L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Computer Information Systems

Fast Frontal Face and Eye Detection Using the Viola-Jones Object Detection Method

Cristina Cutajar* (230802L)

*B.Sc. (Hons) Artificial Intelligence

Study-unit: **Machine Learning: Introduction to Classification, Search and Optimisation**

Code: **ICS2207**

Lecturer: **Mr Kristian Guillaumier**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Cristina Cutajar	
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

ICS2207	ICS2207 – Machine Learning Project
Course Code	Title of work submitted

12/01/2022

Date

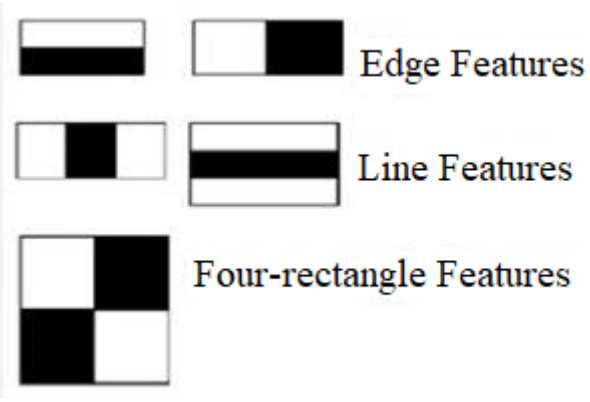
How Viola-Jones Object Detection/Haar Cascades Work

Viola-Jones is an object recognition framework algorithm which is mainly used for facial recognition however it can also be trained to detect different object classes. Despite being developed in 2001 by Paul Viola and Michael Jones, its application is still very powerful and exceptionally notable at real-time face detection [1].

The downside to this algorithm is that it is very time consuming to train it, for it to be very accurate. However, once it is trained, it is capable of detecting real-time faces with impressive speed [2]. Moreover, the Viola-Jones algorithm is more accurate at detecting frontal faces rather than faces looking downwards, upwards or to the side.

In order for the Viola-Jones algorithm to work, the image must first be converted to grayscale. The algorithm will split an image into different small subregions and it will analyse each subregion to try to find specific features based on its training data. Since an image can contain multiple faces of different sizes, the algorithm checks many different positions with different scales.

The Viola-Jones algorithm is capable of interpreting different parts of a face by making use of the 3 types of Haar-like features. The first Haar-like feature would be Edge features which are useful at detecting edges. The second Haar-like feature is called Line features and these are used at detecting lines. The third and last feature would be the Four-sided features which are used to detect diagonal features.

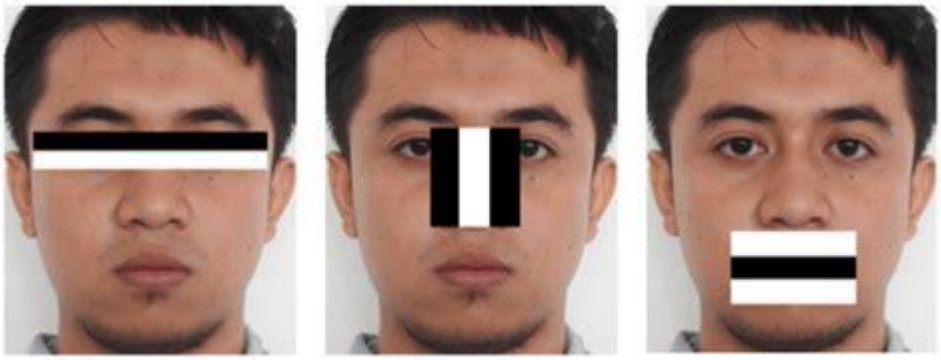


[2]

Figure 1: Haar-like Feautures

These digital image features work by summing up the pixel values of specific regions and comparing them together. The results are then displayed using black and white rectangles to differentiate lighter and darker regions. Hence, an image of a plain surface would provide no useful information as all the pixels would have the same value. Due to universal properties of the human face, these digital features can be applied to identify different parts of a face. Since human faces contain many dark and bright spots, large numbers are returned by the Haar-like features and this in turn makes them very accurate. These features are then combined to check if an image region contains a human face. For example, the nose region is

