Face Detection

1. Introduction

1.1 Context

The aim of this project is to design an application which is used to detect faces, as well as certain features of one's face such as the eyes, nose and mouth in pictures and live camera feed. The application will provide a GUI for the user to choose the type of media on which they would like to perform the face recognition operation and the result will be saved and displayed on the screen.

1.2 Specifications

The application will be implemented using the **OpenCV** library for Python, as well as small bits from packages such as **numpy** and a **TKinter** for building the menu GUI. The OpenCV library provides tools for manipulating the image data, using pre-trained machine learning models for processing real time facial recognition, image manipulation tools and the possibility of handling keyboard and mouse inputs. The **os** library is used to call the terminal commands of running the programs after pressing a button in the GUI.

1.3 Objectives

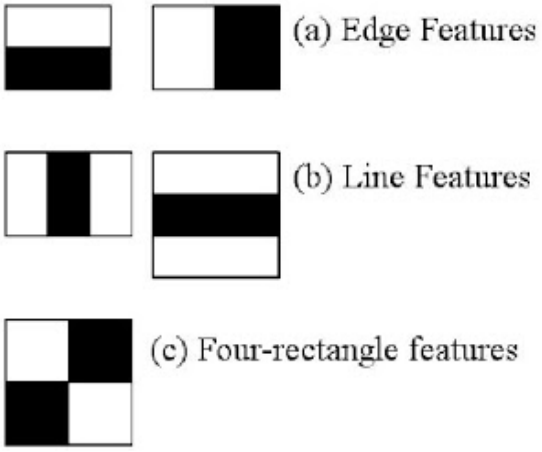
We need to design and implement a method that takes an image file, video and live camera feed and detects the faces as well as the facial features present in that piece of media. We get that by breaking down the image into small windows of 24x24 pixels and we apply a cascade of classifiers on these windows in order to determine if a face is present. The detected face and its facial features

will be highlighted with a rectangular border. The process of cascading and image processing will be detailed below.

2. Bibliographic study

The face recognition algorithm is based on a 2001 research paper by Paul Viola and Michael Jones. For this project we will use the already trained classifier, since it requires a huge data set of images with faces in them, as well as images that do not contain faces. However, it is paramount that we describe the functioning of this image classifier.

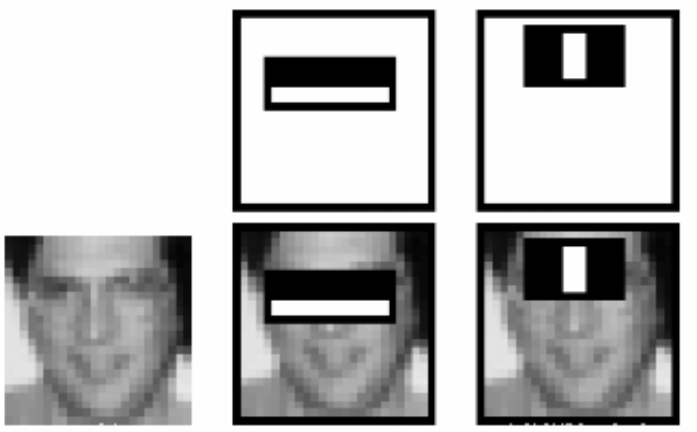
After we provide the images to the algorithm, we need to extract features from it. Every single image is grey scaled and we used the following type of classifiers for parts of the image, each feature being represented by the sum of pixels under the white rectangle and the sum of pixels under the black one:



This is a good start, but now we are need to apply these multitude of features (over 160000) to different groups of pixels. The algorithm splits the image in 24x24 pixels squares, but just considering an 1920x1080 picture, applying 160000 features to each 24x24 pixels square would require an incredible amount of computations. Doing that for every image in the data set is extremely costly. What we do to lower the number of matching features required is to classify the most important features. After we apply all of the features on the training data set, we look for features with the lowest face detection error rate. If we take the best 200 features that match faces, we get an accuracy of 95%. If we take the best 6000, it gets very close to 100%.

