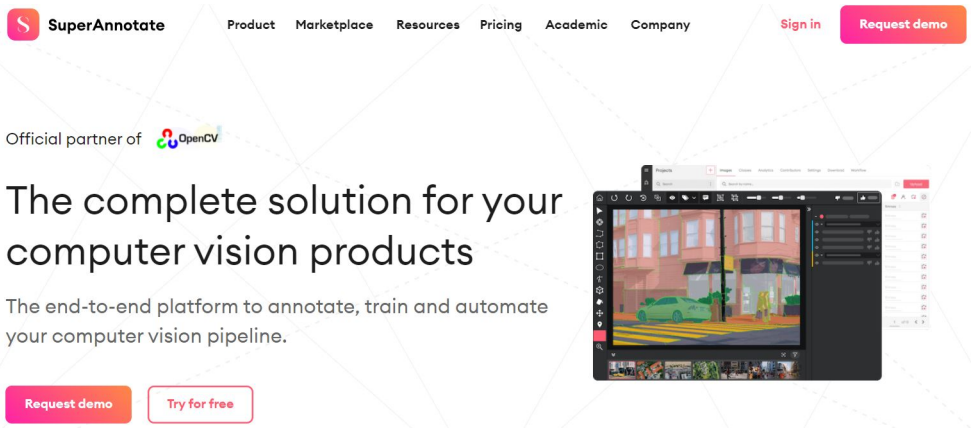

Figure 1: Superannotate.

2 Superannotate

The most time-consuming step of the entire project was the image annotation step: creating a shared ground truth so that we could train the algorithms and evaluate their performance. Being the longest and most tedious task, we tackled it first. *we* because I had help from other students. We created a working group by partitioning the datasets to be annotated. Since everyone then followed a different approach, we tried to annotate as many images as possible so that we could also train deep networks that need a lot of data.

To share the work, we relied on an online platform: Superannotate[1]. It is a platform designed especially for these kind of cases: many tasks of recent interest involving images (detection, segmentation, tracking...) require very large datasets, which often do not exist or do not meet precise needs.

We requested a demo and got in touch with the team. Explaining our project to them, they let us use a premium profile with no constraints. In this version we were able to run a predefined algorithm (a model trained on the Coco dataset) that makes a more general object detection. In a second step we split the images uniformly and corrected (modified, redone or validated) the output produced. It was a long process: there were a dozen of us and we spent several hours each to complete all the images. For this reason I would like to make the results of the annotations available at my github[2] profile.

We annotated two different datasets:

- the first one, which from now on I'll just call *kaggle*, can be found at the following link[3] and contains about 1400 high definition images. These are very various: there are cruise ships, models, paper ships, buoys, partially submerged canoes;
- the second one, that from now on I will call just *mar*, at the following link[4], contains about 4700 images very different from the previous ones. These, in fact, are taken from fixed locations in Venice, and include more standard boats than kaggle. The definition, in this case, is not so high, and the scenes often include many boats that are difficult to distinguish.

2.1 Annotation information

After completed a dataset, Superannotate allows to download a `json` file that includes all annotation information: the name of each image associated with an id, and all rectangles, expressed by vertices, or in the format of start point, width and height (Fig. 3).

[1] `superannotate.com`
[2] `github.com/nicolagulmini/Boat_Detector`
[3] `https://www.kaggle.com/clorichel/boat-types-recognition/version/1`
[4] `http://www.diag.uniroma1.it//~labrococo/MAR/classification.htm`