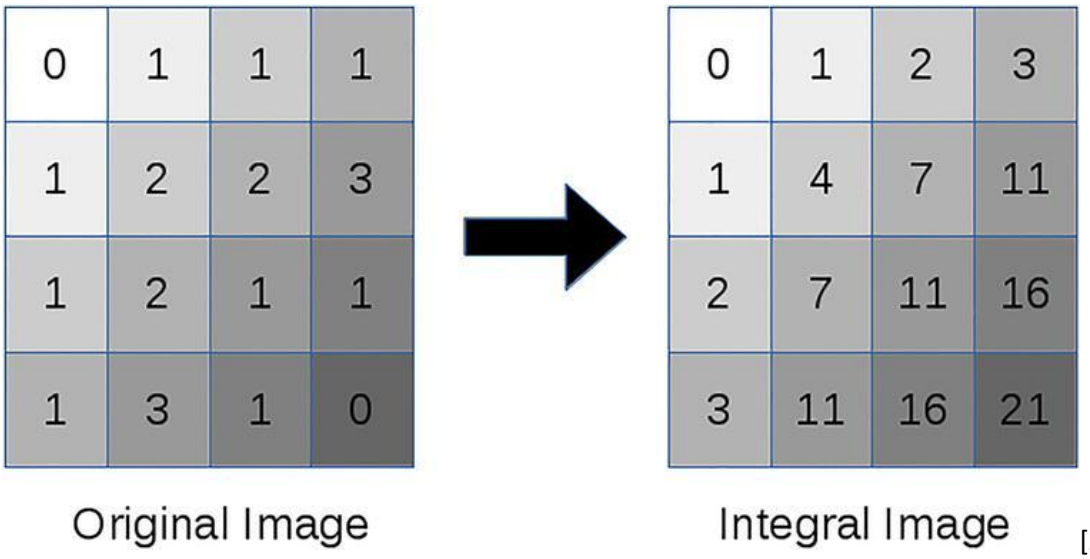
usually brighter than the eye region as well as the middle part of the nose region being shinier than the neighbouring regions. These can be seen below:



[3]

Figure 2: Haar-like Features on Face

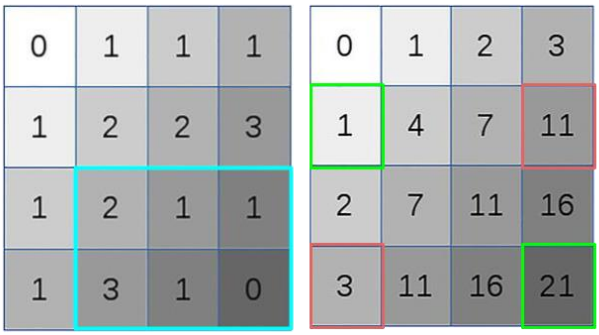
The Viola-Jones algorithm also makes use of Integral Images. An Integral Image is a data structure and algorithm that is used to perform the calculations for the Haar-like features in a quick and efficient manner. As can be seen below, each point of an Integral Image is the sum of the above, left and target points.



[2]

Figure 3: Integral Images

Therefore, a lot of time is saved when using Integral Images as calculations have to be performed on 4 points of the rectangle instead of having to sum all of them individually. From Figure 4 below, the green vertices need to be added whilst subtracting the red vertices in order to find the sum of a particular area, which in the example below is the area marked with a blue border.



[2]

Figure 4: Integral Images

Before the machine is able to identify the required objects, it needs to be trained. The user needs to supply the Viola-Jones algorithm with a lot of positive and negative images in order for the algorithm to be trained. Positive images are the ones that contain the required object to be detected. These images need to be labelled for the algorithm to know which object it needs to analyse to be able to detect. Meanwhile, negative images are those that do not contain the object to be detected. This is done so that the algorithm can further on differentiate the features of what needs to be identified and what doesn't. Viola-Jones algorithm will shrink the images to a size of 24 by 24 pixels to be able to look for the trained features within the image.

Since Viola-Jones makes use of a 24 by 24 pixels detector window, there are nearly 160,000 features present in the detector window. AdaBoost is a Machine Learning Algorithm which is able to identify the important features from all the features found in the detector window. AdaBoost takes in training data to identify the important features which will later on be used to create the final classifier. It is able to learn from the supplied images which in return makes it more accurate as it will be able to determine false positives and true negatives. AdaBoost evaluates the performance of the given classifiers to decide the size and type of each Haar-like feature that goes into the final classifier. It does so by evaluating each classifier on all subregions of the images used for training. If a subregion returns a strong response, it will be classified as a positive as the classifier would think that the object to be detected is present in that subregion. Meanwhile, if a subregion doesn't return a strong response, then the classifier would assume that the object to be detected is not found in that subregion and it will be classified as a negative. Then, the final classifier containing the best performing weak classifiers is returned as the strong classifier.