We now have reduced the number of required matching features to 6000 from 160000. Testing for 6000 features for every 24x24 square is still a big deal computation-wise. We may not be able to reliably detect faces on live video feed this way. To avoid that we cascade the features that we apply. This means that we apply the features in batches. We apply the first feature, if that does not match, we move to the next 24x24 square. In the second stage we apply 10 features. If one of those does not match we drop the square. There are 38 stages in total for applying the whole 6000 features, with an increasing amount of features applied during each successive stage.

In the picture below we have an example of common facial features that are present in all the faces: the first one means that the area of the eyes is darker than the area of the nose and cheeks and the area of the bridge of the nose is always lighter that the eyes. These are just 2 of the 6000 features we apply to each 24x24 window.



3. Analysis and Design

For this project we are to detect the the faces and the facial features of each face. To do this, we need a Haar Cascade Classifier (or multiple) for each facial feature. As stated earlier in the documentation, we will use pre-trained classifiers that come with OpenCV, since training them on our own would require a huge dataset consisting of millions of negative and positive images. The classifiers are trained for a specific minimum size for a certain object. For instance, the classifier for the nose is trained for noses that are at least 25x15 pixels in size. So for lower resolution images that contain multiple faces we might not detect any nose, since the each nose has a resolution lower than 25x15 pixels. The same goes for the face, eyes and mouth.

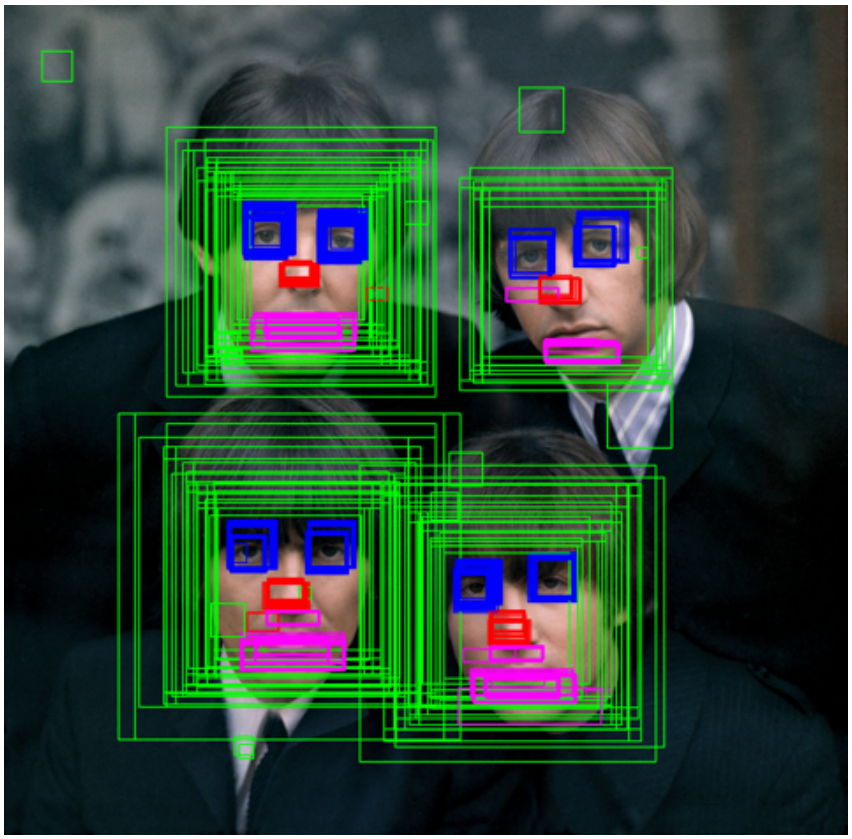
Now that we know the minimum requirements for the normal functioning of the classifiers, we need to use them for detection. We start off by using the most important one, the one for the whole face. To detect the face the algorithm scales down our image multiple times, going with classifier windows through the image at each scaling step (we will discuss scaling later in this chapter), trying to find matches for the classifier. After we've passed the image through the face classifier, we are left with a number of face windows on which to work and try to find facial features.We could search for eyes through the whole picture, but that would result in needless computations and my yield unwanted results.

