Face Detection from Images

Diogo Silva, Nº MEC.: 89348
DETI
Universidade de Aveiro
diogo04@ua.pt

Pedro Oliveira, Nº MEC.: 89156
DETI
Universidade de Aveiro
pedrooliveira99@ua.pt

***Abstract*—Face Detection is a computer vision problem that aims to attempt to accurately identify human faces that may, or may not, be present in a given photograph or video. Many efforts have been directed at this problem since the early 2000's, with it having started as a subbranch of the problem of Object-Class Detection but having evolved into a main focus of machine learning algorithms due to its prevalence and paramountcy to the paradigm of Facial Recognition, another problem that is nowadays used in a wide-branch of technologies and biometrics.**

***Index Terms*—Supervised Learning, Object-Class Detection and Localization, Face Detection, Deep Learning, Classification, Cascade Classifiers, Histogram of Oriented Gradients, State-Vector Machine, Deep Convolutional Networks, Support-Vector Classifiers**

I. Introduction

This report serves as a compendium and thorough explanation of all the techniques, methodologies and algorithms adopted for the elaboration of the final TAA - Tópicos de Aprendizagem Automática (Topics of Automated Learning) - project at Universidade de Aveiro. In sum, the task consisted in the application and testing of several Machine Learning techniques, either developed during class or self-taught, in the solving of one of the several problems, previously proposed by the course's head teacher, Pétia Georgieva.

More concretely, the issue we decided to tackle was entitled *Face Detection from Images* and consisted in the exploration, comparison and tweaking of algorithms capable of accurately locating and localizing human faces in photographs, where locating refers to finding the accurate coordinates of a possible face within the picture, whilst localizing involves demarcating said face, usually within a bounding box. This problem started off as a sub-branch of the problem of Object-Class Detection, which engulfs all computer vision problems related to identifying the presence and/or positions of certain objects in a given image or video. Since the early 2000's, however, Face Detection grew into its own merit due to its major role in paradigms such as biometric analysis, with it being the baseline for the widely studied and sought after challenge of Facial Recognition. At first, the challenge of simply identifying that a picture has faces in it, and what their positions are, may seem like a trivial problem. And indeed, for a human, it is, but not so much for a computer due to how dynamically different human faces are from each other, as well as factors such as head rotation, pitch and yaw, features such as moles, freckles, wrinkles or facial hair, anomalies such as piercings or other accessories, hair color or obscurity (i.e, hair covering notable facial landmarks such as the eyes), age, and the list goes on.

In the following sections we will be describing our approaches to the problem at hand, starting with the a brief explanation of some of the most widely known and utilized algorithms for the Face Detection problem in section 2, proceeding by the analysis of the given data-set in section 3 as well as mention some techniques and preprocessing applied to our images. Sections 4,5 and 6 talk about and thoroughly describe the three machine learning procedures that we decided to implement and study for this problem - Cascade Classifiers, HOG Windows with SVM Classifiers and MTCNNs. At last section 7 serves as a conclusion and landing point where we briefly make an evaluation of the processes applied, make suggestions for possible future improvements and analyze and compare our three implementations.

II. Prior work done on the subject

In this section we will be presenting some of the most widely know algorithms applied to the Face Detection problem, as well as referencing their historical importance and current place in the state-of-the-art.

A. Cascade Classifiers

Cascade Classifiers, first described by Paul Viola and Michael Jones in 2001, are possibly the most famous technique of feature-based face detection. Feature-Based face detection algorithms pride themselves on being fast and effective, which has allowed them to remain relevant for a couple of decades. They center around the idea of attempting to find objects, or in our particular case, human faces, by searching the image for a given set of features or patterns that can be commonly found in the targeted objects. Cascade Classifiers in particular are but one step of the Viola-Jones Object Detection algorithm which requires 4 steps in order to be able to detect whether a face is present or not in a picture:

- Haar Feature Selection derived from Haar Wavelets
- Integral Image Conversion
- AdaBoost Training
- Cascading Classifiers

Let us focus a bit more on what each of these steps entails.

1) Haar Feature Selection: While it's true that no two human faces are exactly equal, there are undoubtedly certain commonalities such as a darker eye region comparatively to the upper cheeks, certain bounded areas where you'd expect to find a nose, eyes or mouth, brows, and other such landmarks [4]. This idea of similar characteristics between faces can be, and is, further expanded when broadening the concept to that of Object-Class Detection, since objects of the same class can be said to share, at least, some aspects. The Image 1 showcases the identification of some of these Haar facial features.

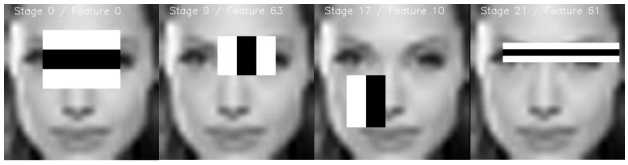

Fig. 1. A subset of images showing the identification of certain Haar Features. It exemplifies the change of light intensity between certain areas of the face [4]

The manner in which each of these *Rectangle Features*'s values are computed boils down to summing all pixels in the black area and subtracting the pixels in the white areas. As such, we have the following formula. Note that *B* refers to the number of black pixels, whilst *W* points to the number of white pixels.

$$RectangleFeatures = \sum_{b=0}^{B} b_{Value} - \sum_{w=0}^{W} w_{Value}$$

Originally, Viola's and Jones' paper referenced three types of rectangle features:

- Two-Rectangle Feature - The difference between the sum of pixels within two rectangular regions.
- Three-Rectangle Feature - The sum within two outside rectangles subtracted from the sum in a center rectangle.
- Four-Rectangle Feature - Computes the difference between diagonal pairs of rectangles.

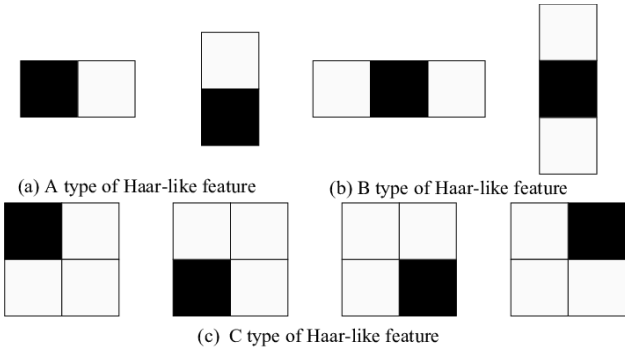

Fig. 2. An image showing the different types of Haar Features referenced in the original paper by Viola and Jones. a) Two Rectangle Features, b) Three Rectangle Features, c) Four Rectangle Features [10]

These rectangles are then applied as a convolutional kernel over the image we want to detect faces on, extensively.

2) Integral Image Conversion: Due to how taxing the prior step of identifying rectangle Haar Features using a convolutional kernel can be, Jones and Viola proposed a way to transform the images' representations, called the Integral Image, that allowed any rectangular sum to be possible using only 4 values, massively speeding up the process of scouring our image for the features. For each pixel (x,y) of the original picture, the **integral image of the pixel** is given by summing the intensity value of all pixels above and to the left of said pixel,

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y')$$

Where *ii(x,y)* corresponds to the integral image of the pixel at coordinates *(x,y)* in the original picture and *i(x,y)* points to the intensity value of the pixel at coordinates *(x',y')*.

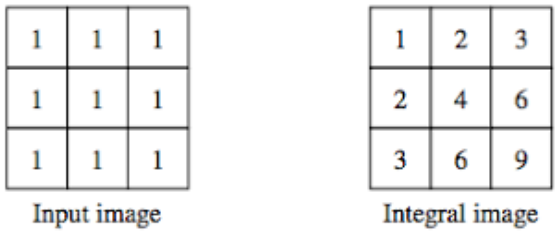

Fig. 3. Example conversion of an image in a whole integral image [11]

Whole integral images are then useful because one could easily compute the sum of pixel intensities in a given area of the original image using, at most, 4 values. For example, as is shown on the image 4, imagine we set 4 named subareas of the whole picture, A,B,C and D. The value at point 1 can be given by the sum of all pixels in area A, the value at point 2 the sum of all pixels in area A plus the sum of pixels in area B, at point 3 we have a value given by the sum of pixels at area C and A, and finally, the value at point 4 is given by the sum of pixels in area A, B, C and D which can be computed by the operation (1.+4.)-(2.+3.) (note that these correspond to the values at point 1.,2.,3. and 4. accordingly). Note how easily we computed the value inside a rectangle using only 4 array references.

When running the convolutional kernel for all of our rectangle features extensively on the image, using the integral image we avoid having to recompute set sums of pixels therefore saving us quite a bit of computation, as well as speeding up the process.

3) AdaBoost Training: Short for Adaptive Boosting, this machine-learning meta-algorithm by Yoav Freund and Robert Schapire is used in the Viola-Jones algorithm to both select a small subset of features (taking into account that for small 24x24 images, it's expected that we'll end up with over 160,000 features [4]) and train our classifier. This algorithm is chosen, both due to its versatility and capability of being used alongside other algorithms, but also due to how much

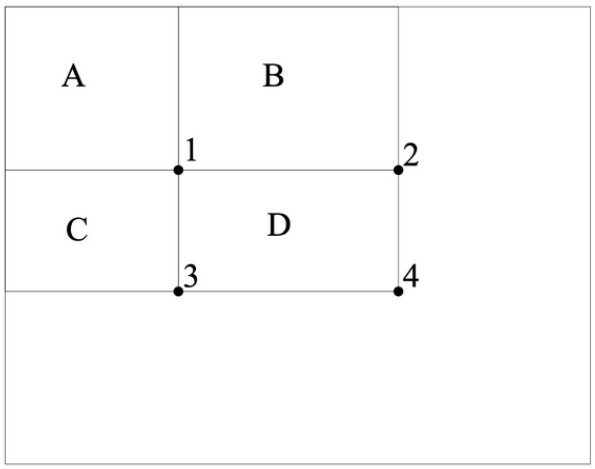

Fig. 4. Example of subdividing an image in 4 sections. The value at point 4. is given by summing all pixels in each area A,B,C,D, or more easily, by performing the computation (1.+4.) - (2.+3.) [4]

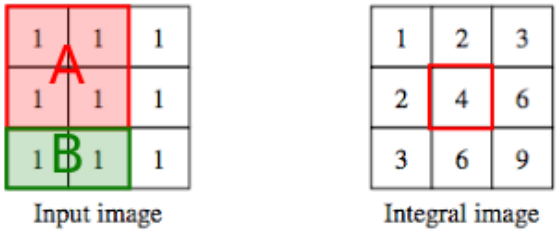

Fig. 5. Another example of usage of an integral image. In this case we can get the sum of all pixel values in region A by simply looking at correspondent integral pixel of the rightmost, down most pixel of the area (sum of pixels in area A = 4). For area B we simply do the same but also subtract the value of area A (sum of pixels in area B = 6 - sum of pixels in area A = 6-4 = 2)

it improves the overall performance of the process. AdaBoost works by being given an ensemble of weak classifiers, obtained by applying the selected features to a set of training images (these classifiers are going to label each of the training pictures as positive or false corresponding to whether they contain or not a face. Note that these training pictures should be simple and equal in size). AdaBoost then combines the quorum of weak-classifiers into a stronger, more accurate model.

4) Cascading Classifier: At last, we are finally ready to talk about the namesake of this subsection, the Cascading Classifier. The idea for this algorithm, as proposed by Viola and Jones, came from the fact that, on most pictures, even those containing faces, most of the screen-space is dedicated to non-faces (such as backgrounds or other objects) and as such, giving equal importance to all subareas of an image, when searching for features, is illogical and computationally unnecessarily taxing. What they proposed was an algorithm that would reject image sub-regions that probably do not contain a face (of course, aiming to reduce false positives to a maximum). The name **cascading classifiers** comes from

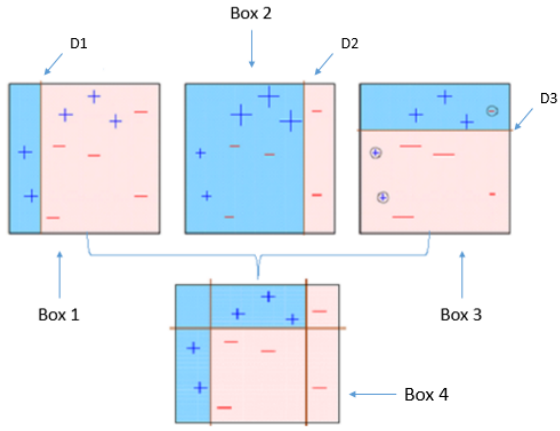

Fig. 6. AdaBoost works by combining an ensemble of weak classifiers into a more accurate one, whilst reducing the amount of features used by finding the sections that best split positive and negative images [13]

the fact that we'll be putting each of the image's sub-regions through a set of classifiers and, if at any point, one of the classifiers defines the region as not containing a face, that window is discarded, if the classifier outputs positive, and is not the last classifier on the cascade, the image is put through the next one, and so one, until it's passed all classifiers, at which point the sub-region gets flagged as potentially containing a face and gets puth through further processing, or gets rejected by one, at which point the entire region is ignored. With this technique we manage to avoid having to fully scour the entire image, since we have divided it in subsets, and shorten the time it takes for a section with no face to be discarded and devoid of further computation and search.

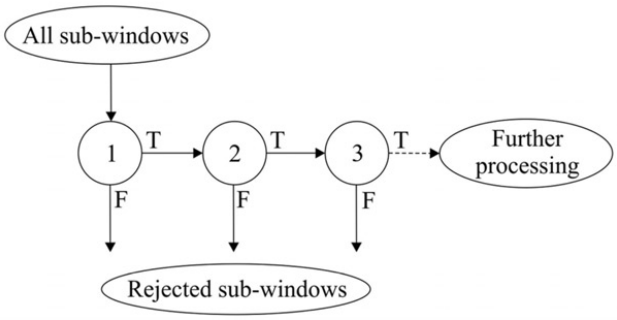

Fig. 7. An example of how multiple classifiers are set in a "conveyor belt". All image sub-sections get put through this conveyor belt, and if at each passing of a classifier the region is set to not have a face, it gets removed.

Ideally the initial classifiers should eliminate most negative examples at a low computational cost, by looking at more telling signs as to whether a face is or is not present. As we cascade further, each classifier should be more minute and filter less obvious non-faces (at a higher computational cost). The idea is that the first classifiers will operate on more regions than the latter ones, and will act as a coarse filter, whilst the

latter, more complex ones, operate only on the most promising regions, dedicating more computation and processing only where its deemed likely (by the prior classifiers) for a face to be [1].

The idea and implementation of Cascade Classifiers has been at the forefront of Face Detection problems for a couple of decades by now, which goes to show how revolutionary this technique was. Throughout the years it has received multiple tweaks and improvements, and has been integrated into a wide-variety of consumer products, such as cameras and smartphones.

B. Object Detection with YOLO

If we consider an algorithm that's already trained to identify faces, we also have to think how it can *detect* those faces. One of the ways to do it is by starting with a region in the picture we're working with, and go stride by stride looking for the object the network was trained to look for.

Needless to say, an extensive research, by feeding all possible regions with a certain size to the neural network will be too computationally expensive, so instead, we build a convolutional network with a layer dedicated to calculating all possible regions and feed all of them at once, therefore improving the overall performance of the network.

The next problem to tackle with item detection is what if one of the sections finds *part* of the item we're looking for, but, since it's only a portion, it won't be recognizable, as depicted in Fig. 8. [12]

Fig. 8. Only a portion of the car is in the window, so it can't be labeled as a car

The solution to this is an algorithm called **YOLO (You Only Look Once)**. YOLO was introduced in a 2016 paper, led by American researchers. [17] As described in their paper, the system they developed was as simple as just one convolutional network that would predict several bounding boxes and the respective class probability. This means that the CNN will only have to inspect the image *once*.