Viola-Jones uses cascading classifiers to discard non-faces from the input image in the detector window in a quick manner as otherwise, it would take a long time for AdaBoost to calculate the best features for each region. This way real-time face detection is achieved in a fast and efficient way which also avoids performing unnecessary computations. For any given image, the cascade classifier first passes through the best and most prominent features. If a positive is found in the subregion, then the classifier will pass through the remaining features. If the subregion is confirmed to be the object which needs to be detected, the user is notified about the detection. If the classifier returns a negative evaluation when it passes through the best features, then the image is discarded so as to avoid unnecessary computations. This also

applies to the remaining classifiers. If any of the classifiers return a negative evaluation, then the image is assumed to not have the required object and it is discarded.

As mentioned above, the Viola-Jones Object Detection algorithm is very useful for real-time applications as it is able to detect the required objects in a very short time. Not only is it able to detect the objects in a short time but it is also very accurate and has the ability to generalise and learn from its findings. This makes it very practical to use especially in real-time applications such as facial and eye tracking from a live webcam feed.

Artefact 1

Implementation

In order to produce this artefact, I first downloaded OpenCV version 3.4.14 [4] and Python version 3.7.0 [5]. I then installed OpenCV on python and created folders in the main folder to later on store the necessary data including the images to train the classifier, the trained classifier, python modules and validation images. Then I copied the executable files from the OpenCV folder to the main folder for Artefact1.

I obtained a part of a dataset containing images of faces [6] and put these images in the 'pos' folder. These will be used for the classifier to know what objects it needs to learn to recognise. The classifier also needs images that do not contain the object to be detected to be able to differentiate what needs to be recognised and what does not. Hence, I also downloaded images that do not contain faces [7] and put these images in the 'neg' folder.

For the cascade classifier to be able to be trained to detect faces, the positions of the faces found in the positive images need to be given. More specifically, it needs the x and y axis of where the face starts from the top left corner as well as the width and height of the face in pixels. To do this I used a website where I had to draw a rectangle around each face, and I received the x and y axis of both corners [8]. I then had to decrease the values to find the width and height of the faces. I later found out that OpenCV has an executable file that aids in doing this and does this in a much quicker manner. Therefore, I created an info.dat file with the location of each positive image, along with its corresponding x and y-axis and height and width of each face in pixels. I also created a bg.txt file with the path to every negative image.

The data to be used for training was ready and so, I executed the **opencv_createsamples** file from the command line and passed in the info.dat file, the number of positive images, the width and height of each image as well as the name of the .vec file which will contain the output of the opencv_createsamples. I set the width and height of the images as 24 pixels since Viola Jones makes use of a 24 by 24 detector window. The opencv_createsamples labels and transforms the given positive images to the required format to be used for training. For the training, I gave 100 positive images as well as 230 negative images. During experimentation, I increased the number of negative images to 300.

Once the images were correctly labelled and transformed, I executed the **opendcv_traincascade** file. In order for this to properly execute, I passed in the folder for the .xml files to be generated, the .vec file of the samples, the bg.txt, the number of positive images, the number of negative images, the width and height of the images (24pixels) and the number of stages to train to cascade.

The more stages, the more the cascade will be trained, however it is important to not overtrain the cascade as then it will not be able to generalise the objects and it will only detect objects that look like the given positive images.

Experiments and Evaluation

In order to experiment, I modified the number of stages, the number of negative images as well as the maxFalseAlarmRate and minHitRate. I also used the code from Artefact2, by passing in the trained cascade, to check the cascade for each modification I performed during its training. Below find the outputs of differently trained classifiers:

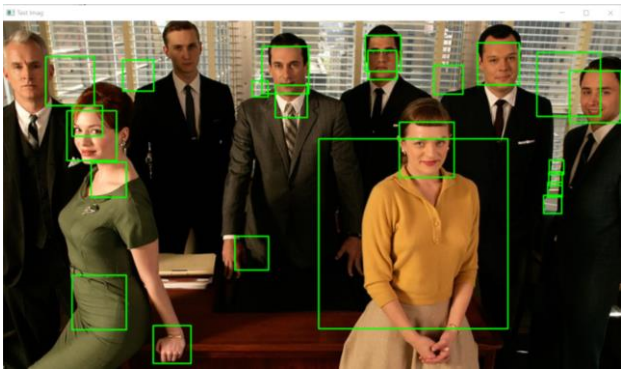


Figure 5: Trained classifier with 100 positive images, 230 negative images and 13 stages of training.