To summarise, up to this point we have detected all the faces in the picture, each face being characterised by a window with a certain height and width in pixels. For each face window we can repeat the same process for the eye, mouth and nose classifiers. This should work well after adjusting the detection parameters (we will talk about those in a second), but before adjusting them we might run into some problems, such as the fact that the eyes have similar cascade features as the mouth, so we might detect the mouth as eyes and vice versa. Also, in some high resolution pictures, the part eyebrows might be detected as a nose. To avoid those problems, we try to split the face window as such: we search for the eyes in the first half of the face, for the nose from 1/4th to 3/4ths of the face, and for the mouth in the second half of the face window (these proportions might change in the actual implementation).

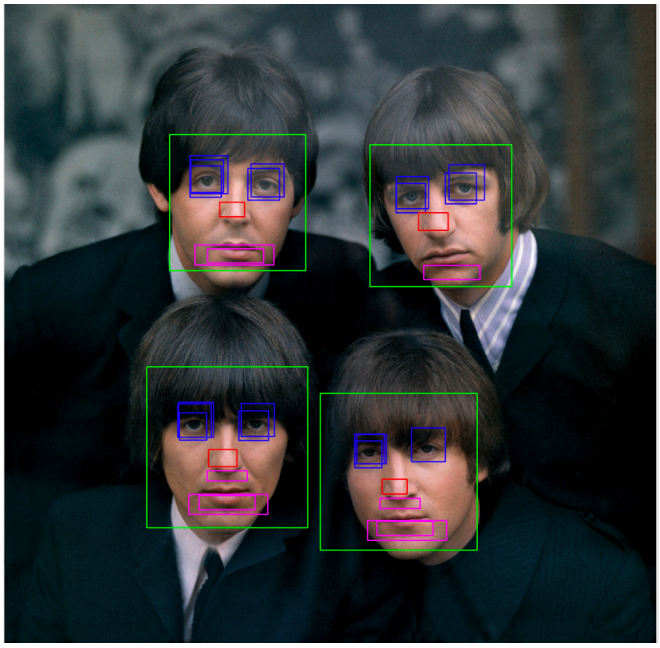
Now that we described way the classifier will be applied, we will discuss the detection parameters mentioned above. For this project we will modify the values (as it suits us) of two important detection parameters: the **scale factor** and the **minimum neighbours** value.

The **scale factor** determines the scaling of the image at each step of the process. For example, we have a detection window for faces that is 24x24 pixels and we look for faces in a 3840x2160 picture. There is little chance that there is any 24x24 face in the picture at that resolution. So at each step we scale down the image and look for a face again. The default scaling factor is 1.1 in our project, meaning that at every step, the scale of the image is reduced by 10%. The algorithm does so until the image is smaller than the window of the classifier. A higher scaling factor means a that the detection will be faster, but might miss some faces since they might have been detected at a stage that we now skip.

The **minimum neighbours** parameter refers to the number of neighbours each candidate rectangle has to have in order to be taken into account for the detection. Now we are concerned with the concept of a neighbouring rectangle in the detection. When we are scaling the image looking, for instance, for a face, at each scaling step we might detect the same face multiple times, that face being enclosed in a rectangular window. This means we detect the same face at multiple scaling levels, so that means that we will have multiple almost overlapping windows detecting the same face. These almost overlapping windows are the “neighbours” of a particular window.



Take the picture above as an example. The green rectangles represent faces. We can see that we detect objects that are clearly not faces, but at a certain scaling these objects were detected as faces. The **minimum neighbours** parameter is set to 0 in this case, so we display these detections, even though they are clearly not faces.



In this second picture we set the parameter to 1 for face detection. Just by this slight modification, we can see that we can accurately detect the faces present. The parameters are still not perfectly tweaked for the mouth, but this was just an example used to describe the **minimum neighbours** detection parameter.