The system starts by dividing the image in an SxS grid, then check if a center of an object is in that cell. If so, the grid then becomes responsible for the object. The cell does this by predicting B bounding boxes and its confidence score. This confidence level is supposed to represent how confident the model is about there being an object within that bounding box. If we don't think an object is in the box, the confidence level should be 0; otherwise it should be the IOU (Intersection over Union) between the predicted box and any ground truth box.

In this context, they specify that each bounding box is responsible to make 5 predictions: the coordinates, (x, y), for the center of the object, the height h and width w of the object and finally the confidence score.

Each grid will predict for every C conditional class probabilities, $Pr(C_i|Object)$, conditioned on the grid cell that contains the object. With this value, it's possible to calculate the probability of the class in the box, by the following formula:

$$Pr(C_i|Object)*Pr(Object)*IOU^{truth}_{pred} = Pr(C_i)*IOU^{truth}_{pred}$$

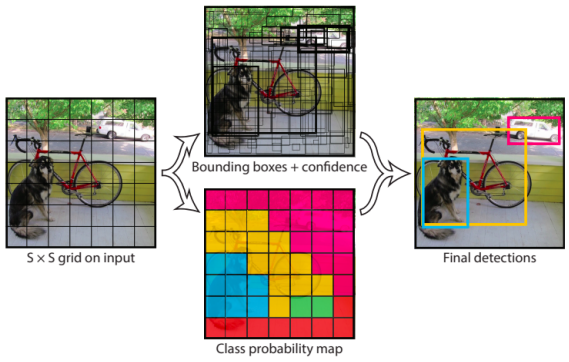

Fig. 9. The Basic YOLO Model

Finally, to quickly go through the network they implemented, it follows architecture shown in fig. 10. It's composed of 24 convolutional layers and two final fully connected layers.

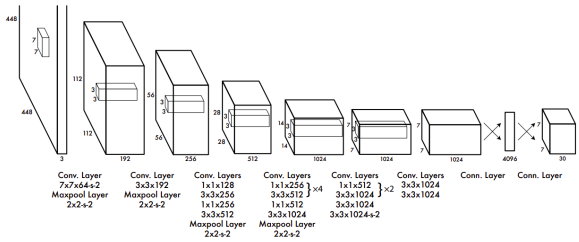

Fig. 10. The Basic YOLO Model

C. Regions with CNN

In 2014, a paper was published by a team from UC Berkeley describing a new way to detect items with computer vision. They combined the techniques from regular convolutional neural networks with region proposal, coining the technique as Regions with Convolutional Neural Network, or R-CNN. While the paper they published focuses on item detection, as opposed to being specifically about face detection, the algorithm would become the State of the Art for the time being, as it's the algorithm with most accuracy in detecting faces.

The R-CNN developed had three different components to it [2]: one to create region proposals, independent of any category; then one large CNN that would extract the features from this region as a fixed-length vector; and finally a class-specific SVM module that would classify the region.

This module was capable of improving on the last item detection algorithm, PASCAL VOC 2012, being more scalable, simpler, and with a 30% improvement rate. They achieved these results by applying two major insights: apply a high-capacity convolutional neural networks to bottom-up region proposals in order to localize and segment objects; and follow a paradigm of pre-training the CNN with a dataset that's abundant with important data, and only then fine tune the network with the scarce data that comes with face detection.

Finally, the authors of the paper finished their work by noting:

> We conclude by noting that it is significant that we achieved these results by using a combination of classical tools from computer vision and deep learning (bottom-up region proposals and convolutional neural networks). Rather than opposing lines of scientific inquiry, the two are natural and inevitable partners. [2]

D. HOG Windows

HOG, or Histogram of Oriented Gradients, Windows are a technique for the solving of problems related to Object Detection. As such, they can be adapted to solve our problem of Face Detection and Localization. The base idea behind this algorithm is to extract an image's features into the form of a vector which can then be fed into something like an SVM which will assess whether a face is present in a region of the picture or not. Features, in the case of HOG Windows, aren't Haar Features like on the Viola-Jones algorithm, but are in fact the distribution of directions of gradients of the image's subsection.

Fig. 11. The original process implemented for human body detected as originally proposed by Dalal and Triggs [4]

Computing the image's gradient boils down to having a sliding window do a passover the entire image so that, on each pixel, we look at its surrounding pixels and determine how dark it is compared to its neighbours (which can be done by using the kernels in Figure 12). The gradient is basically the arrows which show the direction in which the image is getting darker. Extensively repeating this process will make it so *"each pixel in the image is replaced by an arrow which shows the flow from light to dark across the entire image"* [6]. The reason we want to get the image's gradients is because faces tend to commonly contain the same shifts in light, even if their overall structure may vary, and as such we can use these shifts to identify landmarks such as the nose bridge and eye pockets, and in a larger scale, detect a face through the usage of light change patterns, no matter how light or dark the image may be (since we're not looking at overall light but at light shifts).

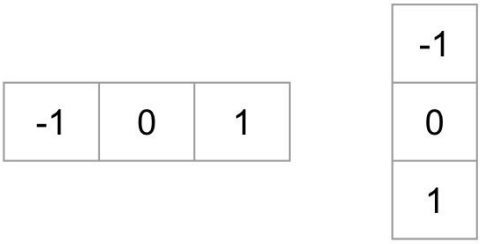

Fig. 12. The two kernels used to compute the gradients

The gradients, however, give us too much unnecessary information and detail, we don't actually need to see the shift in light for every single pixel. What we want is to get the basic flow of light in order to identify basic image patterns. We can achieve this by breaking the image into small sections, usually 16x16 pixels and, for each of these, count how many gradient points in each major direction there are (i.e, how many gradients point up, up right, right, down right, and so on). With this we can build the so-called **Histogram of Oriented Gradients**. We then replace that region with its most common gradient direction.

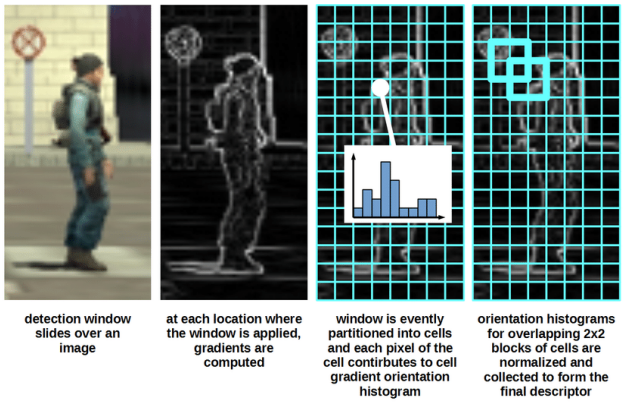

Fig. 13. An example of the process of creating the gradients and applying the HOG algorithm [14]

We then train our model, commonly with an SVM Classifier, which will then produce a pattern extracted from the HOG versions of our training faces. This pattern can then be used with the HOG version of the images we want to detect faces on in order to detect if a similar pattern can be found within the image.

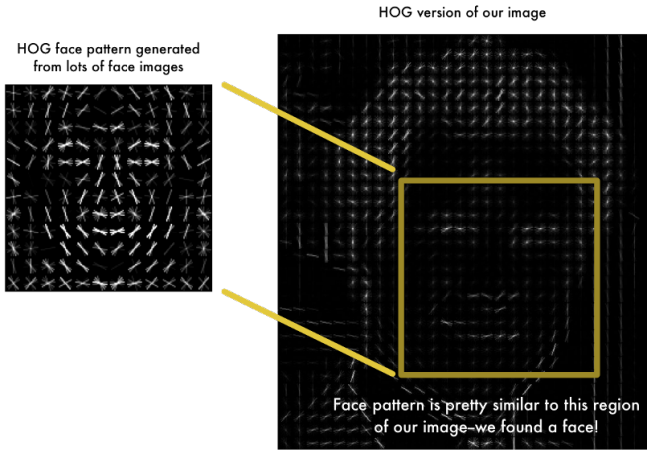

Fig. 14. An example of the process of comparing an HOG face pattern with that of an image we want to try to find a face in

E. MTCNN

Although feature-based face detection techniques such as the Cascade Classifiers or HOG Windows perform decently well with some accurate results and averagely fast processing time, the state-of-the-art for these problems lies elsewhere. Notably, astonishing results have been achieved using Deep Learning methods such as the Multi-Task Cascade Convolutional Neural Network - MTCNN [15]. This technique consists in the usage of three separate convolutional networks arranged in a cascading layout - the **P-Net**, **R-Net** and **O-Net**. From a very basic standpoint, the process used in the MTCNN consists in first rescaling the image we want to detect faces in into a wide-range of sizes. Then, we give this set, called an Image Pyramid, to the first network, the **P**roposal **Net**work, which will propose candidate regions which may contain a face. After this the second model, **R**efine **Net**work will filter the bounding boxes proposed by the first network. Finally the third and last model, the **O**utput **Net**work will propose facial landmarks, i.e, will try to find the positions for the mouth, eyes and nose. This process is exemplified in image 15.

We will go more in-depth about the individual structure of each of the utilized networks, as well as explain all different cogs in motion that allow the MTCNN to function, but for now we want to leave the idea that this algorithm has become the staple face detection solution in the past years due the amazing results it showcased when ran using the established benchmarks. [1]

F. Faceness Net: Face Detection through Facial Features

In 2017, a new paper on face detection was published, which called for the focus on the facial features of the person to better

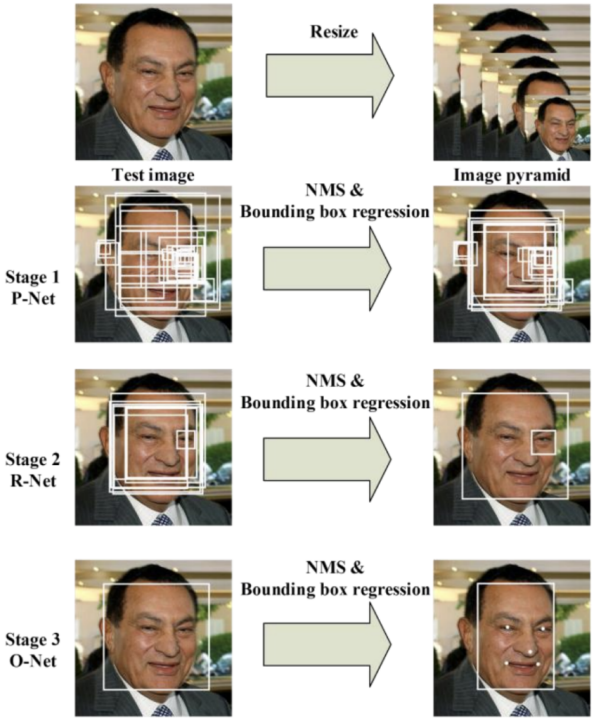

Fig. 15. The processing of finding faces in images using the MTCNN technique. *"The proposed CNNs consist of three stages. In the first stage, it produces candidate windows quickly through a shallow CNN. Then, it refines the windows to reject a large number of non-faces windows through a more complex CNN. Finally, it uses a more powerful CNN to refine the result and output facial landmarks positions."* [15]

detect faces in a picture, even where they may be clouded or not in the first or second plane of the photograph. In the time period of the paper, there were already many different papers being released on this subject, but none that gave as much focus on the actual features of a face, like the nose, mouth, beard, etc.

In Figure 17, we can see that, by focusing the neural network on the features themselves, the algorithm can detect faces even when a great part of the face is occluded. The model is also capable of detecting faces from various angles without having to specifically training the model for it, with a specific dataset.

The algorithm works has two basic stages. The first stage would be to use a set of CNN with the specific purpose of generating a partness map for each feature, which would indicate the location of the feature; with the locations of each feature, the algorithm would rebuild the image, making a face proposal, and give a score; if this rank was big enough, it would mean that the face was believable and therefore present in the picture. The second stage of the algorithm would then focus around refining the candidate window generated from the first stage using MTCNN, where face classification and bounding box regression are optimized.

All of these reworkings of the network resulted in a face detection algorithm with promising results in performance on various different face detection datasets.

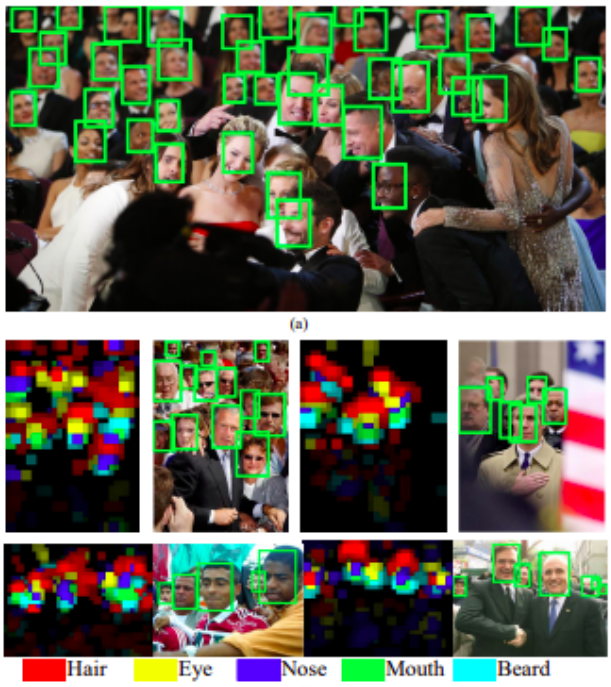

Fig. 16. Facial Features detected from the algorithm

III. Data-set Analysis

A. Data-set description

The data utilized for the elaboration of this project, made available in this link, is one of the most widely used and well known benchmark data-sets for Face Detection solution testing. The **WIDER FACE data-set**, as it's called, is a free set of images released by the Department of Information Engineering's Multimedia Laboratory at the Chinese University of Hong Kong for the purposes of benchmarking solutions to the problem of Face Detection and Localization. It contains over 32,000 images with circa 400,000 faces divided amongst three different groups - **Training**, **Validation** and **Testing** images. These groups' images are then further divided into "thematic folders" which hold images that relate to each other in terms of content. For example the folder *Dancing* contains images of people dancing, whilst the folder *Riot* contains photos of riots. It should be stated that all images differ greatly from each other both in terms of dimensions but also in terms of quality.

These images may, of course, contain one, multiple or no faces. An additional folder called *Face Annotations* is also available for download. Within it lie several text files describing the contents of each image. Of notable relevance are the files *wider_face_train_bbx_gt.txt* and *wider_face_val_bbx_gt.txt* which contain, for every single image in the Train and Validation sets correspondingly, meta information about how many faces are present in that image, alongside their position, expressed with the format *"x1, y1, w, h, blur, expression, illumination, invalid, occlusion, pose"*, where x1 and y1 are the coordinates of the top left pixel of the bounding box surrounding the face, w and h are the total width and height of said box and the other metrics describe what the face is like in terms of blurriness, whether it has an exaggerated expression or not, what the illumination is like, and so on. A better description for each of these fields can also be found on the folder's *readme.txt*. One other file is the *wider_face_test_filelist.txt* which includes the path to each image in the Test folder.