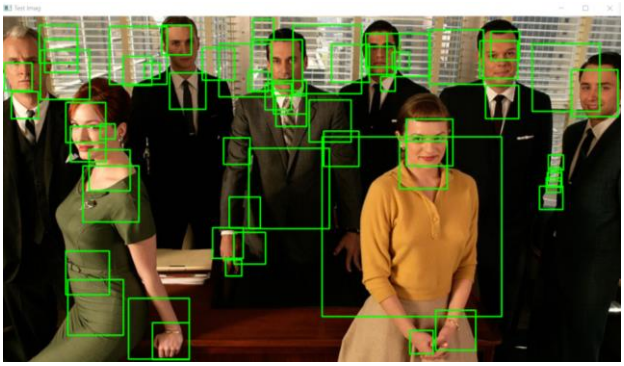


Figure 6: Trained classifier with 100 positive images, 230 negative images and 10 stages of training.

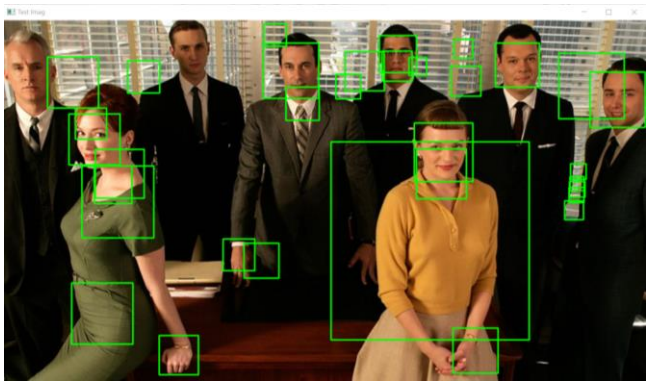


Figure 7: Trained classifier with 100 positive images, 230 negative images and 12 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12.

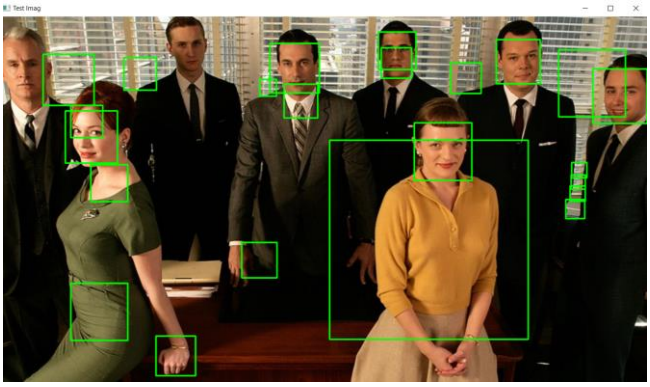


Figure 8: Trained classifier with 100 positive images, 230 negative images and 13 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12 and 13.

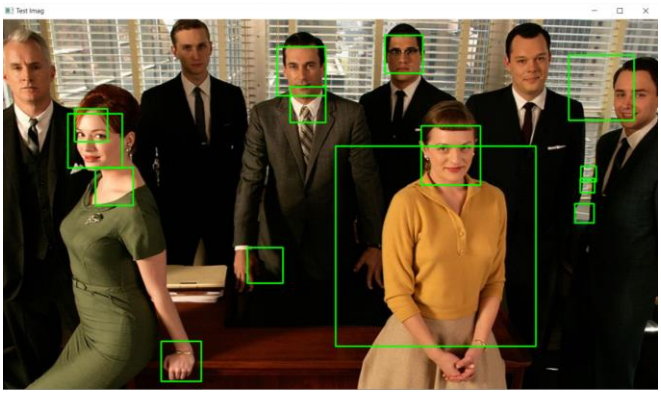


Figure 9: Trained classifier with 100 positive images, 230 negative images and 14 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12, 13 and 14.

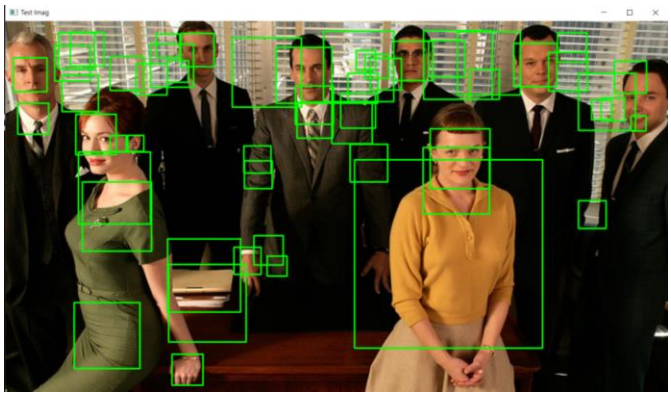


Figure 10: Trained classifier with 100 positive images, 300 negative images and 10 stages of training.