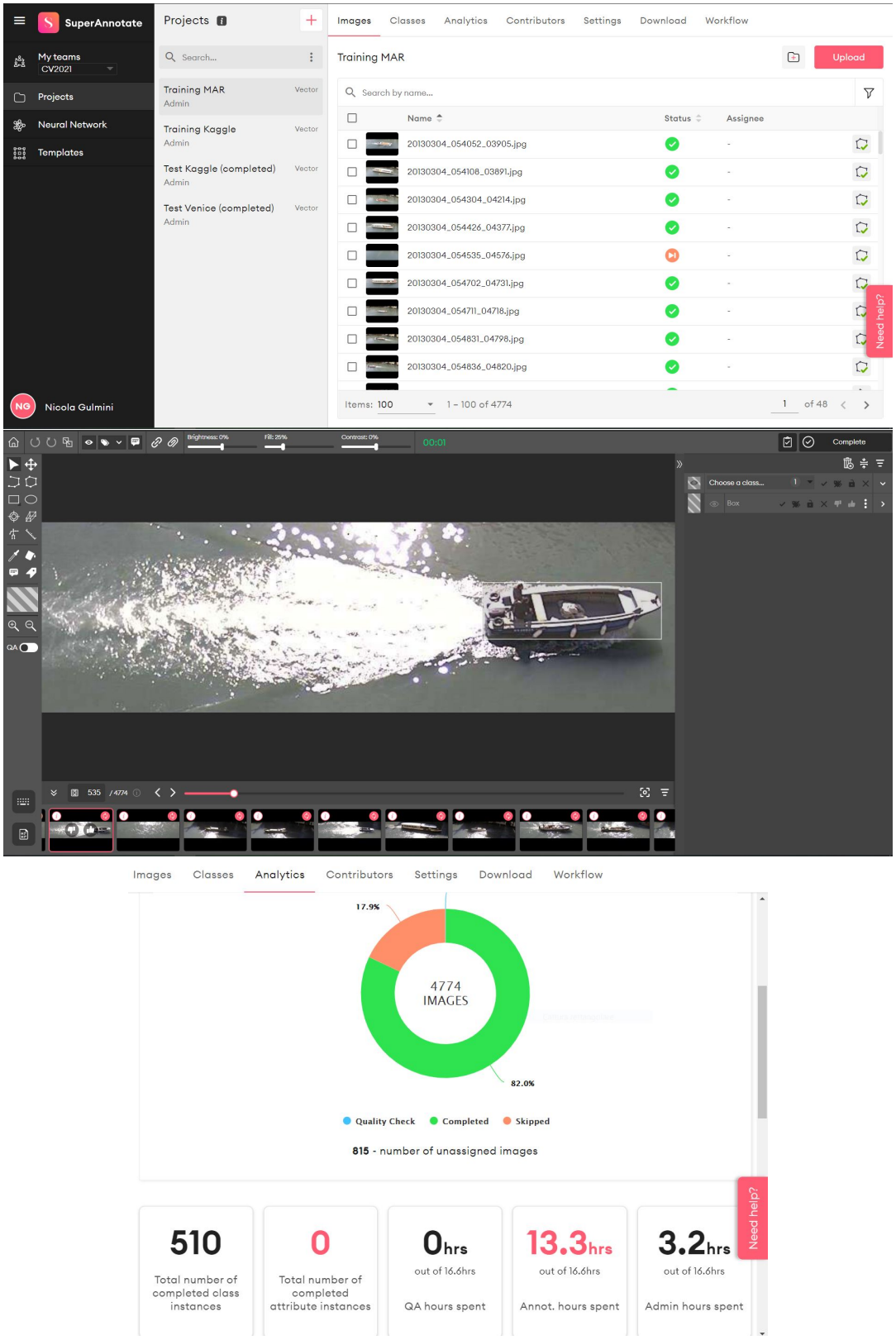

Figure 2: This is the Superannotate GUI. The green marked images are already annotated, the red ones the not annotated yet, in orange those without annotated boats. To annotate an object, it is sufficient to enclose it in a rectangle and assign it the desired class (in our case the class *boat*, but it is possible to annotate any number of objects. Finally, for each dataset there is the possibility to display data about the status of the work.

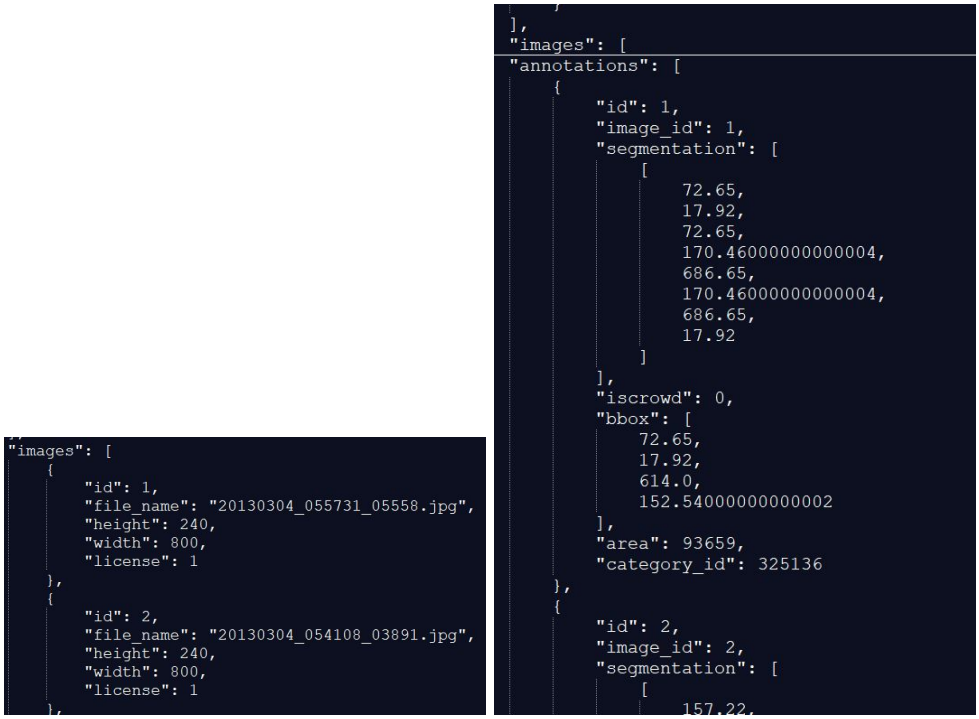

Figure 3: How the contents of the `json` file look like. Each image has an id associated with it, and so does each annotation. To find multiple annotations in an image, simply search for all annotations with the same `image_id`. There are also several formats for annotations in the file: the first contains all points from the top left counterclockwise; the second contains the first point, and the width and height of the rectangle. There is also other useful information that I ignored for this project.