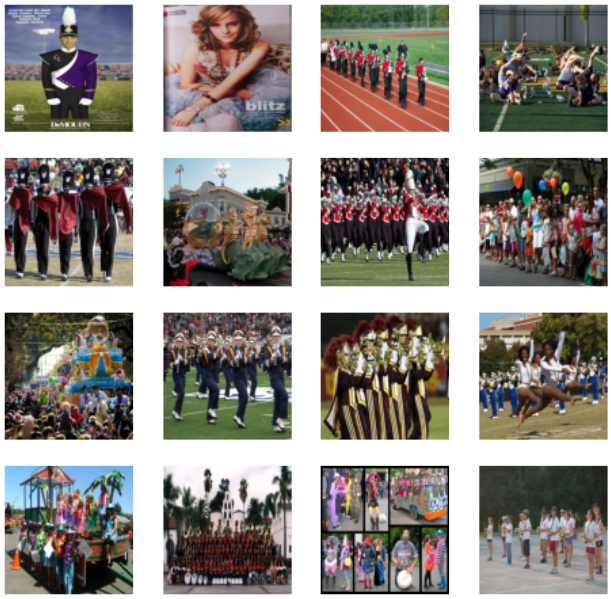

Fig. 17. A small example of the images contained in the 0–Parade folder of the WIDER Train set

```
0--Parade/0_Parade_marchingband_1_117.jpg
9
69 359 50 36 1 0 0 0 0 1
227 382 56 43 1 0 1 0 0 1
296 305 44 26 1 0 0 0 0 1
353 280 40 36 2 0 0 0 2 1
885 377 63 41 1 0 0 0 0 1
819 391 34 43 2 0 0 0 1 0
727 342 37 31 2 0 0 0 0 1
598 246 33 29 2 0 0 0 0 1
740 308 45 33 1 0 0 0 2 1
```

Fig. 18. Example of how each image is described in the wider_face_train_bbx_gt.txt file, The first number, 9, is the total amount of faces in the image, whilst the following lines describe their locations

B. Statistical analysis

The Wider Face set have their images distributed as 40% **Train**, 50% **Test** and 10% **Validation**. More specifically, the Train set has around 13000 images (1.5GB), the Test has about 16000 (1.9GB) and the Validation set sums up to about 3300 pictures (365MB). Each of these images has a varying number of faces, or none at all, and are, as aforementioned, split into folders which group them by theme.

The following figures 19 and 20 showcase how many pictures we have per number of faces on both the Train and

Validation sets. Unfortunately, due to the data's nature these images are rather inconclusive. This is due to the fact that we have an exorbitant amount of pictures with few faces (most pictures having only one face), whilst remaining true that we have photos with way more faces (with some having over 1500), albeit at a very reduced number (most pictures have 1 face but since our range of number of faces is so large we can't even see that one single line).

As these were inconclusive, we created two additional tables. Table 1 shows the number of pictures with 0, 1 and multiple faces for both of our sets. Some additional insight we got was that, by looking at the two images (19 and 20), we can also confirm that both the Train and Validation folders have a very similar data silhouette, by which we mean they have a proportional distribution of faces per image to each other.

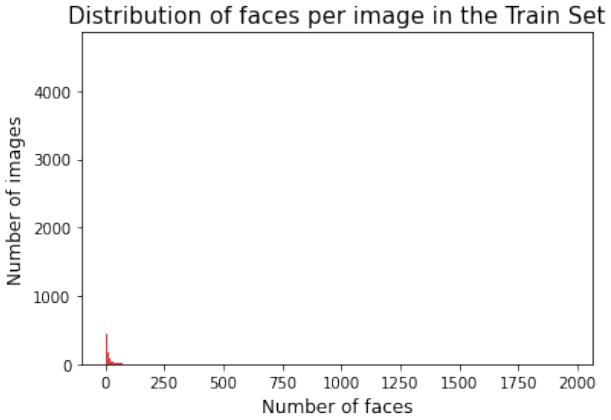

Fig. 19. Number of images per face number in the Train Set

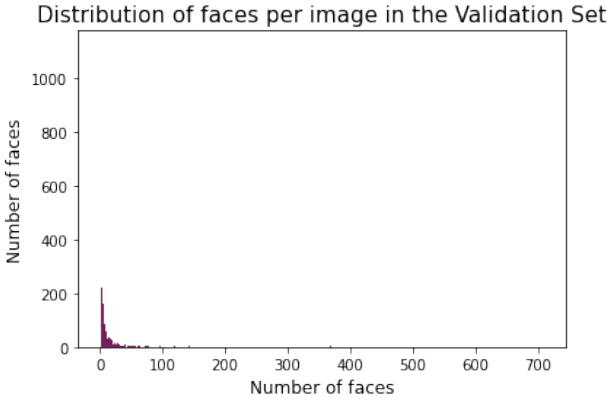

Fig. 20. Number of images per face number in the Validation Set

Two problems arise from the conclusions taken from looking at figures 19 and 20. The first being that we have a very reduced number of negative pictures, i.e, pictures with no faces (as can be seen in table 2) and the second being that its not easy or even surmountable to use pictures with multiple faces to train our models. Ideally we want to use either pictures with no faces, or pictures with only one face for training and model creation purposes. The way we treat these problems will be further explained on the following sections.

TABLE I
NUMBER OF IMAGES PER NUMBER OF FACES

Test	***0 Faces***	***1 Face***	***>1 Faces***
Train	4	4631	8245
Validation	0	1122	2104

TABLE II
NUMBER OF POSITIVE AND NEGATIVE IMAGES

Test	***Positive***	***Negative***
Train	12876	4
Validation	3226	0

C. General Preparation and Data Pre-Processing

Although that, when explaining each of the algorithms implemented we will also be explaining specific treatments to the images for them to go in accordance to what the algorithms require, there were some general tweaks that we had to do to the images in the dataset that are transient to all of the algorithms and which we will be explaining in this subsection.

Firstly, we had the problem that we have no feasible way to discover the classification of the images present in the Test folder. Research indicates that these images were purposely left unclassified as these are the ones that the team at WIDER use to test face detection algorithms, and as such, they'd rather be sent algorithms for testing, rather than having the developers use the Test set themselves. They do include a set of evaluation scripts written in *MatLab*, however these only used the Validation set, rather than the Test one. With all of this in mind we decided it'd be best to create the Test, Train and CV sets on our own, as well as our own evaluation script.

Our first efforts were aimed at dividing our images in the Train and Validation folders into a Train and Test set of images of appropriate sizes. We had to take our images in the Train set and break them down into positive and negative pictures, i.e, images of single faces and images with no faces at all. To create our positive images, we utilized the information in the *wider_face_train_bbx_gt.txt* file, which allowed us to create images that consisted only of the bounding box of each face in each picture in the Train set. We did this until we had exactly 12000 Train positive pictures. Figure 21 shows a sample of some of the positive images we generate.

As for the negative images we created an algorithm that would go through each image in the train set and try to generate a new image of random dimensions that would not intersect any of the bounding boxes around the faces of that particular image. Some of the resulting negative images for the train set can be seen in image 22.

Unfortunately, due to the size of the set of negative images, we've had a lot of problems trying to get random images that were large, as it was advised from the research we

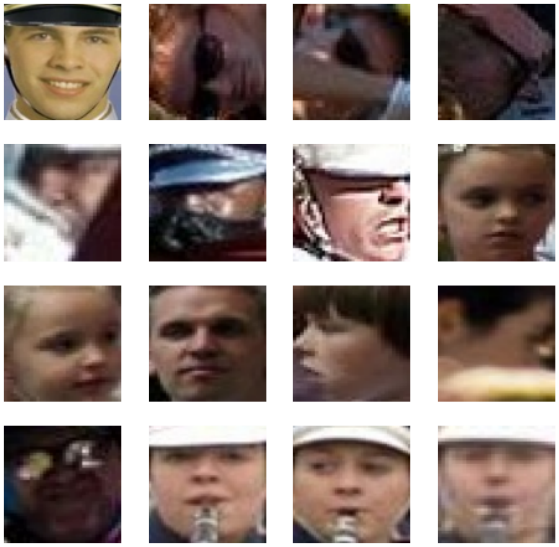

Fig. 21. Examples of some of our positive images present on our Train set

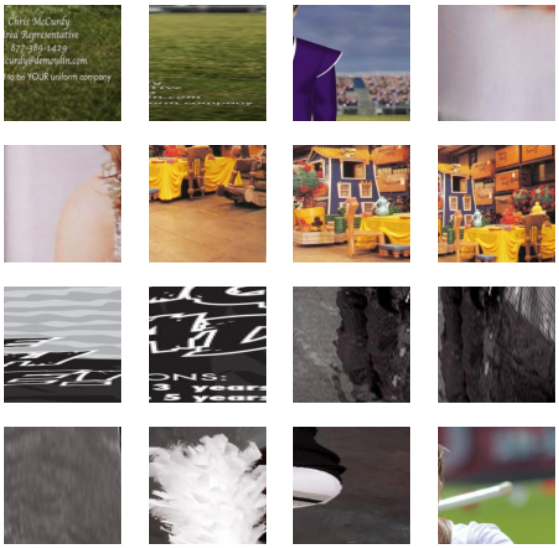

Fig. 22. Examples of some of our negative images present on our Train set

had done, while being completely negative images, meaning having no faces in them. So we decided to use a different dataset. We used a dataset of landscapes from Kaggle [28], landscape photographs usually have really high resolution, and not having any people on them. For the problem we were having, this seemed like it would be an easy fix, resulting in the following examples of negative examples, in image 23.

It should be stated that both positive and negative images were created for our sets. We should also inform that these images vary in dimensions and are encoded in RGB. Two text files were created for each set, the *positive_info.dat* which contains the paths to each of the positive images in the set and the *negative_info.txt* which has the relative path to each of the negative images in the set. Table III shows how many

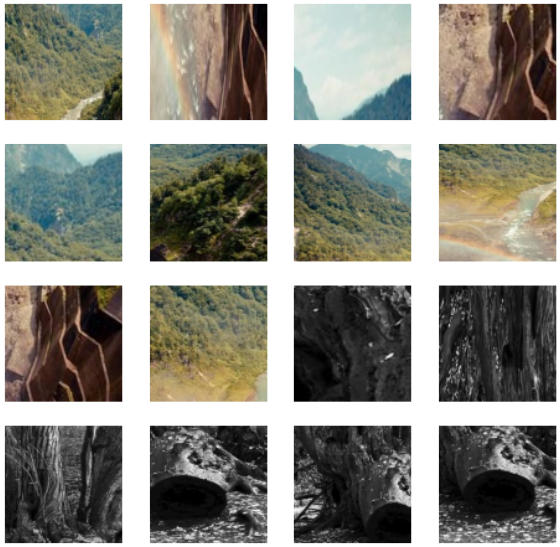

Fig. 23. Examples of the new negative images

positive and negative images our two sets ended up with.

TABLE III
NUMBER OF POSITIVE AND NEGATIVE IMAGES ON OUR TRAIN AND TEST SET. NOTE HOW THE TRAIN SET HAS MORE IMAGES THAN THE TEST SO THAT WE CAN FOLD IT AND CREATE A CV SET AS NEEDED BY OUR MODELS.

Test	***Positive***	***Negative***
Train	12000	12000
Test	10000	10000

Finally, we have to talk about our methods of testing the models we're going to be comparing in these project. As said before, we could not depend on the WIDER Face's evaluation scripts due to some technical problems we had on our part. So we implemented two simpler ways of testing our models. We implemented a test that would be similar to WIDER's way: we used a subset of the images in CV's dataset, as we can trust the images were correctly identified. We named this, our "**complex test**". While this test will tell us how ready the model is to identify faces in a realistic image, it fails to give us precise measures like accuracy or precision in terms of, whether our algorithms can classify an image with a face as positive, and an image with no faces as negative. So we implemented a second way to test the models, based around using another positive and negative set. If the model correctly identifies which is which, this will allow us to see number of true positives (faces the model correctly identified), true negatives (images the model correctly identified as negative), false positives (faces the model thought he identified), and false negatives (faces the model could not identify). We called this our "**binary test**".

Using these metrics, we're capable of calculating three important metrics in a classification problem: global accuracy, which means the ratio of correct answers and total given answers; the precision, which tells us the amount of positives

the model guessed that were actually correct; and the recall, the amount of global positives the model was able to identify. The following formulas are the ones used to calculate each of these values:

$$acc = \frac{TP + TN}{TP + TN + FP + FN}$$

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

It's important to have these three metrics when describing the performance of a model, as simply going by accuracy tells us to little about how it's doing: a 50% accuracy may mean that the model only guessed half of the positive results and half of the negative, or it may mean it guessed all the negatives and no positives. A model with high precision will tell us that we can be fairly confident in the positives it has identified, but it may be overall not guessing a lot of the positives in the set, resulting in a low recall; while a model with high recall will tell us that it's been guessing all the positive images, but it may be identifying all images as positive, resulting in a low precision.

IV. Cascade Classifiers

A. Introduction

Now moving on to actual implementations and testing of these algorithms let us start off with the predecessor and, at the time groundbreaking, algorithm for face detection using Haar feature-based cascade classifiers -the Viola-Jones algorithm. We have already explained in section 2 the base steps for this technique, but to reiterate, we have:

- Haar Feature Selection derived from Haar Wavelets
- Integral Image Conversion
- AdaBoost Training
- Cascading Classifiers

To further elaborate, we can **extract the Haar Features** (of which we will be using the three types shown in image 25) using a sliding window. Each feature is nothing more, nothing less than the value we obtain by subtracting the sum of pixels on the white rectangle from the sum of pixels under the black rectangle. These features are basically our convolutional kernel, and as such, all possible sizes and locations of each kernel (each Haar Feature type) on the image will have to be computed, which is not viable since, even for a small 24x24 images, this would result in over 160000 features to be calculated. [18].

To ease the computation of summing the pixels under the white and black rectangles, Paul Viola and Michael Jones proposed the conversion to **Integral Images** which is able to reduce the calculations for the Haar Feature of any given pixel to an operation involving just 4 pixels (no matter how large the image may be). This is explained in more detail on Section 2.

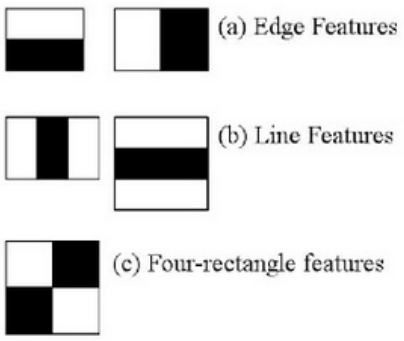

Fig. 24. The three types of Haar Features [18]

The third concept to keep in mind is that most features calculated are actually irrelevant. Figure 25 shows the application of two Haar Features to an image. These are both appropriate, the first one focusing on the fact that the region of the eyes is often darker than the region of the nose and cheeks, whilst the second being useful due to the fact that normally, faces tend to have a darker eye region comparatively to the nose bridge. Applying these same kernels to the cheek or chin area, however, would not be appropriate as there is no meaningful information or patterns to extract. This is why we use **Adaboost** in order to pick amongst the best, most promising features, whilst disregarding the others. The basic way it works is, for our training images, each and every feature will be applied. For each of these, Adaboost will find the best threshold which will classify the faces as positive or negative. Then we can pick only the features that produce the minimum error rate, i.e, the ones that more accurately allow us to classify the image has having a face or not. *"The process is not as simple as this. Each image is given an equal weight in the beginning. After each classification, weights of misclassified images are increased. Then the same process is done. New error rates are calculated. Also new weights. The process is continued until the required accuracy or error rate is achieved or the required number of features are found"* [18]. In the end, we'll end up with a final classifier which has been created by the weighted sum of all of our weak classifiers.

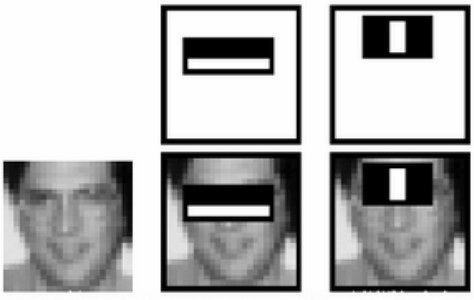

Fig. 25. Application of two Haar features, one to detect that the eye region is darker than the nose bridge and one to detect that the eye region is darker than the cheeks and nose regions [18]

Lastly there's the concept of **Cascade Classifiers**, per se. This comes from the fact that, on most images, there are more faceless regions than not, and as such applying all of our features to all of our image's sub-windows would be inefficient. Preferably we'd like to be able to discard the less-promising areas as soon as possible to avoid unnecessary computations, allowing us to focus on the regions that might actually have a face. Cascade Classifiers work by, instead of applying all features on a window at once and seeing the result, we group them into different stages of classifiers, which will then be applied one-by-one. Ideally the first classifiers should be of fairly low computation and act as a general filter for the most obviously faceless regions. As we progress in the step hierarchy the classifiers should be more strict and act to coarse out for more minute details. Originally, the authors of this algorithm used a 6000+ feature detector, divided amongst 38 stages with 1, 10, 25 and 50 features in the first five stages accordingly and managed to obtain over **95% accuracy** in a short amount of time.