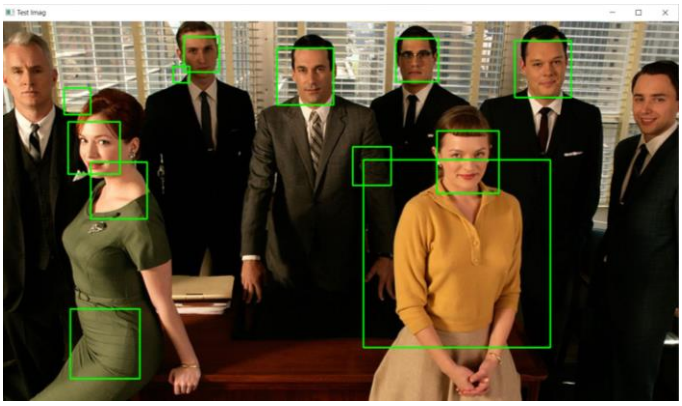


Figure 11: Trained classifier with 100 positive images, 300 negative images and 12 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 11 and 12.

Figure 12: Trained classifier with 100 positive images, 300 negative images and 13 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 11, 12, 13.

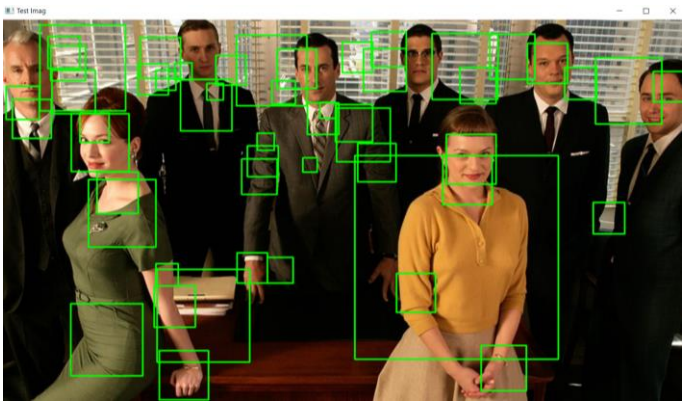


Figure 13: Trained classifier with 100 positive images, 300 negative images and 7 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 5 and 7.

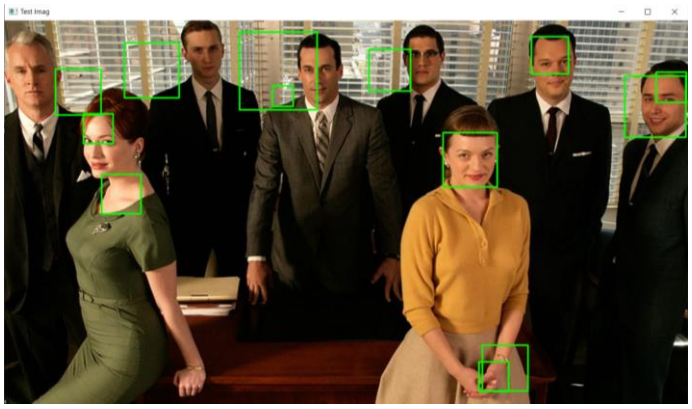


Figure 14: Trained classifier with 100 positive images, 300 negative images and 9 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 5, 7 and 9.

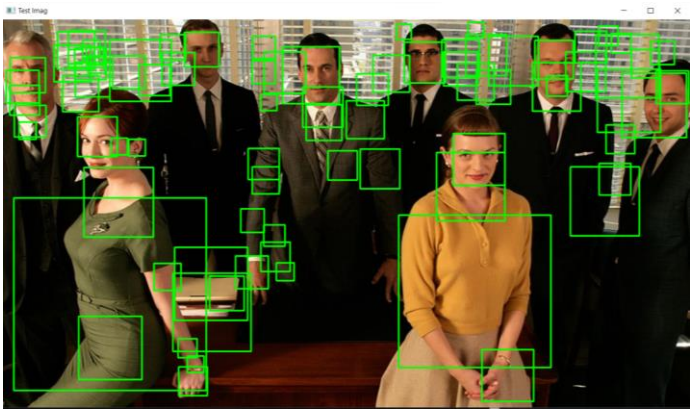


Figure 15: Trained classifier with 100 positive images, 300 negative images and 9 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 9.