3 Parsing and Image Processing

The first program I wrote is to parse the `json` files downloaded from Superannotate and extract only the useful information for the project. Since my goal is to train a cascade of weak classifiers, it will be necessary to produce *positive* images (containing annotated boats, whose position within a given image is known) and *negative* images (background or completely different images, not containing boats, to let the model learn to distinguish them). With the same code then, in testing phase, it will be possible to extract the annotation information of the test images in order to visualize and compare the results of the model.

First of all, to be able to parse the `json` file I relied on the `github.com/nlohmann/json` library because it is the one I found easiest to install and use.

3.1 Data augmentation

To allow the model to generalize and be able to identify boats even in the presence of noise or other adverse conditions, I *augmented* the dataset. To make the process faster I decided to exclude boats too small (I chose a minimum size of 64×64) and not to flip or rotate for the following reasons:

- I have assumed that in the testing phase the images are all straight and therefore no boat to be identified is upside down;
- the available datasets are already quite heterogeneous and contain a multitude of boats oriented and seen from different points of view, so it seemed not useful to insist;
- since this program only processes images, it is necessary to save in memory everything it produces. Saving even the flipped and rotated versions of the images would have been too spatially onerous.

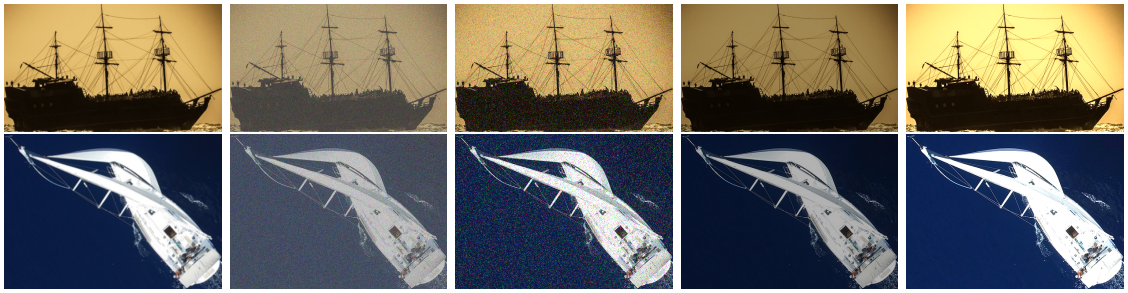

Figure 4: This figure shows two images to which the mentioned filters have been applied. They are color images for display purposes, but during all the following phases of the project the images used by the programs are in grayscale. Note, in the last version of the second image (bottom right), how the increase in brightness tends to remove details that might be important. For this reason I avoided to increase further.

Despite this, in the code there is a section to flip and rotate, and I have placed it after the filters because they are more demanding from a computational point of view: reversing the order means to get to apply the filters three times more.

For each boat the program produces five images:

1. a smoothed version of the image, obtained by applying a gaussian blur filter with square kernel of size $(3, 3)$ and sigma 3;
2. one with added gaussian noise;
3. one with salt and pepper noise;
4. one with 30% more brightness;
5. one with 30% less brightness.

I did not exceeded with the brightness otherwise the brighter/darker images in the dataset would become *too* bright/dark; and for the noise I used the method `addWeighted()`. Finally, I also reserved the same type of treatment for images that *not* contain boats, so that the models could generalize to negative cases as well. In Fig. 4 there are two examples of images produced by the program just described.

4 Training Cascade

At this point I have trained the cascade. Since my version of OpenCV is too recent (4.5), the app `opencv_traincascade`[5] is deprecated, and I couldn't access it in any way. I therefore relied on Cascade Trainer GUI[6], which is a compiled version of the app, just for those who are in the condition of not being able to use it. The GUI looks like in Fig. 5 and requires some parameters, the most important of which are the following:

- the path to a folder containing:
  - the folder `'p'` which contains the positive images, i.e. containing boats;
  - the folder `'n'` which contains the negative images, without boats;
- the percentage of positive and negative images to use for training;
- the number of cascade stages to be trained (default: 20);

[5] `github.com/opencv/opencv/tree/master/apps/traincascade`
[6] `https://amin-ahmadi.com/cascade-trainer-gui/`