In short, this is the list of steps to be taken:

1. Selecting a Haar Cascade Classifier for every feature that we want to detect.
2. Finding the faces in the picture, by applying the face classifier to the whole picture.
3. Finding the eyes, nose and mouth on each face window by applying their respective classifier.
4. Tweaking the detection parameters so we have a more accurate detection.

4. Implementation

The first thing we do in the implementation of the project is to load the classifiers, which come preinstalled with the OpenCV library (except for the nose):

```
face_cascade_path = 'haarcascades/haarcascade_frontalface_alt2.xml'
eye_cascade_path = 'haarcascades/haarcascade_eye.xml'
right_eye_cascade_path = 'haarcascades/haarcascade_righteye_2splits.xml'
left_eye_cascade_path = 'haarcascades/haarcascade_lefteye_2splits.xml'
nose_cascade_path = 'haarcascades/haarcascade_nose_2.xml'
mouth_cascade_path = 'haarcascades/haarcascade_smile.xml'
```

After loading the classifiers we create trackbars in the OpenCV UI, so that we can change the values of the **minimum neighbours** and **scale factor** parameters while we are analysing an image. There will be a separate trackbar for each parameter of each feature (eyes, mouth, nose and face).

```
cv2.createTrackbar(trackbar_face, window_name, 11, 30, nothing)
cv2.createTrackbar(trackbar_face_neighbours, window_name, 5, 30, nothing)
```

This is an example for trackbar creation for the detection of the face. In the first line we create a trackbar for the scale factor and in the second one we create one for the minimum neighbours. The values go from 0 to 30 and the default value is 11 for the scale factor (which will translate in a scale factor of 1.1, but we can't have float numbers on the trackbar, so we change the order of magnitude) and a default of 5 minimum neighbours. Even though the scale factor slider can go down to 0, the scale factor will never go below 1.1 (so any value below 11 on the slider does not change anything).

Inside the main program loop we check the input mode (camera or image). If it is an image (mode 0), we call a method to detect the faces and from that method we call another one to detect the facial features on each face. If we are in camera mode (mode 1) we break down the video feed into each frame and apply the same methods on every frame recorded by our camera. This process repeats whenever we press a key (except for the Esc key which closes the program), so when we modify the trackbar parameters, we compute everything again with the new value. This means that for continuous video detection we will have to hold down a key.

For the detection of any feature (we will again use the face as an example) we apply the following method:

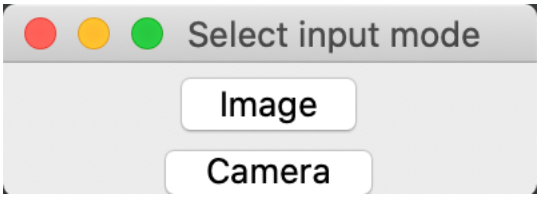
```
faces = face_cascade.detectMultiScale(
    gray,
    scaleFactor=FACE_SCALE_FACTOR,
    minNeighbors=FACE_NEIGHBOURS,
    flags=cv2.CASCADE_SCALE_IMAGE
)
```