B. OpenCV Implementation

The OpenCV library contains an implementation of the Cascade Classifier algorithm as proposed by Viola-Jones - the **cv::CascadeClassifier** class, which is capable of receiving and reading trained classifier models (stored in xml files) using the **cv::CascadeClassifier::load** method. Alternatively, it also provides a constructor which can also receive the model to be used. Both the method and the constructor receive as parameters the path to the file that contains the model. We can then utilize the **cv::CascadeClassifier::detectMultiScale** method to detect the objects within a given image. This function returns a list of boundary rectangles surrounding the detected faces, eyes, or both. As inputs, this method has the following parameters:

- **image** - A matrix of type CV_8U (a specific OpenCV binary image representation) corresponding to the image we want to detect faces on
- **scaleFactor** - Parameter that specifies how much the image is reduced at each image scale. This is used because OpenCV's cascade classifier implementation makes use of an **Image Pyramid** - a mutli-scale representation of an image done to make it so the face detection is *scale invariant*, i.e, we can detect both small and large faces using the same size of detection window. We could either due this or use a variable size window, which would be both more cumbersome and harder to implement.
- **minNeighbors** - Parameter that specifies how many neighbors each candidate rectangle should have to retain it. It will basically affect the quality of the detected faces. A lower value will cause more false positives, but a higher, more strict value will run the risk of creating false negatives.
- **minSize** - Minimum possible object size. Objects smaller than this are ignored
- **maxSize** - Maximum possible object size. Objects bigger than this are ignored.

Now that we know how we can use OpenCV's Cascade Classifier to detects faces, we are now ready to move on to actually implementing it. Before that, however, we would like to make it clear that this algorithm can be used, not only for Face Detection, but also for any other type of Object-Class Detection and localization problem, granted the appropriate model is trained or given.

C. Using a pre-trained model

Firstly, let's use one of OpenCV's pretrained models for face detecting using Haar features. There are several models available for free download and use in this repository. Due to being the one suggested for use on our problem, we chose the *haarcascade_frontalface_default.xml* model [4].

We started off by implementing a set of methods that would create a new cascade classifier and get the bounding rectangles around the faces of a given image. Using the default values of the *detectMultiScale* method we took 104.6 seconds to run through our complex test method and 117.4 seconds for our binary test. Overall the results obtained were somewhat underwhelming, as seen on tables IV and V. The algorithm seems to be disregarding a lot of the faces in the images, which is leading us to a high number of False Negatives, as well as identifying objects that are, indeed, not faces, which is increasing our False Positives. All of these factors lead us to very unsatisfying accuracy, precision and recall values. Figure 26 exemplifies how the algorithm is misclassifying regions as containing faces.

TABLE IV
NUMBER OF TRUE POSITIVES, FALSE POSITIVES AND FALSE NEGATIVES OBTAINED FOR THE TEST SET USING OPENCV'S PRETRAINED MODEL

Test	*T. Pos*	*F. Pos*	*F. Neg*	*T. Neg*
Binary Test	835	349	9194	9790
Complex Test	1252	3607	18753	-

TABLE V
ACCURACY, PRECISION AND RECALL VALUES OBTAINED FOR THE TEST SET USING OPENCV'S PRETRAINED MODEL

Test	*Precision*	*Recall*	*Accuracy*
Binary Test	0.705	0.083	0.530
Complex Test	0.257	0.06	-

There are two main parameters that we can change that will have a notable impact on our results - the **Scale Factor** and the **Minimum Neighbors**. Changing the **Scale Factor** will change the scale increment noted on each of image pyramid's steps. Basically our model has a fixed size (depending on the training it received) for the pattern it's trying to recognize faces with. As such, it means that it will be more capable of detecting faces of a certain size over others. By creating an image pyramid, however, and re-scaling our input image, we're effectively re-scaling the sizes of each face in the picture. This means that, even if for the original size a face may be too small or big to be recognizable, on the next resize iteration

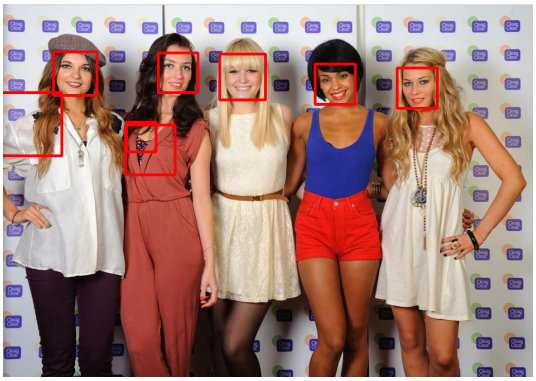

Fig. 26. One of the images we supplied. As we can see we're detecting all faces correctly, but we have some additional detection that are in fact not faces.

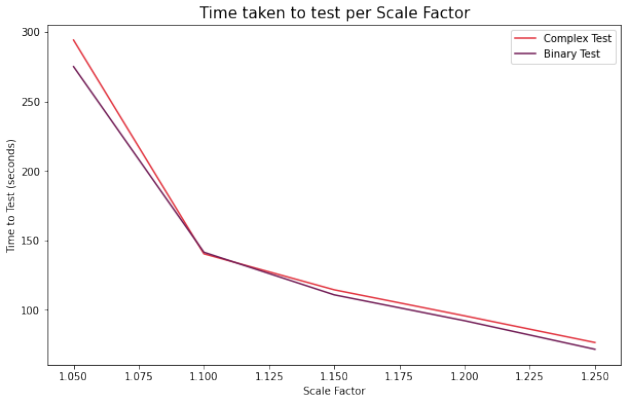

Fig. 27. The time in seconds that it took to go through our test set depending on the scale factor. The smaller the step, the more time it takes

it might be just right. By using a smaller step, presumably, we increase the chances of matching the face's size with that of the model, however, since we'll be taking smaller steps and doing more passes through our image it is also expected that it will take longer to work the testing algorithms on each individual image.

Indeed, figure 27 does prove that, the bigger the step, the less time it takes for us to run the Viola-Jones algorithm on our test images. Figures 28 seems to indicate that, although that for smaller steps we have quite the decent number of true positives, for our complex test we do have quite a big number of false positives and negatives. This discrepancy between the two tests may be due to the fact that, since on the complex test we're using complex images that may contain multiple faces of varying sizes scattered throughout the image, the scale factor will hold more influence over the results, than for the binary test (which uses simpler images that either do, or do not have a face).

Looking at figure 29 we can see that our precision seems to increase the bigger the step we take, whilst the recall and accuracy tend to have better values for the smaller steps. This can be easily explained by looking back at the values in figure 28. Our precision goes up because, whilst our true positives are taking a hit and decreasing, our false positives are shrinking way more substantially. The other two values can be explained in similar ways by looking at the rates at which the true and false positives and negatives are evolving.

Now, as for the **Minimum Neighbors**, this parameter specifies for each candidate bounding box how many other neighboring bounding boxes it needs to have in order to continue being considered a possible face ping. Basically, when our sliding window goes through the image and detects a face it draws a box around it. Depending on the stride this means that the same face can be inside the sliding window multiple times, causing multiple boxes to be drawn around it - neighbours. It's expected that this is the parameter that will affect our detection accuracy the most as having a higher minimum neighbor count will make the detection stricter leading to way less false positives. However, we run the risk of being too strict and discriminating too much, ending up with an increasing number of false negatives.

As figures 30 illustrate, we do end up creating way more false negatives (and by extension, less true positives), the more neighbours we use, but at the same time it is also observable that we're getting way less false positives and way more true negatives. This is due to the fact that, as aforementioned, we are being stricter with the faces we're detecting. Looking at the graphs in figure 31, our overall accuracy does suffer the more neighbours we're getting, because technically we are getting less correct predictions (due to our false positives). However our precision improves dramatically, with a notable spike at about 7 neighbours, which is caused by us having a higher proportion of positive identifications actually being correct (i.e, we're being more strict with our detection by forcing each identification to only be viable if surrounded by a bigger number of neighbouring boxes bounding to the same region, hence being more sure that each face we detect is an actual face). We should also note that the time it took to run the tests was also accounted for, and unlike the scale factor, this didn't seem to depend on the *minNeighbors*, and as such, to avoid visual clutter, that analysis has been omitted.

Looking through our obtained graphs, we believe that an acceptable value for our *minNeighbors* would be around **6**, since, although the accuracy may be rather low, our precision was pretty good. This means that we'll be, overall, running the risk of missing some faces, however, we'll be more sure that the faces that we do find are indeed faces. There's also the fact that there may be other reasons as to why our accuracy has been so low consistently through this section and that we may possibly be able to improve (as we'll explore in a bit).

For our *scaleFactor*, we chose to go with the default value of **1.10**. While it is true that for a smaller value, again our precision and recall are at their highest, it is also valid that our tests take much longer to run, and our precision is also a bit lower. As such a step that scales the image by 10% on each iteration comprises an acceptable balance of time to test and results obtained. Tables VIII, IX show our obtained results using these values, and whilst a good effort, we can't help but

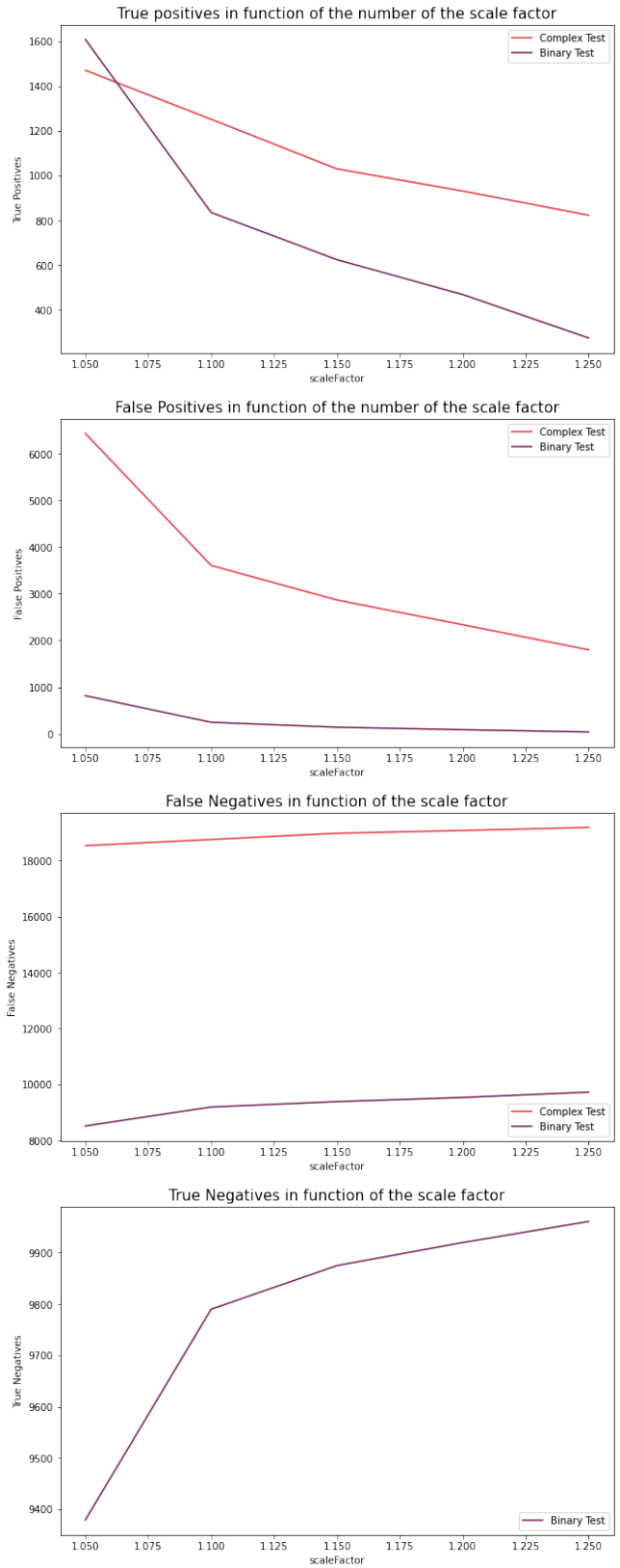

Fig. 28. The True Positives, True Negatives, False Positives and False Negatives obtained in function of the scale factor.

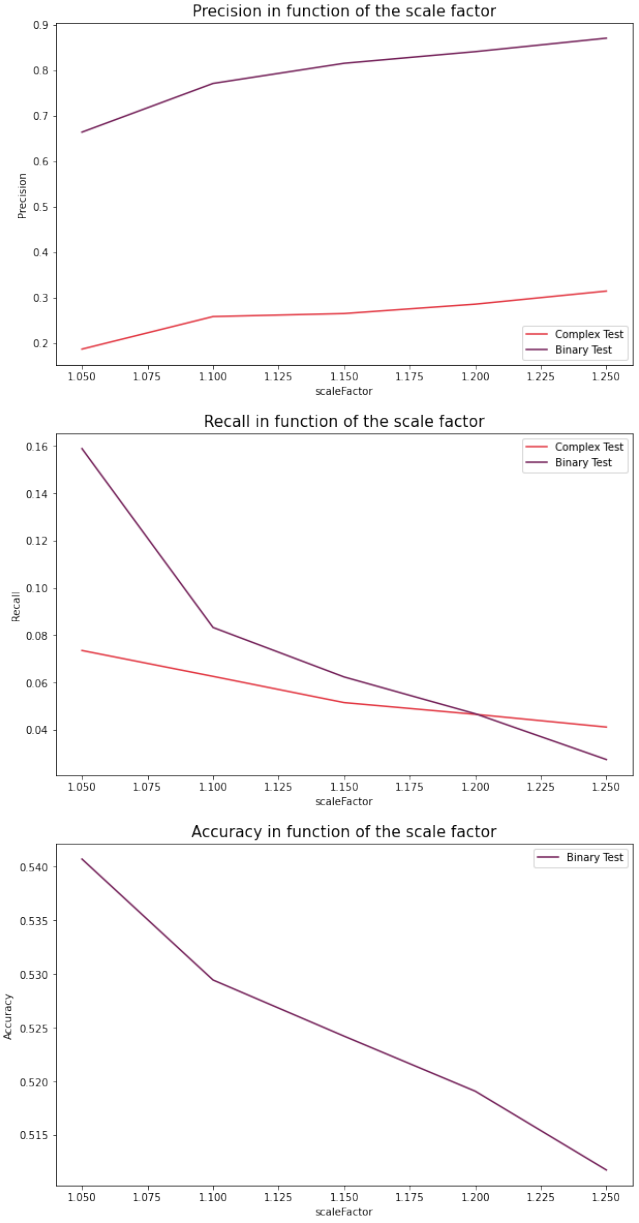

Fig. 29. Accuracy, Recall and Precision in function of the scale factor.

feel a bit disappointed by them.

Even though we managed to reduce the number of false positives by quite a big amount (we have 1465 less false positives - 40.6% decrease , but 377 less true positives - 30.9% decrease for our complex test ; for our binary test we have 449 less true positives - an unfortunate 34.7% decrease, but 290 less false positives - a positive 16.9% decrease), our true positives also went down unfortunately. Our true negatives, however, also showed a positive increase on both tests, caused by our more strict minimum neighbours required setting. Image 33 shows how with the right values, the same image's faces were now more aptly classified, whilst figure 47 shows a small sample of the results obtained in our complex test using these set parameters.