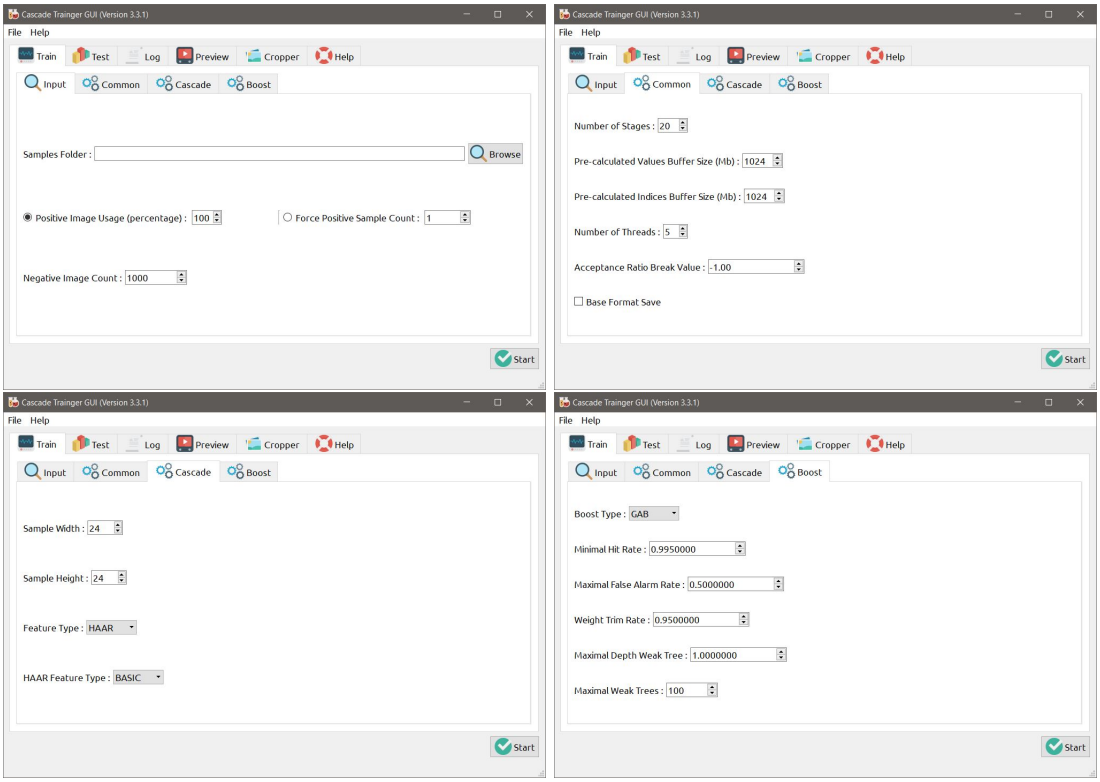

Figure 5: The figure shows the part of the program I used, but it also allows to do the detection phase in a very simple and fast way. I consider it very useful and I will use it for further experiments in the future.

- the number of threads dedicated to the program in the training phase (default: 5);
- sample height and width;
- the features type, between HAAR and lbp (with lbp the training is faster)
- the minimum desired hit rate for each stage of the classifier;
- the maximum desired false alarm rate for each stage of the classifier.

All these parameters are better explained in the OpenCV tutorial[7].

The program then produces some files, including the `log` of the training (shown for instance in Fig. 6) and a `xml` file containing the cascade. This is the most important file because it can be used by different programs and for different purposes. The cascade is the only thing produced that is needed for the next steps.

Personally, I decided to use a cascade of weak classifiers because it seemed to me the most interesting method, and I was curious to try it and study on my own how it performs in cases different than the face detection studied. Indeed, the philosophy between the two tasks is quite different: in the case of face detection there is the trend to prefer a false negative than a false positive, for security reasons for example. Also in this case I could adopt such approach, but I don't consider it necessary. Also, even if it is a machine learning based approach, it requires more knowledge of what is going on, since the various steps that this task requires (annotation, processing, training, detection) are separate but consequential, so it is not like giving a dataset to a deep neural network that does everything.

Finally, since it is the first time I work on a project like this, I preferred to train more cascades and then compare the performance and the results obtained on the test images. Obviously the results are