The first argument is the gray scaled image, while the next two arguments are inputs taken from the trackbar. This returns a tuple with the coordinates of the upper left corner of the beginning of the face and the width and height of the rectangle that surrounds the face. We draw the rectangle around the face using the below function:

```
cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

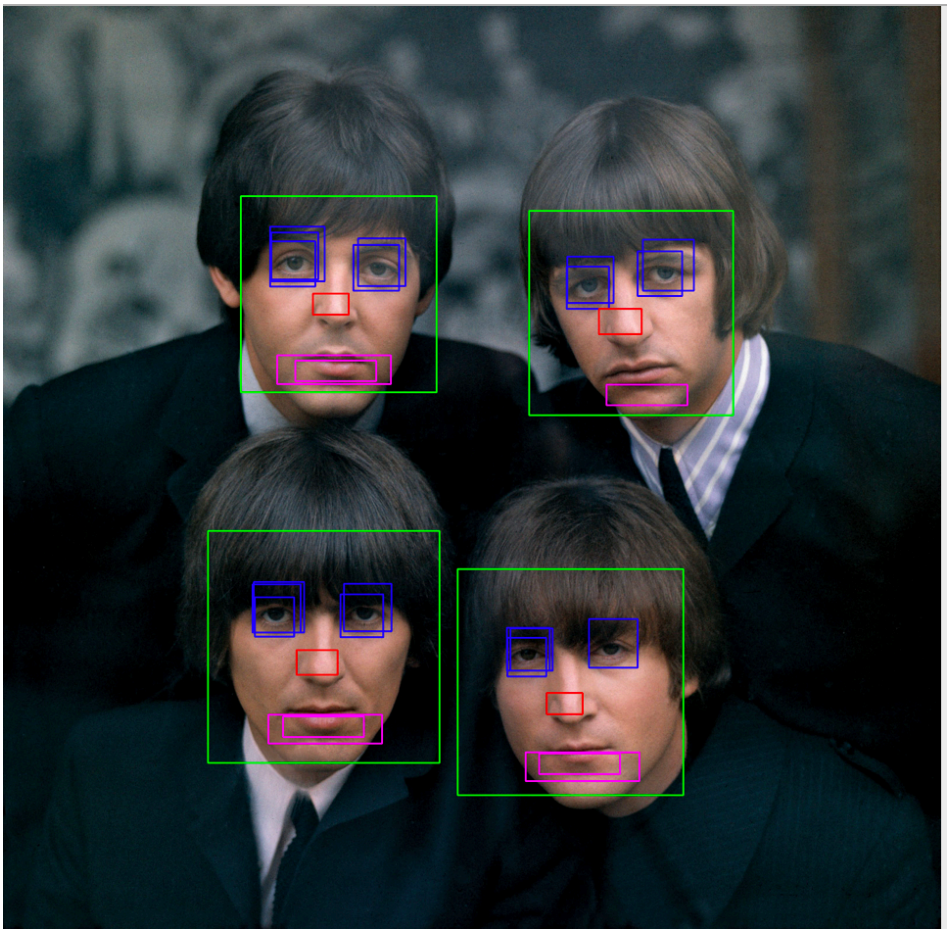
The first argument is the image on which we draw, the second argument is the upper left corner of the computed face rectangle, while the third argument is the computer lower right corner. The fourth argument determines the colour of the rectangle (in BGR space, in this case the colour will be green). The fifth parameter is just the thickness of the line. We draw a rectangle around each face and for each face we draw rectangles around the eyes, mouth and nose.

Finally, we use a basic GUI form to select the input mode. If we select the Image we will be prompted with a browse window to select an image and if we choose the Camera mode the OpenCV window will open with the live camera feed.

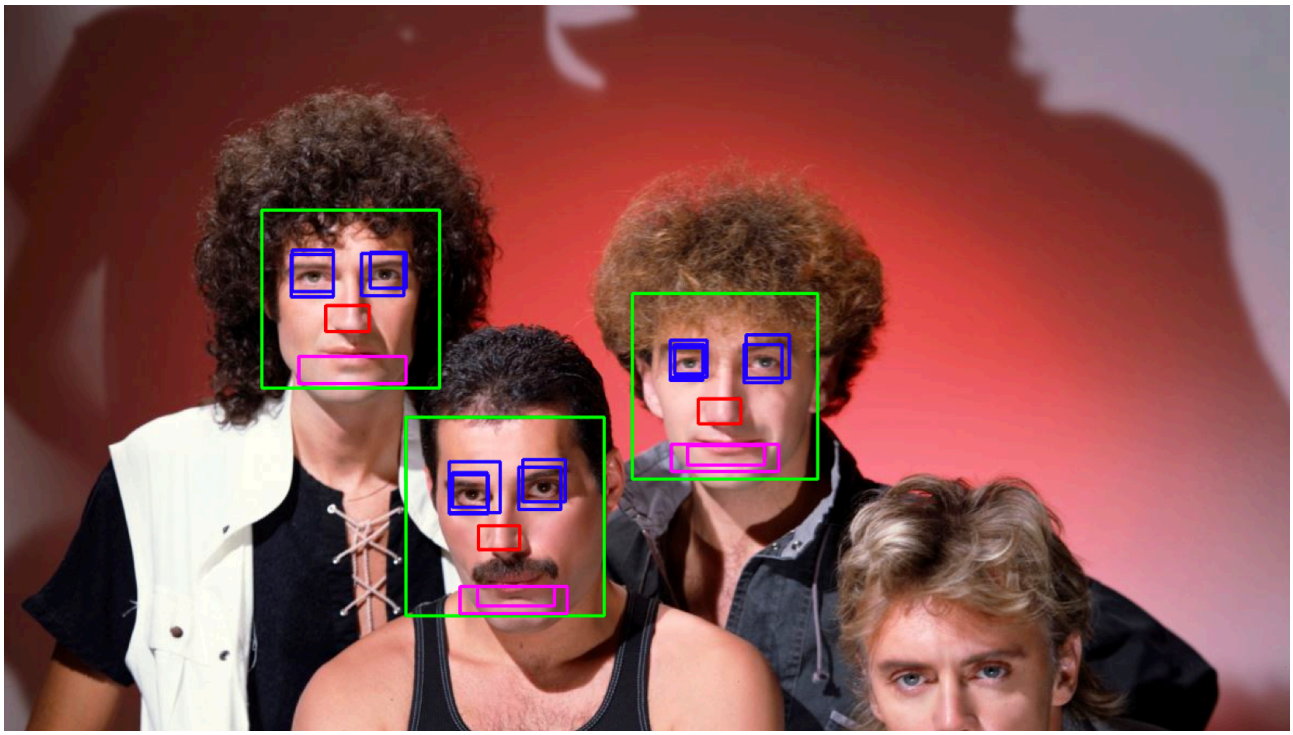


5. Testing and Validation

We can observe that we can detect all the faces and their features in the image below. The reason there are more rectangles around the eyes is that we actually use three classifier for the eyes. One for the general set