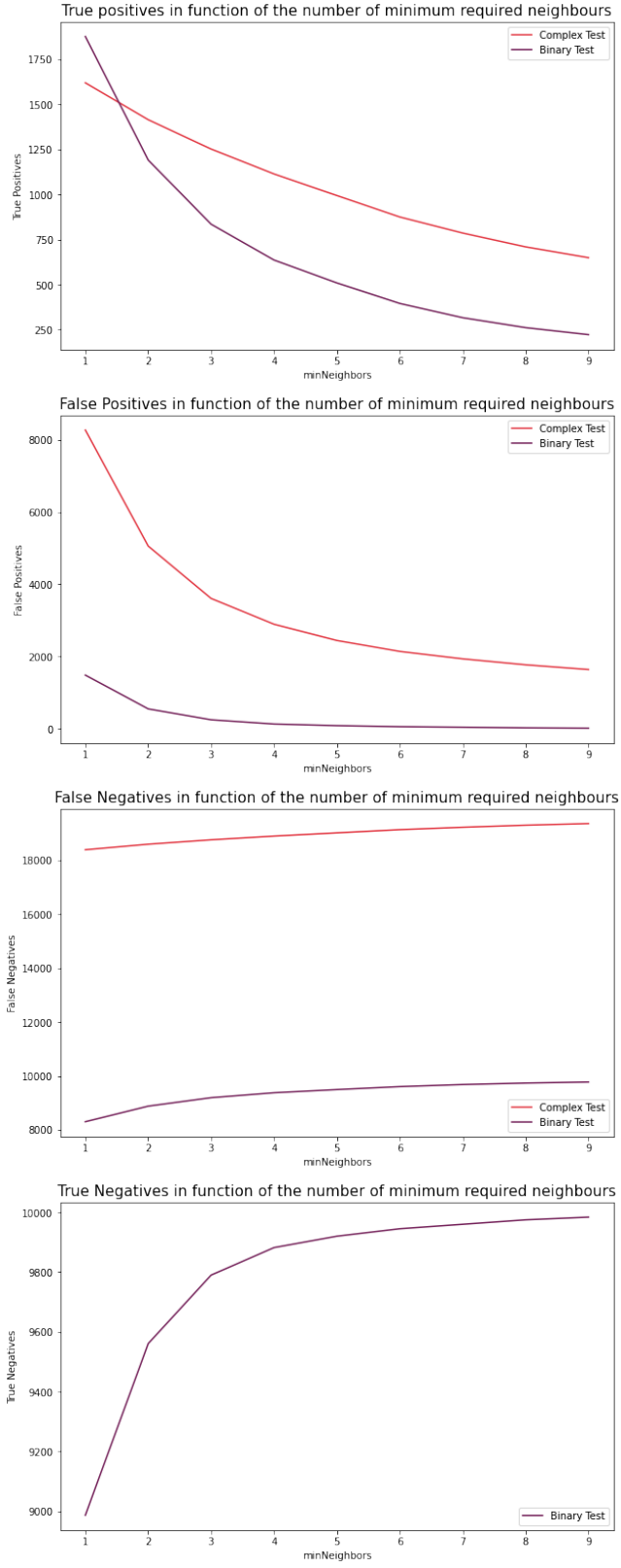

Fig. 30. The True Positives, True Negatives, False Positives and False Negatives obtained in function of the minimum neighbours variable.

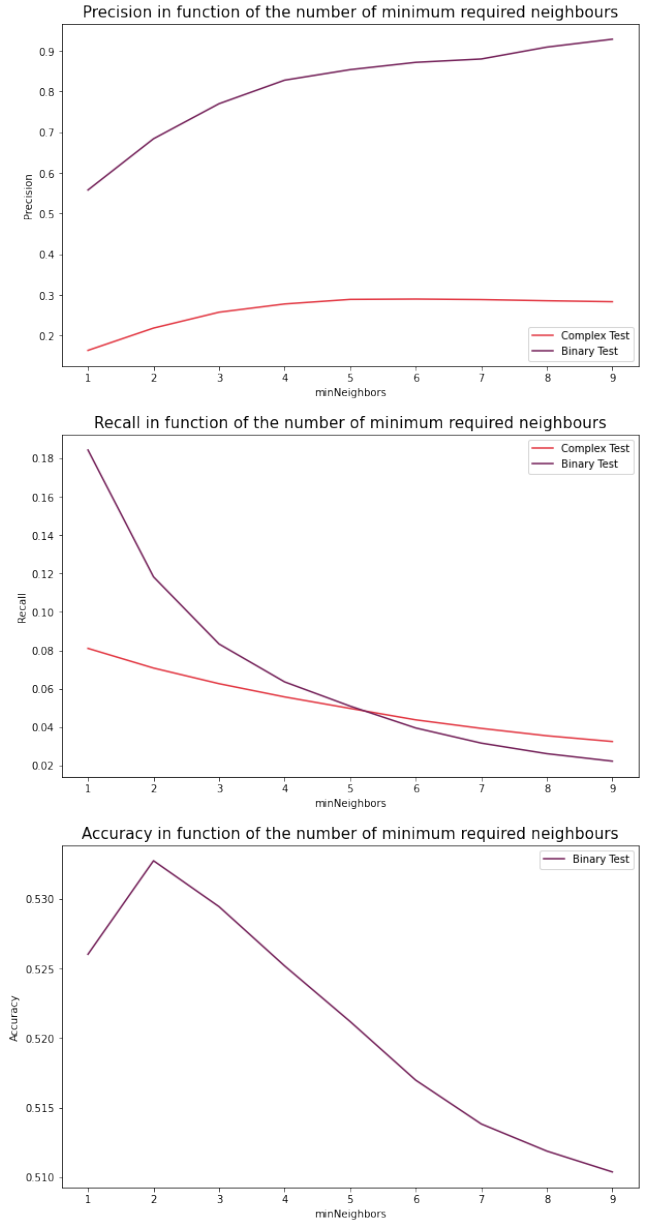

Fig. 31. Accuracy, Recall and Precision in function of the minimum neighbours variable.

TABLE VI
NUMBER OF TRUE POSITIVES, FALSE POSITIVES AND FALSE NEGATIVES OBTAINED FOR THE TEST SET USING OPENCV'S PRETRAINED MODEL WITH OUR FINAL CONFIGURATIONS

Test	***T. Pos***	***F. Pos***	***F. Neg***	***T. Neg***
Binary Test	395	59	9194	9603
Complex Test	875	2142	19130	-

TABLE VII
ACCURACY, PRECISION AND RECALL VALUES OBTAINED FOR THE TEST SET USING OPENCV'S PRETRAINED MODEL WITH OUR FINAL CONFIGURATIONS

Test	*Precision*	*Recall*	*Accuracy*
Binary Test	0.705	0.083	0.530
Complex Test	0.257	0.06	-

Fig. 32. A sample of our results obtained in the complex test using 6 minNeighbours and 1.1 scaleFactor.

But this was all using a pretrained model given to us by OpenCV, that was mostly trained to identify faces from a front-facing angle, which may, in fact, be one of the root causes leading to our accuracy being so low, as we're not being able to correctly detect faces that present themselves at odd angles or that might be partially obscured, hence causing our low true positive numbers. On the next section we'll be exploring how we trained our own model and what the results differences were.

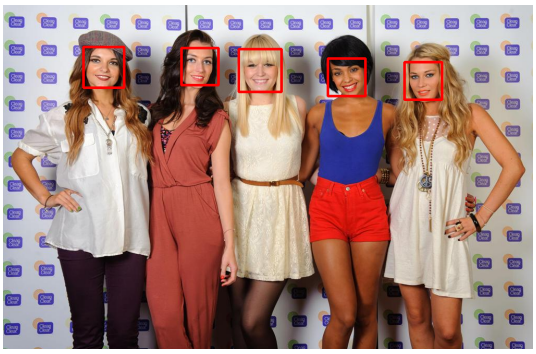

Fig. 33. The same image but now with the modified parameters. As we can see we're detecting all faces correctly on this particular image.

D. Training the Cascade Classifier

The results on the prior section were a good starting point, but they were made with a model that we did not train ourselves. Instead they utilized OpenCV's free pre-trained model for Frontal Face Recognition. This is a decent model, but it's far from perfect as it is ultimately not great at identifying faces at an angle and struggles with faces that aren't facing the camera as can be seen on image 34.

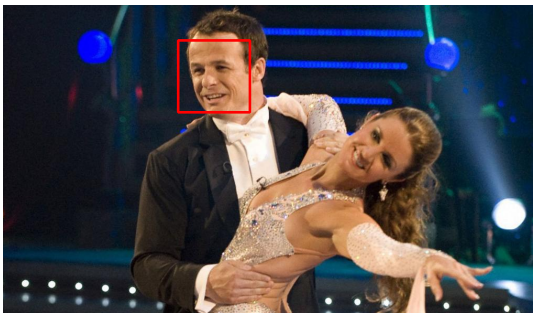

Fig. 34. With the pretrained model we were able to correctly find the upright left face, but since the other face is at a 90 degree angle we weren't capable of identifying it

Let us try to use our own dataset to train our model. To do this we will be using some tools made available by OpenCV on this link. More specifically we will be making use of *opencv_traincascade* and *opencv_createsamples* tools as per specified on the official OpenCV training documentation [19]. Utilizing our set of positive and negative train images (as specified in Section 3.) we start off by utilizing OpenCV's *opencv_createsamples* to transform our set of positive train images (as described on our Train set's *positive_images.dat* file) into a sample vector of 12000 24x24 grayscale images of faces. Note that for this training we don't actually need to manually divide our Train set to create a CV set as this tool aromatically performs a k-fold on the train samples.

Next we can actually start training our boosted cascade of weak classifiers. This can be done using OpenCV's *opencv_traincascade* by supplying it with the images we prepared. For it to work we need to supply it with both our positive images vector, as well as a text file containing the relative path to every single one of our negative images. The output will then be a fully trained model, in the form of an XML file, that can be used to detect and localize faces on images. The remainder of this section will be utilized to describe the results obtained from our Cascade Classifier, with the same parameters used in the prior section, changing the hyper parameters of our model's trainer.

*1) **Number of Stages**:* The first hyper parameter we'll be the testing, *num_stages*, changes the number of cascade stages (also known as steps or filters) that are to be trained. Each stage in the trainer follows the steps:

1) First grabbing a number of positive and negative images for the stage (on our trainers we chose to pick 1000 images from our set on each stage)

2) We take the first feature from the complete feature pool which allows us to classify the set of positive samples perfectly accurately.
3) Then we calculate the FA (False Alarms, or the proportion of negative samples that are incorrectly passed through) that this single feature yields on the negative samples and check if this is below our default max false alarm rate.
4) We iteratively add an extra feature from the feature pool that ensures the positives are still correctly classified and do not drop under the minimum hit rate. That also ensure that we have a drop in the FA rate of the negative samples.
5) We continue to add features until the maxFalseAlarmRate is exceeded.

When we then move on to the nest stage we:

1) Discard all positive samples that were wrongfully classified in the previous stage and swap them out for new images (which is why we don't give our trainer all our images on each stage's training).
2) Remove all negative samples that were correctly classified and grab new windows until we have as many positive images as we have negative.
3) Train the new stage.

Allegedly, the more stages our model has the faster our face detection should be as there will be more filters to more quickly coarse out less promising image regions. However, the more the number of stages the more time it will take for our model to actually finish training. Below, figures 35, 36 show the time it took to train the classifiers with a varying amount of stages and how much time it took to run the Cascade Classifier to detect faces. Looking at image 35 it does seem like the more stages there are the longer it will take to train our model, with the seconds required to train increasing exponentially as the number of stages increases. Surprisingly, however, the more stages we had the faster our detection actually were, as seen in image 36. This may be due to the fact that less stages struggle to quickly discard non-promising picture areas and, as such, have much more ground to cover, whilst more stages imply each region passes through more smaller computations, but overall avoid the high costs of the later classifiers and post processing (Note that we weren't able to try to detect the images on the test set using 5 or less stages because the time it took was insurmountable to our machines).

Meanwhile, in terms of results, figure 37 and 38 show that our accuracy and precision do tend to go up the more stages we have, however, after 25 stages we do see a drop in accuracy, most probably caused due to the higher number of filters starting to cause an overfit which in turn leads us to sub-optimal results in our test. Our recall is consistently going down, though. This may be because the more stages we have the more strict we're being, as we're running each image's regions through more filters to coarse out the least promising, which is causing an increase in false negatives. Please note that, due to time constraints, we were forced to use only the

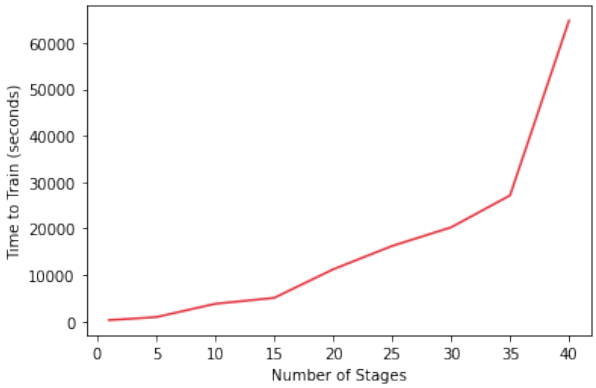

Fig. 35. The time in seconds that it took to train our model depending on the number of stages. These values were obtained running the train script on 1 thread in a Hexa-Core processor.

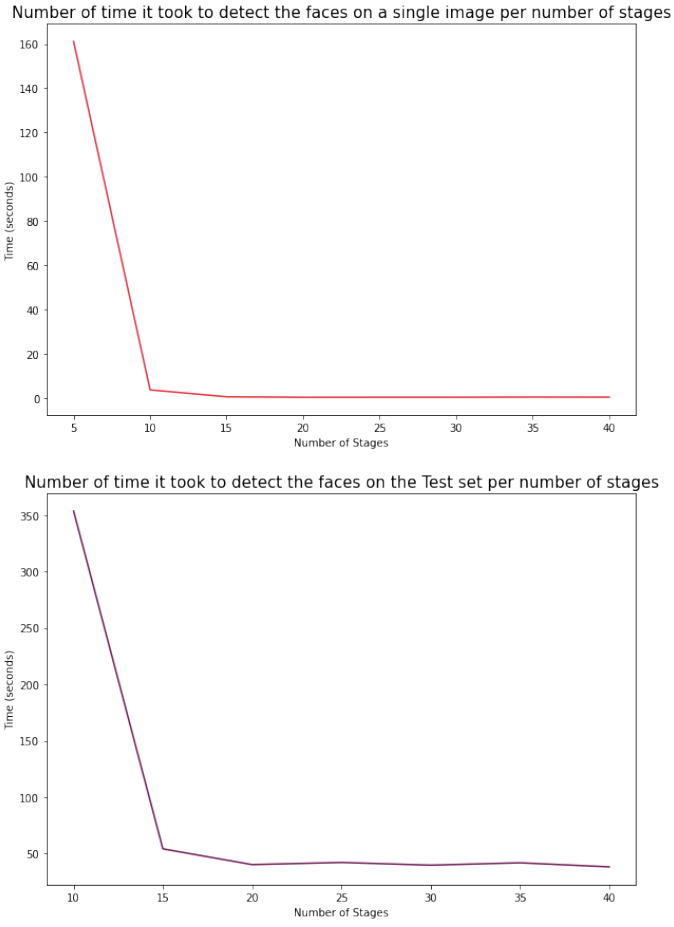

Fig. 36. Time taken to detect the faces in function of the number of stages used. The upper graph shows how much time it took to detect faces in a single picture, whilst the lower graph shows the time taken to detect the faces in 100 images

Binary Test in this section. When we showcase the results final hyper parameter choices, however, we'll be sure to include the complex test alongside the binary.

After this analysis, we have determined that the optimal number of stages would be around **27**, and that is what we'll be using on our final model. However, since this does take quite a lot of time to train (around 8 hours), for purposes of seeing the impact of the next hyper-parameters we'll be using and comparing them using only 15 stages in order to expedite the process (since, as proven in figure 35, this is the point where the time it takes to train starts becoming acceptable).