[7] `https://docs.opencv.org/4.5.2/dc/d88/tutorial_traincascade.html`

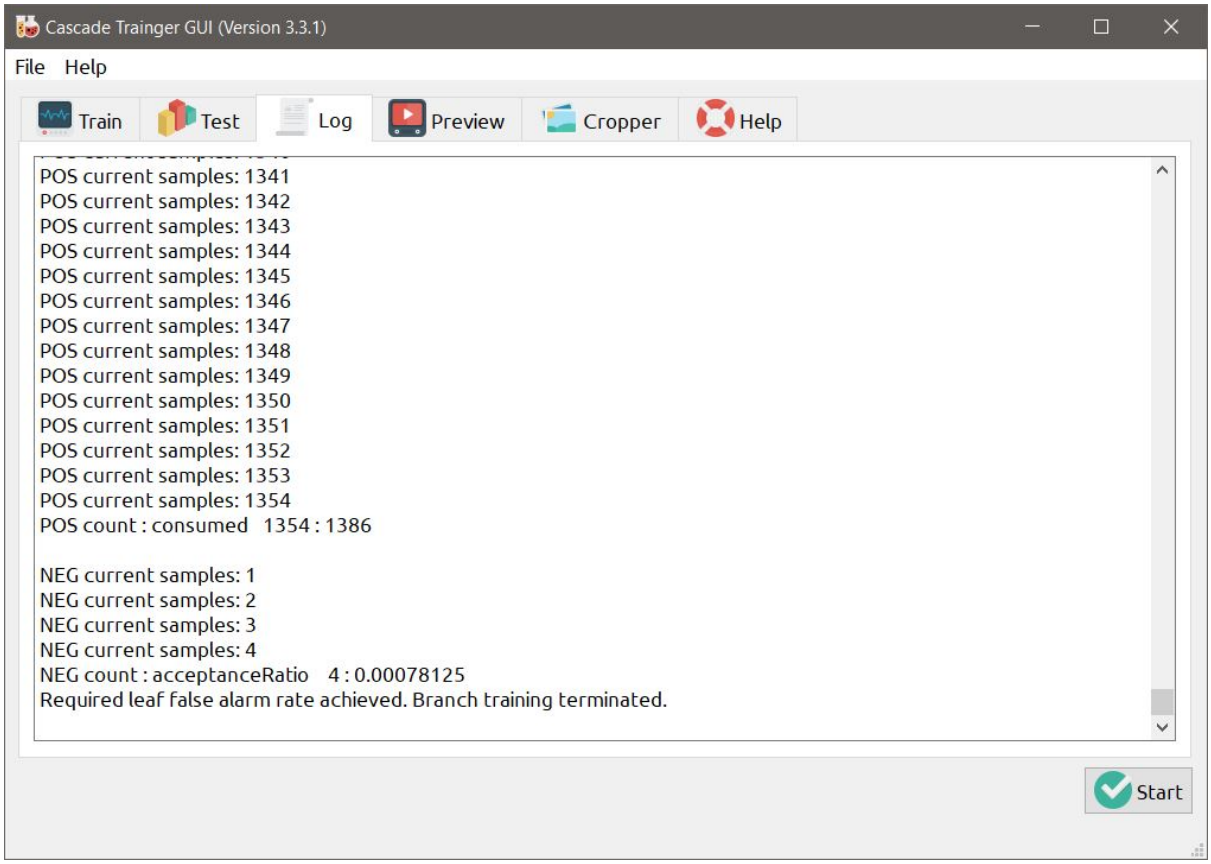

Figure 6: The `log` produced during the second training. It stopped earlier since the required leaf false alarm rate was reached.

not comparable to those achievable through deep learning, but the purpose of this project was not to get the best possible results.

4.1 First cascade: raw kaggle dataset

The first training I did was on 70% of the kaggle images, without preprocessing, with windows of size 24×24 and with default parameters, to see how much the dataset allowed the cascade to be able to generalize. The results are after all satisfactory, since after less than 1 hour of training the model is able to compete with other models trained for more time on more numerous and processed datasets.

4.2 Second training: preprocessed mar dataset

The second training I did was on about a thousand preprocessed mar images. Given the performance of the previous model I did not increase the number of images, especially because the mar images are very similar to each other. Instead I have drastically reduced the number of negative images, again because this dataset does not offer a lot of them. Finally, I increased the window size to 60×60, and reduced the number of stages to 3 to make the training faster. Also these results were satisfactory (Fig. 6).

4.3 Third training: preprocessed kaggle dataset

This was the longest training ever, taking over a day and an half on my computer. I wanted to test how well a cascade could do, and I got terrible results. I include in this report also the results produced by this cascade for completeness, but they are not comparable to the others. In this case I used 6000 processed images and a thousand negative images. To avoid spending too much time, I initially set the

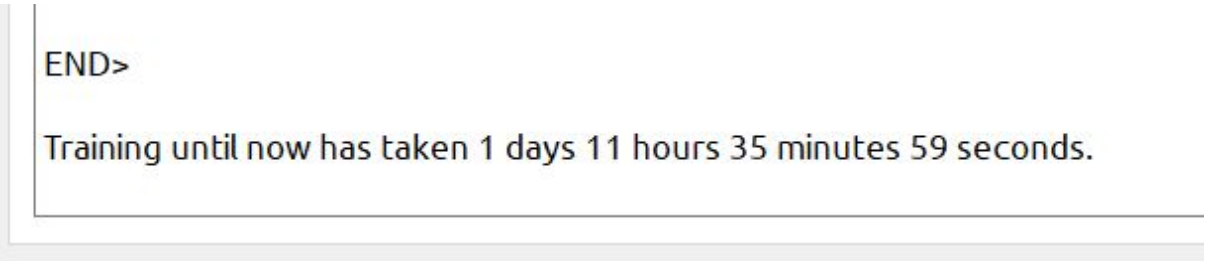

Figure 7: The longest training done.

number of stages to 5 instead of leave 20. All other parameters are the same as the previous trainings (Fig. 7).

5 Testing and Performance Evaluation

Once I got the cascade, I wrote a program that makes the detection using it. This is the main program, and it can be used also for other types of detection: it depends on the cascade. It requires 2 or 3 parameters passed as command line arguments:

- the path to the image in which to detect the boats;
- the path to the cascade;
- (optional) the path to the json containing the ground truth of the image for visualization and comparison purposes.

I've spent a lot of time making the program as easy to use as possible, with notifications that update the user on the computation steps, or that report errors. I would have liked very much also to make a GUI or a web app but I did not have enough time, given my current knowledge. I don't exclude that in the future, just for fun, I could continue to work independently on this project.

5.1 Brief description of the program

The program first loads the cascade and the input image. The method I used to make the detection is `detectMultiScale()`, which returns *square* bounding boxes around the detected objects. For this reason I transform the image under analysis into a square image, and then transformed it back when the detection was complete, to display the results.

When I resize the image with the method `resize()` I can choose the type of interpolation to use. I chose `INTER_LANCZOS4` because it is the one that allowed me to obtain the most satisfactory results. I noticed that different interpolations make the detection results change significantly, especially if the original image is very large.

Moreover, before arriving at the detection, the image undergoes a `equalizeHist()`, also to improve the results. In a first attempt I expanded this processing phase, including sharpening filters, but they did not give me the desired results, so I removed them.