of eyes, one specialised for the left eye and one specialised for the right eye. It is difficult to detect eyes in certain positions, so we needed to use multiple different classifiers to cover when one was not working properly in a setting. We also have multiple rectangles for the mouth in three of the four cases, but if we make the detection more strict for the mouth we will end up with mouths that are undetected.

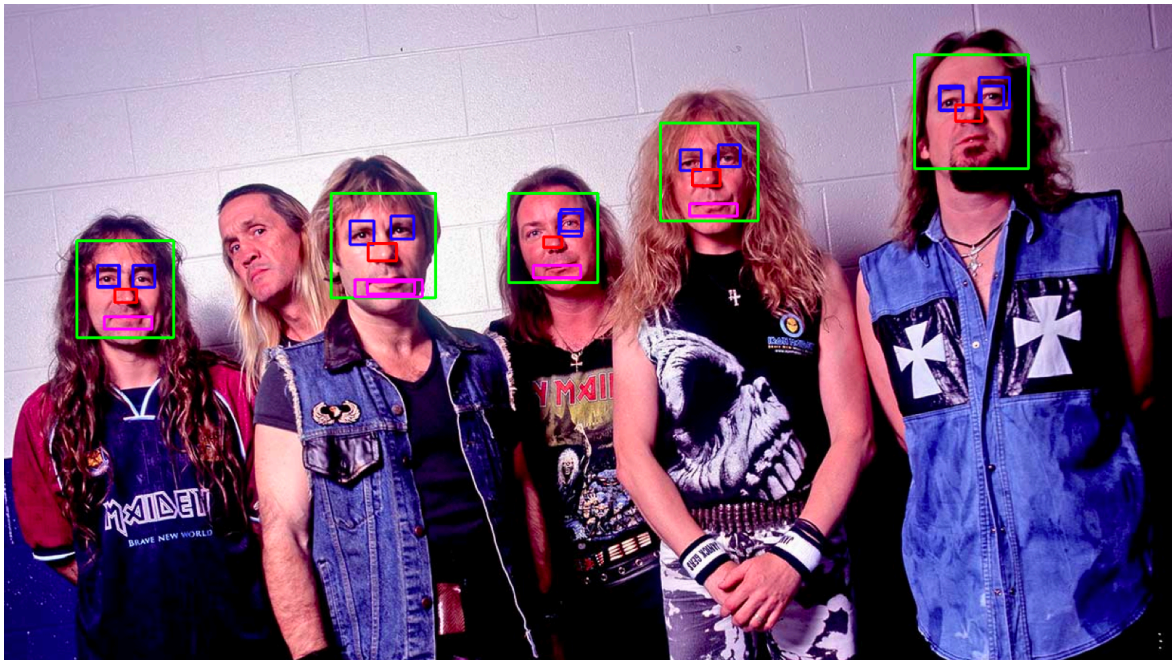


In the picture below we can see one of the shortcomings of the algorithm. We can not detect the fourth face to the right, since we can't see it in its entirety, so the classifier won't detect it.

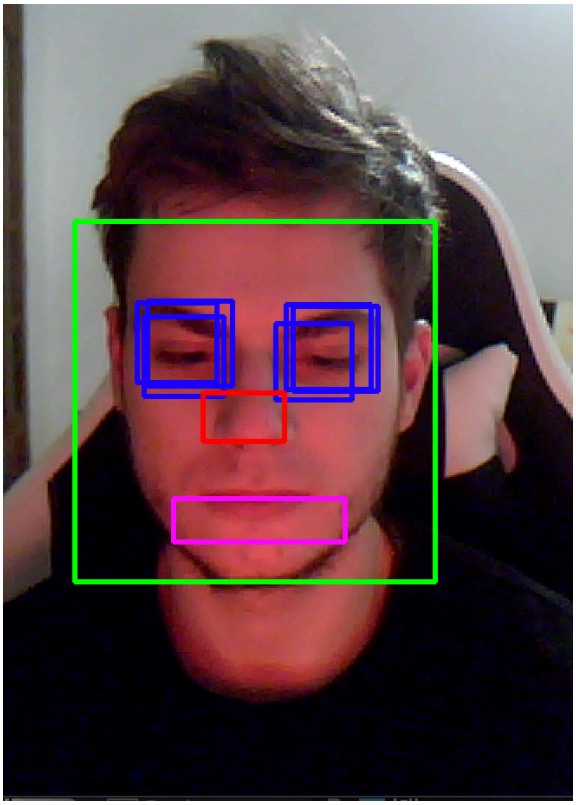


As for the image below, we can detect all the faces and the mouth of every face. But we are having trouble detecting all of the noses, as well as almost any of the eyes. This is caused by the fact that the undetected noses and eyes have a size smaller than the size of the trained classifier window, which means there is no way to detect those facial features.

Another possible errors could occur when the head is not facing the camera entirely in some images, or the head is tilted. This will result in our program not detecting the face in its entirety. Also, as it can be seen in the righmost face, we can not detect the mouth, most likely because of the facial hair.



As it can be seen below, the detection works well on live camera feed. If the camera would have a lower resolution, or the face would be further from the camera, we might run into detection issues, just like in the group photo above.



6. Conclusions

We managed to implement a project that can fairly accurate detect faces and the main facial features of each face with a fairly good degree of accuracy. It is not entirely perfect since it may have certain shortcomings as shown in the testing section.

Due to the existence of advanced image processing libraries such as OpenCV, as well as the availability of well trained classifiers, the project was not very difficult to implement in code. The interesting part was understanding how the algorithm operates in order to achieve a high degree of accuracy with the available resources. The Viola-Jones algorithm works extraordinarily well even when compared to the newer algorithms that use Convolutional Neural Networks to detect faces and thus still remains the algorithm used in modern cameras for facial detection.

7. Bibliography

1. http://alereimondo.no-ip.org/OpenCV/34
2. https://docs.opencv.org/master/db/d28/tutorial_cascade_classifier.html
3. https://docs.opencv.org/master/dc/da5/tutorial_py_drawing_functions.html
4. https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection#TOC-Image-Pyramid
5. Rapid object detection using a boosted cascade of simple features (2001) - Paul Viola, Michael Jones