2) ***Minimum Hit Rate****:* Next we have the *min_hit_rate* which specifies the minimal desired hit rate that we want to aim for on each stage, i.e, the minimum percentage of positive images that are correctly classified as such (i.e, our minimum precision on the train set). By default this is set to 0.995, which means that we only allow 0.05% of our positive images to be misclassified before moving on to training the next stage. Beforehand, we presume that increasing this number will naturally decrease the number of false negatives on our test set hence leading us to a more accurate model, however it will also most likely increase the time it takes to train our classifiers since we're aiming for a higher demand of positive identifications. Let us see if these statements hold true by testing the minimum hit rates of 0.95, 0.99, 0.999 and 0.9995. Indeed figure 39 does show that the higher the minimum hit rate the more time it will take to fully train our model. However, it's rather perplexing that increasing this parameter also causes the time it takes to detect faces on our images also increases. This, however, can be explained by the fact that we're training our weak classifiers to be stricter, which will cause, even the first ones which should be more light weight, to require more computations before discarding a region as non-promising.

Now analyzing figures 41 and 42 we can also confirm our assumptions. The higher our minimum hit rate, the more true positives we're going to be able to detect, and our false negatives also drop considerably. However, our false positives also seem to increase the higher our minimum hit rate, which is leading us to a drop in precision.

By looking at our graphs, we decided to proceed with using a minimum hit rate of around **0.99**, as this seems to strike a good balance of accuracy, precision and recall.

3) ***Maximum False Alarm Rate****:* Finally we have the *max_false_alarm_rate*, the maximum desired false alarm rate for each stage of the classifier, i.e, the max percentage of negative images that we classified as positive on each stage. The smaller this number the more complex each individual stage needs to be as they will have to reach a better classification of guessing. By default this is set to 0.5, which is just a bit better than randomly guessing whether a non positive region indeed contains or not a face. With this in mind we presume that increasing this number will lead to faster training but negative regions will propagate to later stages with a higher probability.

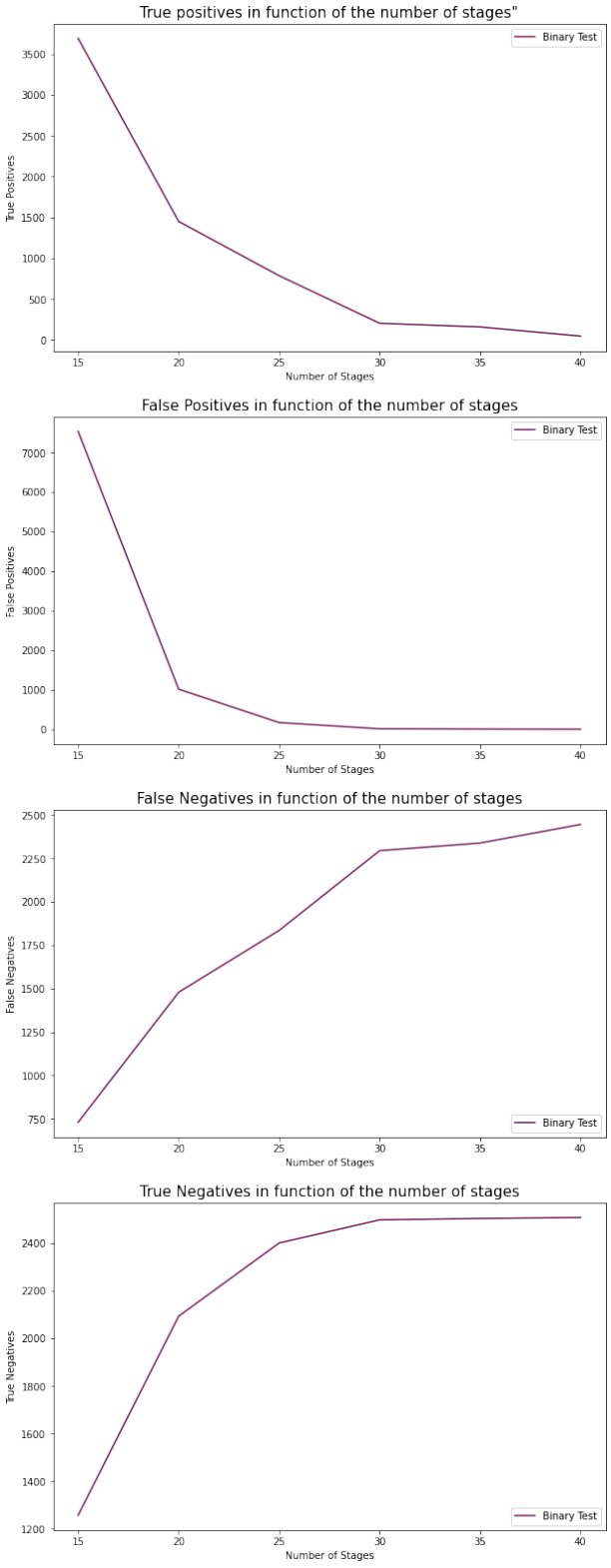

Fig. 37. The True Positives, True Negatives, False Positives and False Negatives obtained in function of the number of stages used in the trainer.

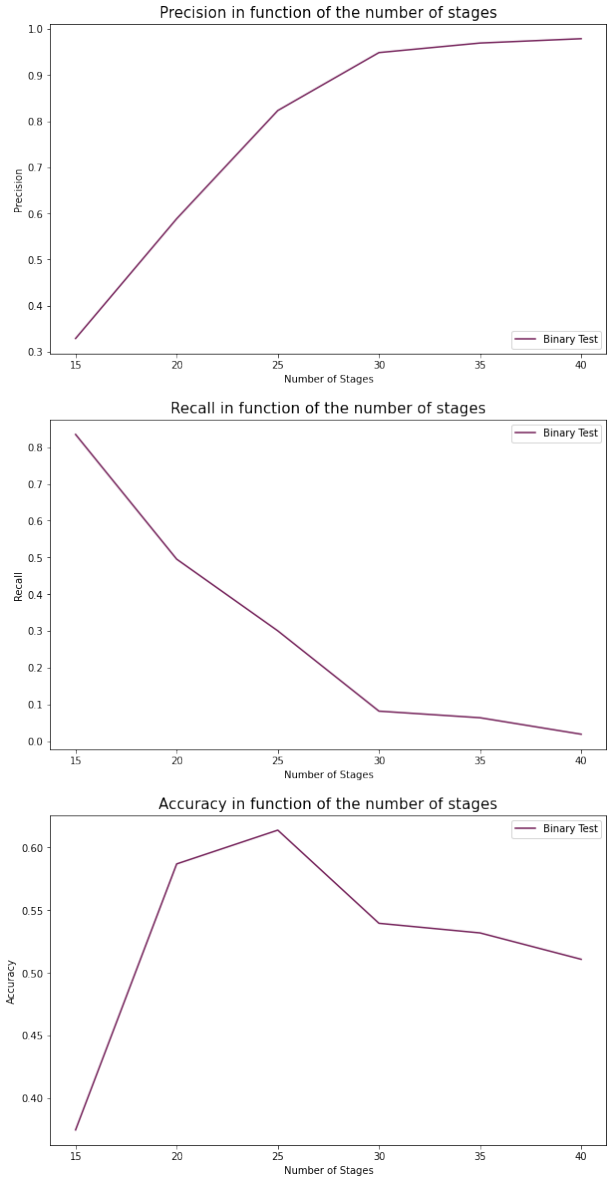

Fig. 38. Accuracy, Recall and Precision in function of the number of stages used in the trainer.

To test this out we chose to use *max_false_alarm_rate* values of 0.3, 0.4, 0.5 and 0.6. Figure 43 does prove that the smaller this value is, the more it will take to train our model, and this seems to decrease linearly the higher we set the max false alarm to. Meanwhile figure 44 is rather perplexing. It seems that the higher our max false alarm rate is, the more it will take to run the test as for smaller values our model will be capable to more quickly dispose of non-promising regions, which may lead us to a faster discarding of images, hence the algorithm runs faster.

Looking at figures 45 and 46 we can see that the maximum false alarm rate does have quite the substantial impact on our model's performance, although perhaps not in the expected way. Both our accuracy and precision seem to achieve optimal

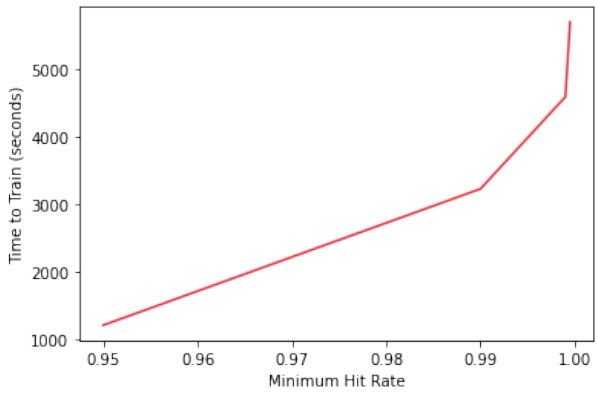

Fig. 39. The time in seconds that it took to train our model depending on the minimum hit rate. These values were obtained running the train script on 1 thread in a Hexa-Core processor.

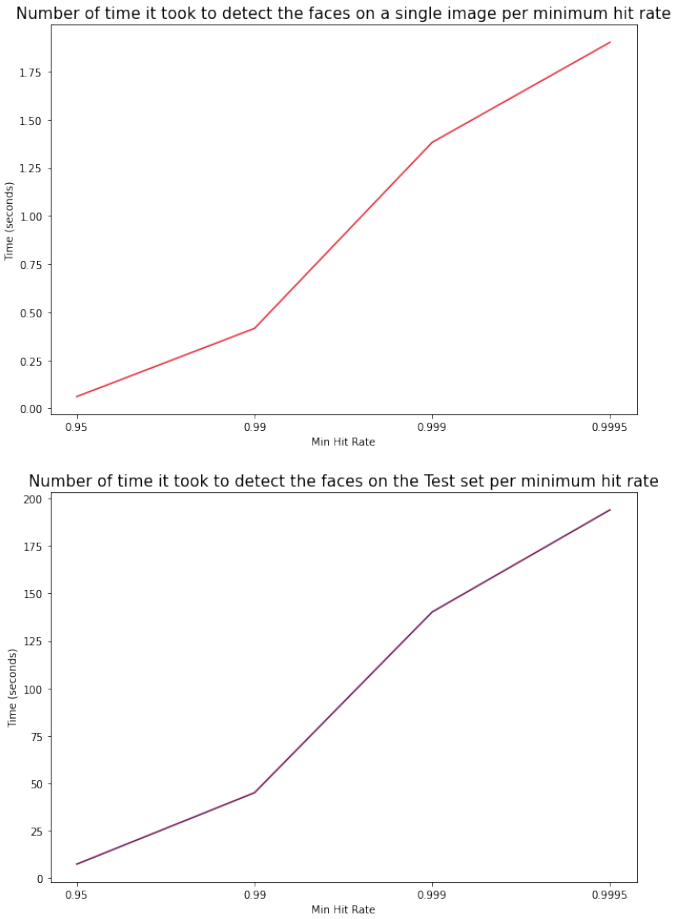

Fig. 40. Time taken to detect the faces in function of the hit rate used. The upper graph shows how much time it took to detect faces in a single picture, whilst the lower graph shows the time taken to detect the faces in 100 images

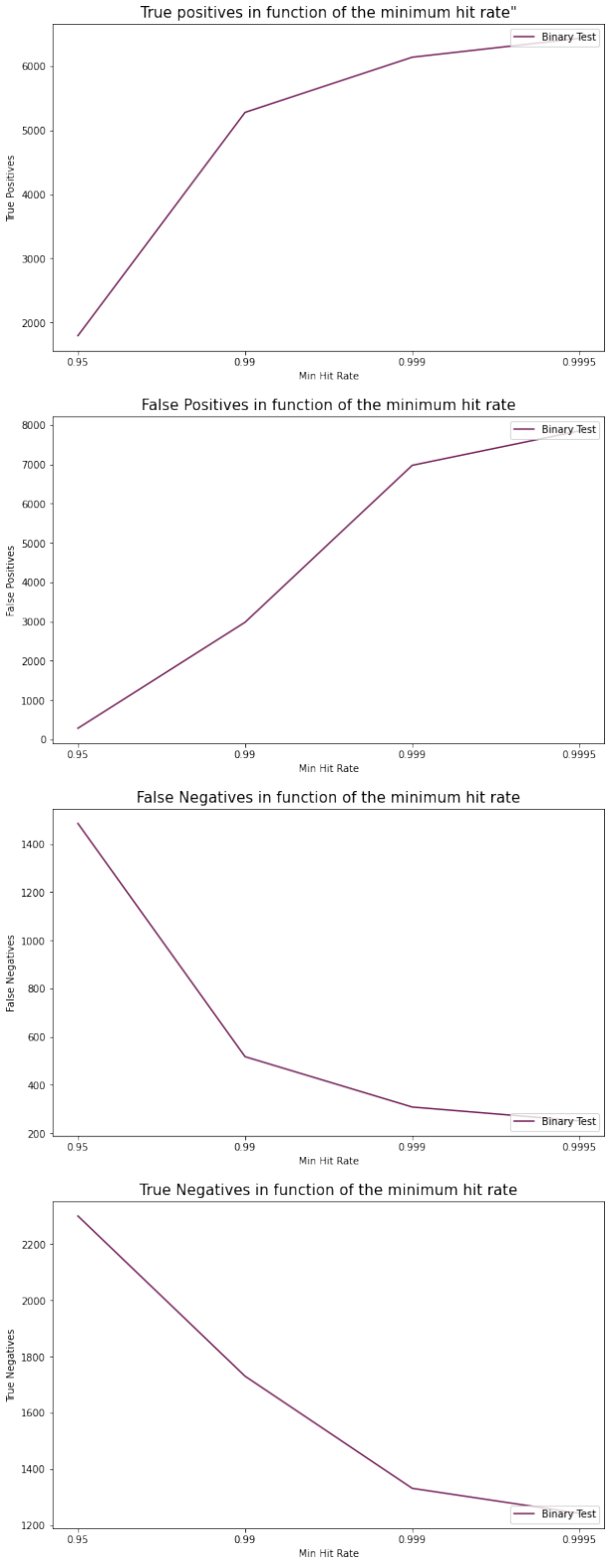

Fig. 41. The True Positives, True Negatives, False Positives and False Negatives obtained in function of the minimum hit rate used in the trainer.