The detection phase, as mentioned above, is done through the method `void cv::CascadeClassifier::detectMultiScale` whose most important parameters are:

- the input image;
- the vector `vector<Rect>` which will contain the bounding boxes;
- the vector of double `levelWeights` that will contain the certainty of classification at the final stage;

- the scale at 1.05 to improve detection on multiple scales, at the cost of increased computation time;
- set the number of neighbors to 6;
- set the minimum size to 64×64.

The program then allows to visualize the result of the detection through red rectangles drawn on the original image. At this point I have added a threshold based on the level of certainty of the detection: I have decided to draw only the rectangles of which the model is sure, where for *sure* I mean that only the rectangles which are associated with a level of certainty higher than 60% the maximum certainty will be displayed. I determined the 60% empirically, and in Fig. 10 there is a comparison by varying the threshold.

If the last argument is specified, then the program goes ahead and draws the ground truth on the image as well, using green rectangles. This is shown in the next figures of the report. In this part of the program some of the previously mentioned `json` parsing operations are taken, since I assumed that the ground truth comes from an annotation similar to the one described in the section on Superannotate. Finally, to evaluate the performances, the program computes the *Intersection Over Union* (IoU) among all the bounding boxes produced, as recommended in the project assignment. Many of the obtained results, however, are more understandable when visualized, than when evaluated using IoU, so some of these will be omitted.

5.2 Comparison

I performed performance evaluation on the following two datasets:

1. mar images[8] (appropriately removed from the training set);
2. kaggle images[9] (appropriately removed from the training set).

On both datasets I evaluated the three models I trained, and in this report I show only the most interesting results.

5.2.1 Comparison on mar images

Three columns are shown in Fig. 8. In the first one there are the detections of the first trained model, with ground truth. In the second column there are the performances of the model trained only on mar images. In the third column, finally, there are only the results produced by the model trained only on kaggle. In all three cases the threshold is 60%. The fact that is immediately noticeable is that the model trained only on kaggle always produces central rectangles. From a comparison between the images, the rectangle is not fixed: it moves a little, but it is still a very bad result. The other two models instead are able to do a good detection, although they are very inaccurate and fail to detect larger or longer boats, or those too small. When they overlap, however, the results are even worse.

Referring to the original names of the images in this dataset, in Tab. 5.2.1 are the best IoU values for the first 8 images in the dataset, produced by the second model, which is the one that obtained the best results. The results shown are the IoU values obtained from the detection and ground truth associated. For each boat, the highest IoU is reported, i.e. that of the most correct bounding box: I excluded the wrong bounding boxes that overlap with the ground truth.

[8] `https://drive.google.com/file/d/1kCgOFIP7meuUDh49BYxGyTYTg-kJQwys/view?usp=sharing`
[9] `https://drive.google.com/file/d/1PToX_LH4JsjU2rSiD4vJQSV8Ocojxy5m/view`

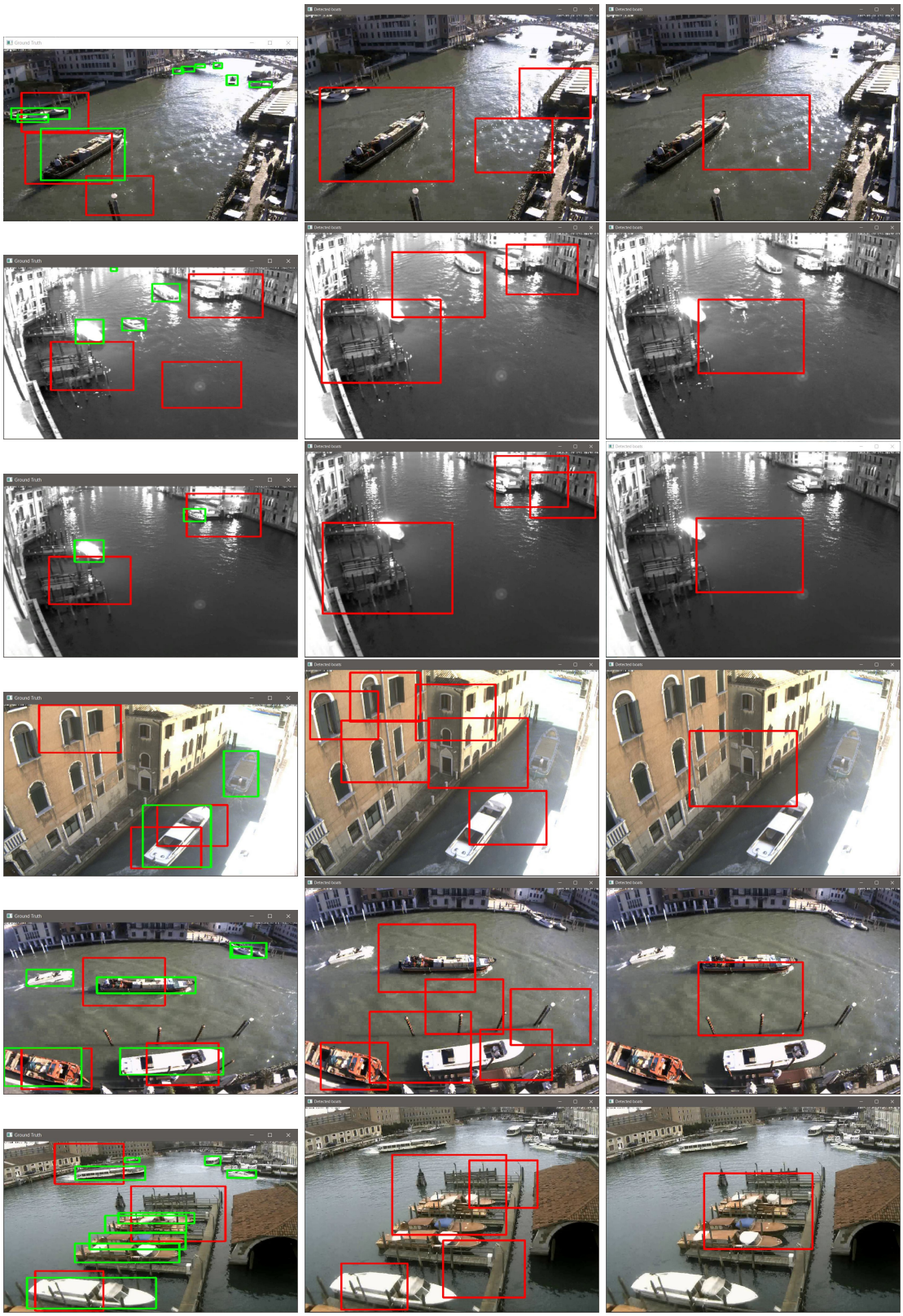

Figure 8: Aside from the last column which is not interesting, one thing that can be seen from comparing the first two cascades is that the second, when there are a lot of boats, tends to produce more bounding boxes, or is more certain of those produced, or better yet the certainty among those produced is similar.

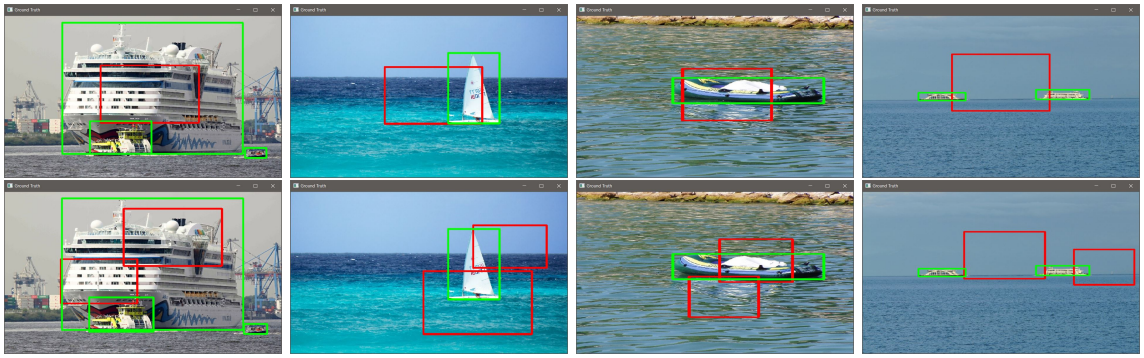

Figure 9: This figure shows only the images from which the most interesting differences emerge. First of all it can be seen that the cascade trained on kaggle does not produce much better results than those produced by the cascade trained exclusively on mar, with fewer images. Between the two, therefore, the model trained on mar is to be preferred, since with such a different dataset it can make detection also on these images. In addition, also in this case it emerges the fact that the cascade trained on kaggle tends to produce bounding boxes quite central and of fixed size, which makes this model not very reliable.

file_name	best detection	other detections		
00.png	0,41895			
01.png	0,0990903			
02.png	0,0467411	0,0429752		
03.png	0,300415			
04.png	0,619655	0,21668	0,187713	0,145486
05.png	0,412295			
06.png	0,277019			
07.png	0,245381			

5.2.2 Comparison on kaggle images

In this case, the comparison is perfectly analogous to the previous one, and the results produced are as well. To vary, therefore, only a few images are present in Fig. 9. I performed a comparison between the model trained only on mar and only on kaggle. The reason I did not train models on *both* datasets is that the results seen so far have shown that cascades trained on a small number of images are able to generalize even on very different images, which did not justify the need to make the dataset even more diverse.

5.2.3 Certainty $\neq$ Correctness

In Fig. 10 is shown a comparison between two different thresholds: one at 60%, the other at 0. As expected, some bounding boxes are excluded from the threshold, but an unexpected result is the following: not all the excluded bounding boxes are necessarily worse than those where there is more certainty. In other words, when the model is sure with respect to a detection, this does not assure that such detection is more correct with respect to others, indeed. This fact emerges in more appreciable way when in the same scene there are more boats. In many images, actually, there are not so many low-certainty detections, indeed the model tends not to produce too many.