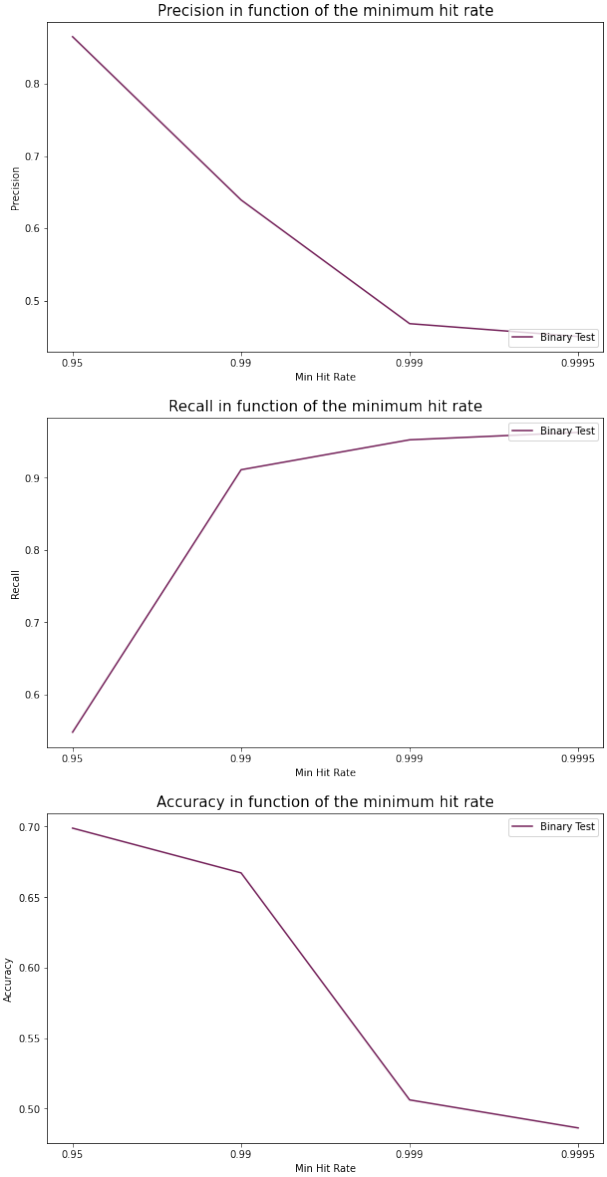

Fig. 42. Accuracy, Recall and Precision in function of the minimum hit rate used in the trainer.

values for the lower maximum false rates, with precision maxing out at 0.3 and accuracy reaching its apex at 0.4. Our recall however, seems to be the complete opposite as the higher the maximum false alarm rate the better it gets. Looking at our true positives this does seem to make sense. The bigger our max false alarm rate the more lenient we are, hence we'll end up correctly identifying most faces, hence the proportion of true positives found increases (i.e the recall). The caveat, however, is that by being more lenient we also increase the number of false positives which causes both our precision and accuracy to go down.

A good value for our maximum false alarm rate seems to lie between 0.35 and 0.45 since this seems to be a good compromise between precision, recall and accuracy values. So

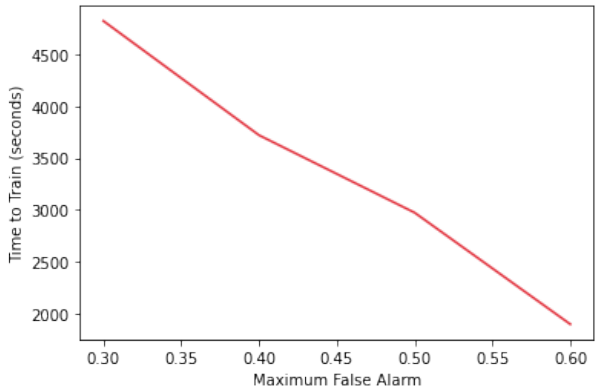

Fig. 43. The time in seconds that it took to train our model depending on the maximum false alarm. These values were obtained running the train script on 10 threads in a Hexa-Core processor.

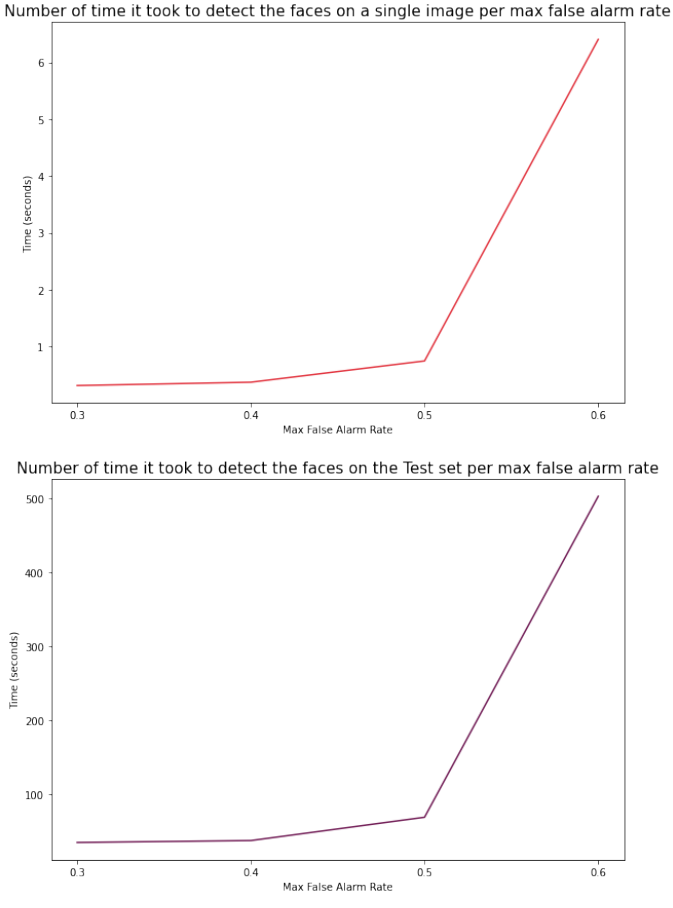

Fig. 44. Time taken to detect the faces in function of maximum false alarm rate used. The upper graph shows how much time it took to detect faces in a single picture, whilst the lower graph shows the time taken to detect the faces in 100 images

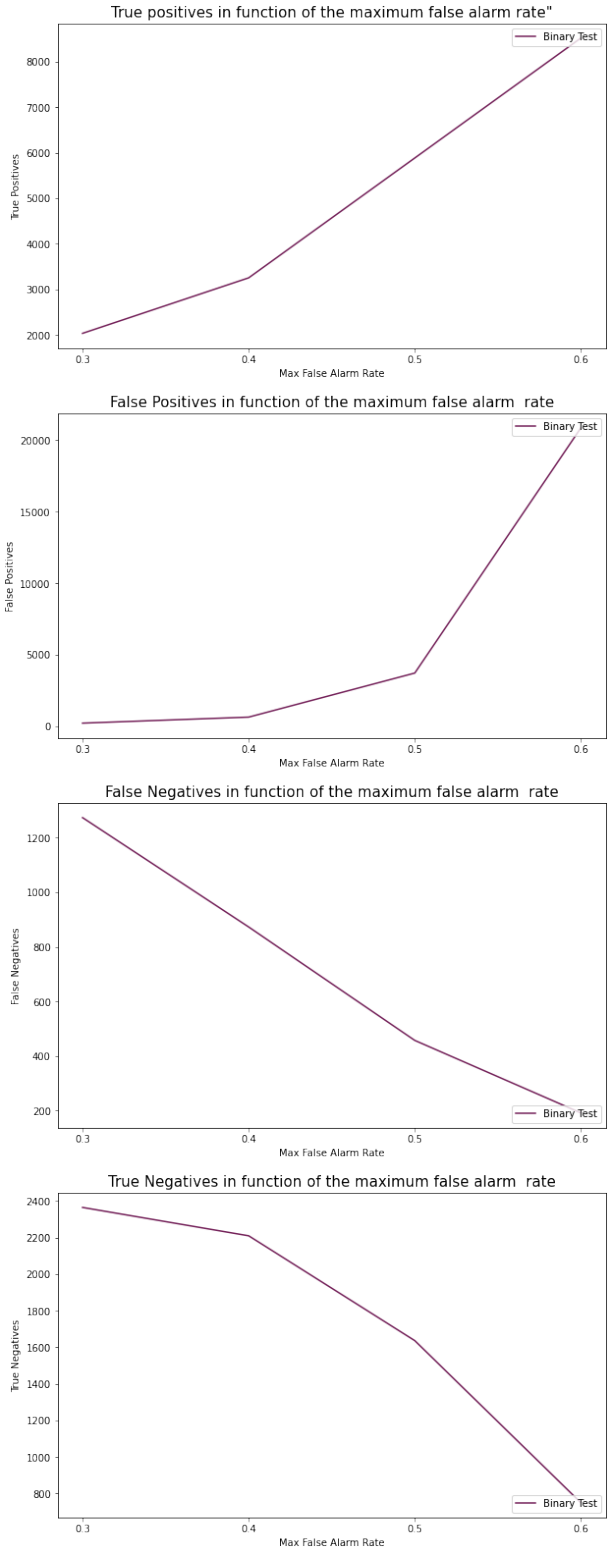

Fig. 45. The True Positives, True Negatives, False Positives and False Negatives obtained in function of the maximum false alarm rate used in the trainer.

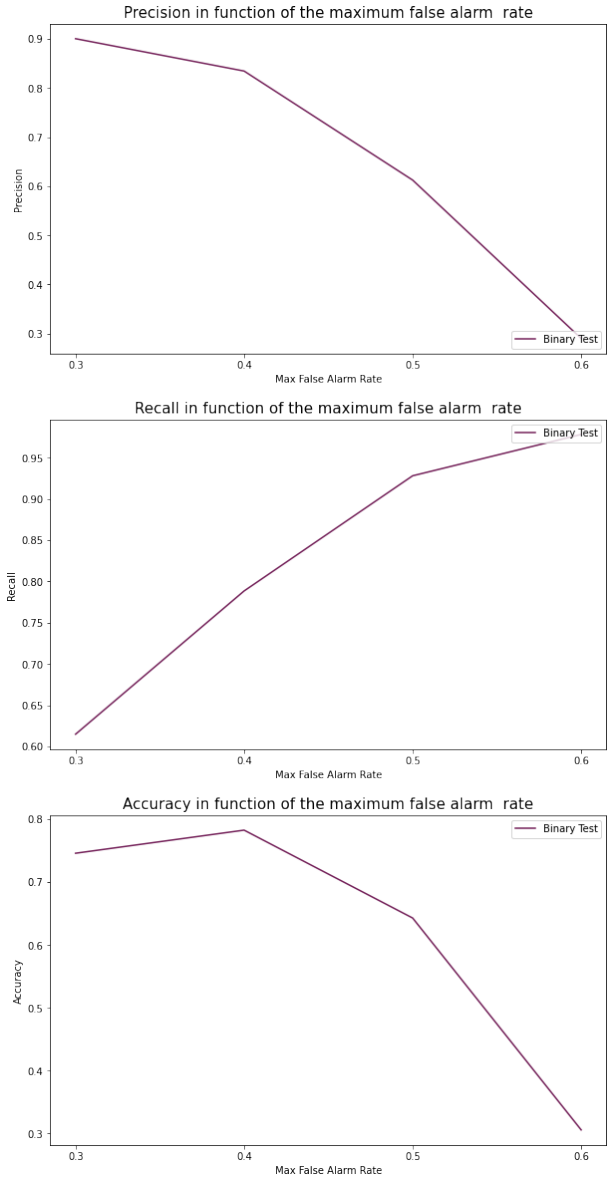

Fig. 46. Accuracy, Recall and Precision in function of the maximum false alarm rate used in the trainer.

onward we'll be utilizing **0.4** for our final model, meaning that we discard possibly negative regions of an image with a probability slightly better than just 50/50 (i.e random guessing).

*4) **Final Model**:* For our final model we trained for our Cascade Classifier, and after taking into account all of the aformentioned analysis, we decided to go with **25 stages**, **0.99 minimum hit rate** and **0.4 maximum false alarm rate** and to see how it compares to the pre-trained model's final results. Tables VIII and IX are a bit underwhelming. Although its true that we managed to obtain better results than the pretrained model given to us by OpenCV, we didn't managed to beat the aforementioned value of 0.95 accuracy that the original authors, Viola and Jones, managed to achieve. This may be due to several factors, from our combination of hyper parameters, to the images we used to train our algorithm with.

TABLE VIII
NUMBER OF TRUE POSITIVES, FALSE POSITIVES AND FALSE NEGATIVES OBTAINED FOR THE TEST SET USING OUR OWN TRAINED MODEL WITH 25 STAGES, 0.99 MINHITRATE AND 0.4 MAXFALSEALARM

Test	***T. Pos***	***F. Pos***	***F. Neg***	***T. Neg***
Binary Test	1794	9	8379	9992
Complex Test	2123	2885	17882	-

TABLE IX
ACCURACY, PRECISION AND RECALL VALUES OBTAINED FOR THE TEST SET USING OPENCV'S PRETRAINED MODEL WITH OUR FINAL CONFIGURATIONS

Test	***Precision***	***Recall***	***Accuracy***
Binary Test	0.995	0.177	0.584
Complex Test	0.424	0.11	-

What's more astonishing, however, is the fact that in a prior section we managed to obtain results better than these! When we had a combination of 15 stages, 0.995 minimum hit rate, and 0.4 max alarm rate we managed to achieve a precision of over 0.82, recall over 0.77 and accuracy of around 0.78. A probable cause for this may be because of the way the maximum alarm rate hyper parameter affects the manner in which less promising regions get discarded. By default a region that's deemed improbable of containing a face gets discarded with a probability with 50%, by reducing it to 40% (with our 0.4 value) we're being more discriminate against these regions. By having more stages, the possibility of one of those regions being taken out of the picture is even greater, which may lead to an increase in false negatives, hence why our performance may have gone down.

V. HOG WINDOWS WITH SVM CLASSIFIER

A. Introduction

Moving on from exploring our cascade classifiers and the Viola-Jones algorithm, we would like to dedicate a section of this report into scratching the surface of what is possible to achieve utilizing a State Vector Machine classifier and extracting our image's features using HOG Windows. Our first step into implementing this type of face detection and localization algorithm was to apply some preprocessing to both our train dataset. To extract our HOG features off of our train set we first utilized OpenCV to both convert both our positive and negative images into grayscale, and to reshape them into 128x128 samples that can then have their HOG Features computed.

As a reminder of how HOG works, it comes from the base idea that we can discern patterns capable of being fed into a classifier by extracting the distribution of directions of gradients in the images. Basically we want to, for each pixel, see how it compares in terms of brightness to it's neighbours. This works because objects tend to be affected by light in

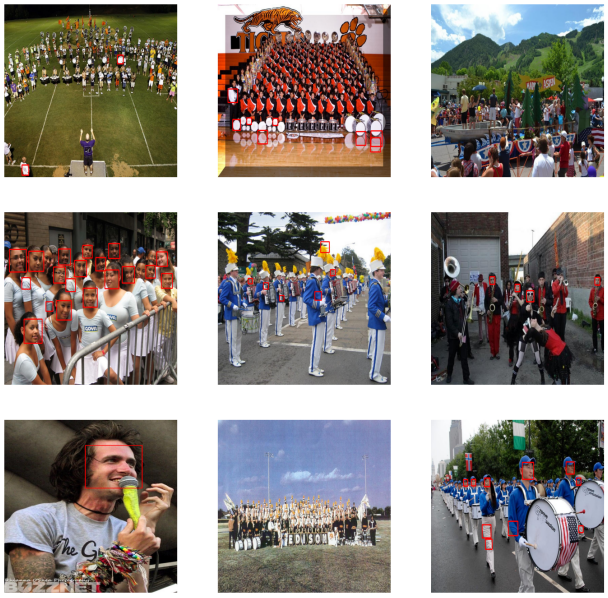

Fig. 47. A sample of our results obtained in the complex test using our own model.

similar ways. For faces, in particular, certain features like the eye socket region being darker than its surroundings, and the nose bridge being brighter are some very common patterns. The fact we look at the changes in lightning comparatively to the image's own pixels also allows us to theoretically identify faces regardless of overall image brightness, since as long as the patterns are kept we'll still be able to detect their presence. Figure 48 shows, for a picture of a face, how its corresponding gradients seem to almost act as a bezel around the main features. We can clearly see how we get rid of most information but the base feature outlines, such as the face outline, the eyes, nose, and so on.