6 Concluding Remarks

In my work I needed first Superannotate, then Cascade Trainer GUI, so I didn't do everything from scratch. However, I am satisfied that I produced everything else only with the help of Visual Studio

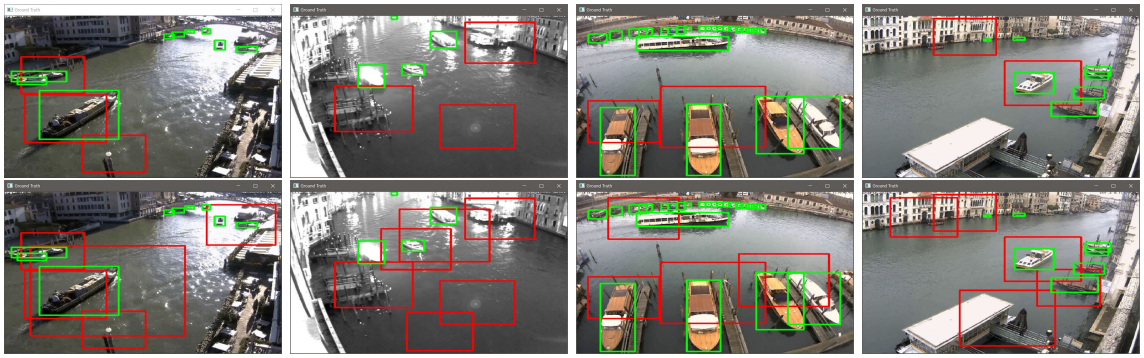

Figure 10: In this image the same detections are compared. In the upper row with the threshold set to 60%, while in the lower row without any threshold. Obviously, in this last case the upper bounding boxes are also included. Images with more boats present have been chosen to better appreciate the differences. The most interesting fact is that not necessarily the detections on which the model is more sure are the most correct.

in C++ and OpenCV (and other libraries). The performance is not what I had hoped for, however by documenting myself I found that it is reasonable performance for a non deep learning based method for such a difficult task. In this report I have not reported all the results and all the images because they are very numerous, but further interesting analysis by varying parameters or considering other methods could be done.

6.1 Further Implementations

For example, it might be interesting to change the characteristics of the training sets on which to train the models to see what consequences they have. Or progressively change the number of stages to see how increasing the number of stages leads to better results, quantifying them in terms of false positives, not only with the IoU. Other possible extensions of this project could be: improve data augmentation, as done in `http://note.sonots.com/SciSoftware/haartraining.html`; or do detection in video, as done in `https://docs.opencv.org/4.5.2/db/d28/tutorial_cascade_classifier.html` for faces and eyes. Finally, it could be anyway interesting to try models like YOLO to appreciate their performances.

To make the project funnier, I also tried to make a bot for Telegram, called `where_is_the_bot`. Unfortunately, it is not working because the library I used to make it in C++ does not provide for the exchange of images, but only text. I have not been able to install other libraries. An easier way to accomplish this would be to use Python.

In Fig. 11 there are other results on random images taken from Google.

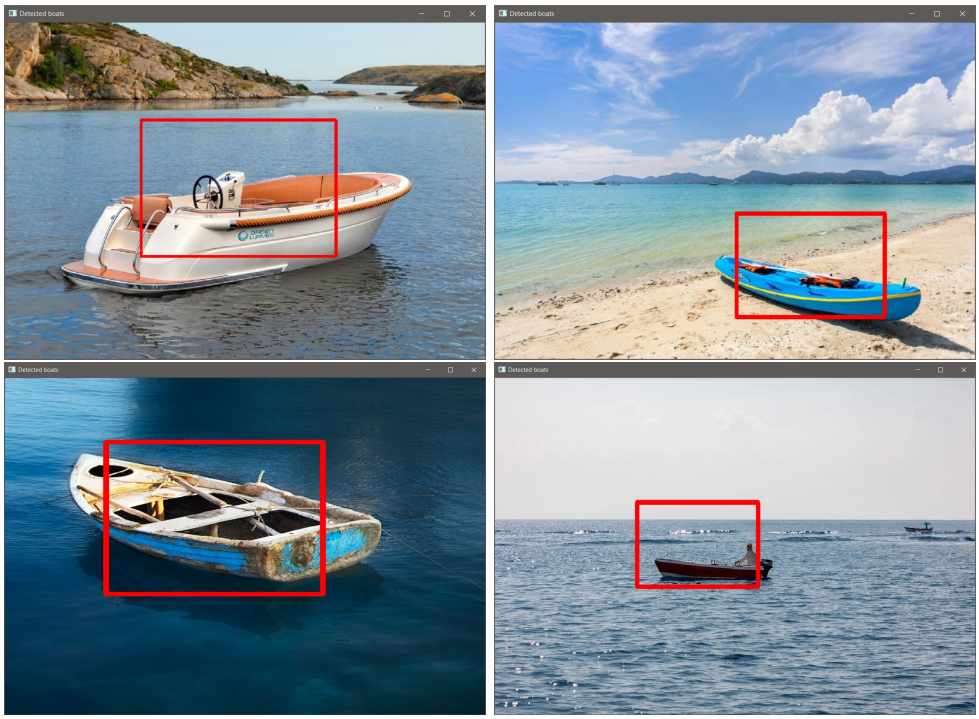

Figure 11: Other results. The bounding boxes are not perfect, but the program can clearly detect the boats in these cases. Note that for smaller boats it is not so easy.