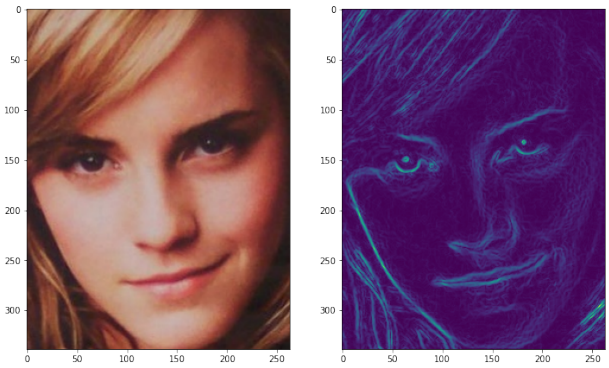

Fig. 48. A side by side comparison of a normal image and its computed gradients

After we get the gradients we can compute the so called **History of Oriented Gradients**. We do these because gradients themselves give us too much unnecessary information that would end up increasing the computations needed to use them as our patterns. What we want isn't to know whether a pixel is brighter or not than its neighbours, but the overall direction in which light seems to be moving in the picture. The base form we achieve this is by recurring to **Skimage**'s *feature.hog* method. What this does is divide the image into several sections of 16x16 pixels (hence why we chose to have our samples be of 128x128, so that we could divide them into 8 clean regions). For each of these sections we count how many gradient points in each direction there are. That is, we count how many max gradients we have going up, down, left, right, up right, down right, up left and down left. We can then use this information to build a histogram of oriented gradients in order to compute which direction has the most gradients. Knowing this we can then replace that region with its its common gradient direction (i.e, change the pixels in that area so that we have the max gradients going in the corresponding cardinal direction). Figure 49 shows what the HOG representation of an image might look like. Note, however, that this image has not been resized, unlike our train set, and as such we aim to simply provide a visual representation of what the HOG features look like, and how they can capture an object's patterns. In this particular case we can clearly identify patterns such as the eyes, the outline of the nose and mouth, and so on.

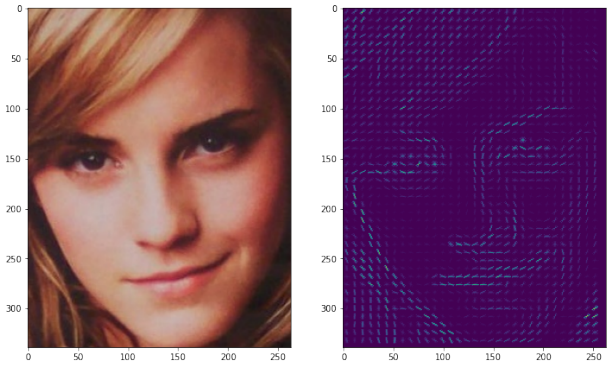

Fig. 49. A side by side comparison of a normal image and its HOG visualization

The *feature.hog* method returns both a visualization of the image's HOG counterpart, which can be used for display purposes, as has been done in figure 49, but also a vector of the HOG features that we can then supply to our classifier. In our case, the classifier is going to consist of a simple State Vector Machine Classifier implemented using the **Sklearn** library. For purposes of testing we'll be trying out both a normal SVC, as well as a Linear SVC, to which we'll be supplying our set of positive and negative image's HOG features, obviously shuffled. It should be noted that once again we do not require to give our SVMs a specific CV set, as the library itself takes care of performing a K-Fold on our Train set in order to create it.

B. SVC

We started off by creating a model using **Sklearn**'s SVC class. This implementation of a Support Vector Classifier is

a rather simple one and it aims to to basically fit the data we provide, and return a "best fit" plane that divides our data by their categorization, i.e whether they have a face (1) or they don't (0). After we get this plane (i.e the model has been trained) we can feed it some features (in our case, our test image's HOG feature vector) and the classifier will attempt to predict the image's class (which in our case can either be 0 or 1, hence why the problem of face detection is technically one of binary classification).

By default, our SVC implementation will be using the **Squared Hinge Loss Function**, a squared variant of the Hinge loss function which is considered to be the standard SVM loss function. One of the main hyper parameters that can be changed in this classifier is its **C** parameter. By default this value is set to one, and the higher we set it, the weaker the regularization strength will be. We started off by, obviously, getting the HOG features of all of our 12000 positive and 12000 negative images, classifying them as such (with positives being labeled as 1 and negatives as 0), and then shuffling the data. Afterwards we fed this data directly to our SVC. Note that we don't actually have to create a CV set ourselves as, using our train data, the sklearn::SVC module will automatically create a 5-fold on it. We then fit our SVC module to our train data and tested for several possible **C** values (1.0, 2.0, 4.0, 8.0), and ascertained, using Sklearn's *best_params* method, that we had our best results for a C of 4.0.

After this we did some small adaptations to our **Binary Test** since we had to convert and compute the test image's HOG features in order to try to classify them. We also had to create a **Sliding Window** function that would create a window, the size of our samples (128x128) and go through the entire image (with a stride of 2 pixels) to try to find a face in. Unfortunately we were incapable of using our **Complex Test** due to the fact that, when detecting faces using the sliding window, our algorithm will find the same face multiple times and draw multiple bounding boxes surrounding the same face (due to the our small stride value, which was, however, necessary). Normally, and this is what happens with algorithms like OpenCV's *detectMultiScale*, if multiple bounding boxes are around a face, they get merged into a single one, and as a matter of fact, for a face to be confirmed as a face it needs to have a minimum number of neighbouring boxes around it. Due to time constraints we weren't able to perform this adjustment, and as such, when our window slides through a face it will create multiple boxes around it, rendering the **Complex Test** unusable.

Tables X and XI show that this methodology can produce results that are nothing short of astonishing. Despite the fact that the process of training and testing do take a while to conclude, since having to compute each image's HOG features and having to use a sliding window to walk through the test images does take a while (in total it took us **265.567 seconds** to run our binary test), the train time was still much much smaller than our Cascade Classifier's models training, and the test times weren't too bad, being on par with some of the Cascade Classifier's models tests. The fact we got an accuracy, precision and recall values that are all over 95% is the truly remarkable thing to notice.

TABLE X
NUMBER OF TRUE POSITIVES, FALSE POSITIVES AND FALSE NEGATIVES OBTAINED FOR THE SVC HOG FEATURE CLASSIFIER

Test	***T. Pos***	***F. Pos***	***F. Neg***	***T. Neg***
Binary Test	9587	229	409	9772

TABLE XI
ACCURACY, PRECISION AND RECALL VALUES OBTAINED FOR THE TEST SET FOR THE SVC HOG FEATURE CLASSIFIER

Set	***Precision***	***Recall***	***Accuracy***
CV Set	-	-	0.974
Binary Test	0.976	0.959	0.968

C. Linear SVC

Both SVC and Linear SVC perform in similar ways, they're both Support Vector Classifiers after all. But allegedly the SVC has a *"fit time that scales at least quadratically with the number of samples"* [26], making it impractical when we have a lot of train samples. As such we thought it would be wise, since we have around 20000 total samples, to also test out the Linear SVC module from Sklearn which is similar to SVC but with the parameter kernel (one of SVC's possible parameters) set to **linear** and implemented in a way that grants greater flexibility and scalability for a larger number of samples.

We gave our Linear SVC the same loss function and messed around with the **C** parameter in the exact same way as we did with the regular SVC. As we can see in table XII the Linear SVC manages to finish training 8 times faster than our normal SVC, which does make point as to why it should be used.

TABLE XII
TIME TAKEN (IN SECONDS) TO FIT AND TEST OUR MODELS

SVM	***Train Time (s)***	***Test Time (s)***
SVC	96.77	265.567
Linear SVC	12.45	214.641

Despite the faster training and testing times, however, the Linear SVC is incapable of producing as good results as our normal SVC. Looking at tables XIII and XIV all of our results went down comparatively to the ones presented in the prior sub-section, with the most notable change being the fact that we have way more false negatives. This was to be expected as the Linear SVC produces a much simpler fit in comparison to the normal SVC.

VI. MTCNN

All the latest improvements in the face detection problem seem to be revolving around an implementation of multi-task cascaded neural network, an **MTCNN**. So we thought it'd be fitting to check the results the basic face detection algorithm

TABLE XIII
NUMBER OF TRUE POSITIVES, FALSE POSITIVES AND FALSE NEGATIVES OBTAINED FOR THE LINEAR SVC HOG FEATURE CLASSIFIER

Test	*T. Pos*	*F. Pos*	*F. Neg*	*T. Neg*
Binary Test	8642	450	1354	9551

TABLE XIV
ACCURACY, PRECISION AND RECALL VALUES OBTAINED FOR THE TEST SET FOR THE LINEAR SVC HOG FEATURE CLASSIFIER

Set	*Precision*	*Recall*	*Accuracy*
CV Set	-	-	0.925
Binary Test	0.951	0.864	0.901

could perform. For this, we used the python module MTCNN [24], using the default settings and training weights.

After analysing the implementation present in the GitHub repository [25], the MTCNN is structured with three different neural networks, which with their specific function [27], as can be seen in the Figure 51.

- First, one NN, composed of four convolutional layers with PReLu function between each layer, that would recreate the image on different sizes and create image sections of 12x12 to serve as input, the output will discard any low confidence kernels and return the most likely to have faces.
- Second, another NN, with two convolutional layers, one flat layer and three dense layers, the output of these will be the same as the first NN; however, for better results, a bit of preprocessing is done to the images, i.e. add padding to be a 24x24 image, and normalize all values to be in a -1 to 1 range
- Third, the last NN is composed of four convolutional layers, and four dense layers, with a softmax activation function near the end; it will resize the boxes to a 48x48 picture and return the final values for the face detection of the image.

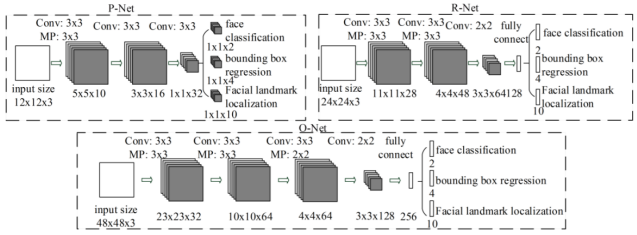

Fig. 50. Structure of an MTCNN face detection algorithm

The return values of the face detection isn't just coordinates and dimensions of the bounding box, but also a list of key points in the face, i.e. nose, eyes and corners of the mouth, and confidence level the model has for the face in the picture, as can be seen in Fig. 51. In these tests, we've only been checking for the bounding box they create, not the rest of the features, but it is interesting to note that the model is prepared to return other important features.

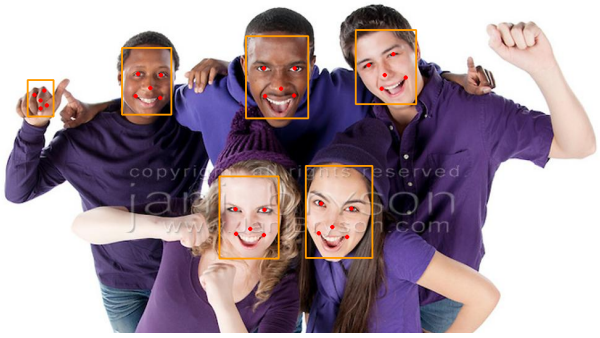

Fig. 51. Results of the MTCNN face detection algorithm

To test the model, we used both the complex tests, and the simple binary tests. The results of both tests are shown in the tables XV, XVI. It's important noting that it only takes 330.16s to process all images on the Complex Test, with good precision rounding at 86%; as for the Binary test, it only took 334.68s to process and get quite results, with an accuracy of around 84% and near 100% precision.

TABLE XV
NUMBER OF TRUE POSITIVES, FALSE POSITIVES AND FALSE NEGATIVES OBTAINED FOR THE TEST SET USING MTCNN WITH THE DEFAULT WEIGHTS

Test	*T. Pos*	*F. Pos*	*F. Neg*	*T. Neg*
Binary Test	6874	39	3128	9964
Complex Test	7069	1128	12936	-

TABLE XVI
ACCURACY, PRECISION AND RECALL VALUES OBTAINED FOR THE TEST SET USING OPENCV'S PRETRAINED MODEL WITH OUR FINAL CONFIGURATIONS

Test	*Precision*	*Recall*	*Accuracy*
Binary Test	0.994	0.687	0.842
Complex Test	0.862	0.353	-

These results mean that the model is actually quite good in terms of performance, as it didn't need any pre-processing for the images, and it could quite quickly detect images. It also showed that it may not take the risk of detecting faces, but the faces it detects can be trusted to be correct.

VII. FINAL CONCLUSIONS AND ABOUT FUTURE IMPROVEMENTS

In this report we have talked at length about the algorithms we have implemented, how they work and the results each model has produced.

We have seen that the cascade classifier had the worse results in terms of accuracy, recall and precision: our best model reached some rather underwhelming results, although its true that we did manage to do better than the pretrained model; this makes sense considering it's the oldest algorithm we studied, being created in the 2000's, after which a lot of technical

and engineering breakthroughs have been achieved. The most egregious fact about it is that it takes a lot of computational resources to train and passing all images through several weak classifiers also brings the test time down.

We have also seen that the HOG processing with an SVM reached some pretty high scores in terms of precision and recall, but fails in trying to identify faces in a regular photo, i.e. while it is pretty good at telling whether the picture its looking at is or isn't a life, it can't locate the face itself. For that, we had to use a sliding window approach, which can be pretty taxing to the computer itself. In our point of view, a very good model could be made by joining the YOLO algorithm, having a neural network that will divide the image in various regions, with the HOG algorithm to classify does regions.

Finally, we have studied what is now the basics for any model for face detection, an MTCNN algorithm. While the values in the binary test didn't reach HOG's, it was the best model in terms of performance and getting results that we have implemented, reaching a precision of 86% while only taking slightly more than five minutes in our machines. The algorithm itself was implemented in 2016, and since then a lot of different variations have been implemented, e.g. Faceness Net which uses these types of models on facial features to detect faces.

References

[1] Jason Brownlee, "How to Perform Face Detection with Deep Learning", 2019
[2] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, 2014
[3] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, Gang Hua, A Convolutional Neural Network Cascade for Face Detection, 2015
[4] Maël Fabien, A guide to Face Detection in Python, 2019
[5] Shuo Yang, Ping Luo, Chen Change Loy, and Xiaoou Tang, Faceness-Net: Face Detection through Deep Facial Part Responses, 2017
[6] Adam Geitgey, Machine Learning is Fun! Part 4: Modern Face Recognition with Deep Learning, 2016
[7] Chi-Feng Wang, How Does A Face Detection Program Work? (Using Neural Networks), 2018
[8] Ming-Hsuan Yang, Facial Recognition Using Kernel Methods, 2018
[9] S.Kevin Zhou, Integral Images, 2016
[10] Yuanshen Zhao, Three Types of Haar-like features
[11] Kaiqi Cen, Study of Viola-Jones Real Time Face Detector
[12] Petia Georgieva, Deep Learning Lecture: Object Identification
[13] Akash Desarda, Understanding AdaBoost, 2019
[14] Marek Kraft, 2017 Formation of the histogram of oriented gradients descriptor
[15] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, Yu Qiao, 2016 Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks
[16] Shuo Yang , 2015 WIDER FACE: A Face Detection Benchmark
[17] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, 2016 You Only Look Once: Unified, Real-Time Object Detection
[18] OpenCV Team, Cascade Classifier
[19] OpenCV Team, Cascade Classifier Training
[20] Manik Galkissa, 2017 Training SVM classifier with HOG features
[21] Manik Galkissa, 2017 Training SVM classifier with HOG features
[22] Neeraj Dixit, 2017 object-detection-with-svm-and-opencv
[23] Jake VanderPlas, 2016 Python Data Science Handbook - Application: A Face Detection Pipeline
[24] Iván de Paz Centeno, MTCNN 0.1.0
[25] Iván de Paz Centeno, 2020 MTCNN Face Detection GitHub Repository
[26] SKLearn Team, SVC
[27] Chi-Feng Wang, 2018 How Does A Face Detection Program Work? (Using Neural Networks)
[28] Arnaud Rougetet, 2019 Datasets of pictures of natural landscapes

Addendum

A. Division of labor

For this work we met online via tools such as Jitsy and Discord. Both students collaborated an equal amount of work hours developing all algorithms and analyzing every result in tandem.
Diogo Silva - 50%
Pedro Oliveira - 50%