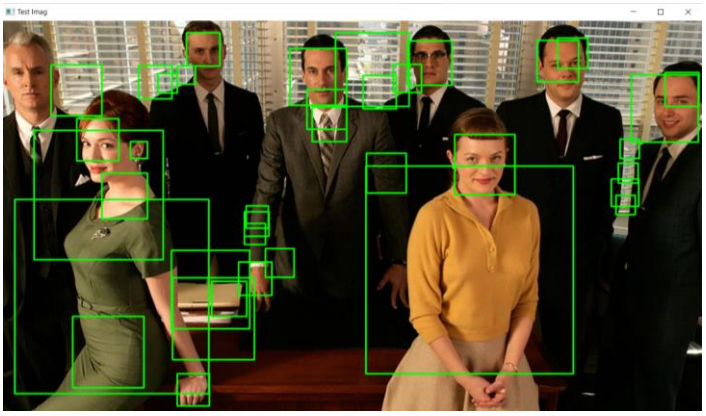


Figure 16: Trained classifier with 100 positive images, 300 negative images and 12 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 9, 10, 11 and 12.

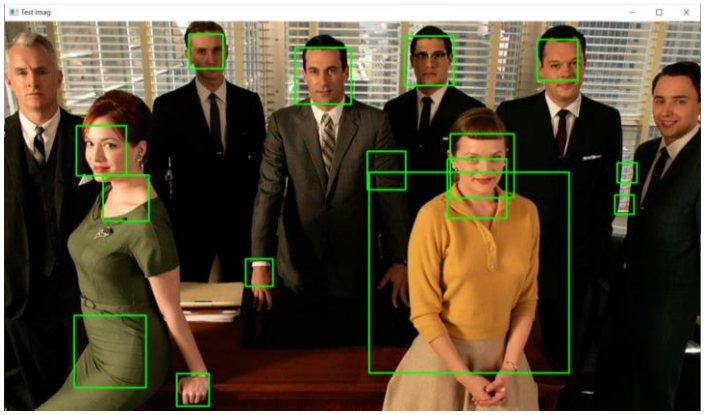


Figure 17: Trained classifier with 100 positive images, 300 negative images and 13 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 9, 10, 11, 12 and 13.

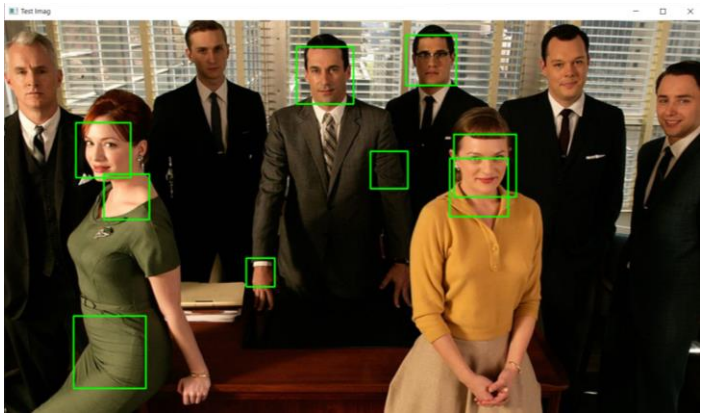


Figure 18: Trained classifier with 100 positive images, 300 negative images and 14 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 9, 10, 11, 12, 13 and 14.

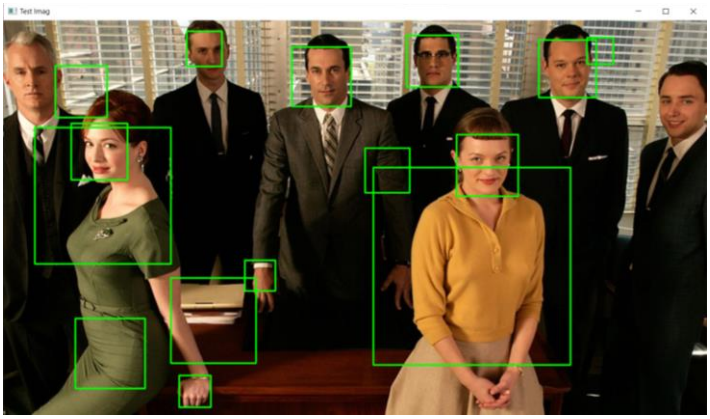


Figure 19: Trained classifier with 100 positive images, 300 negative images and 13 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12 and 13.

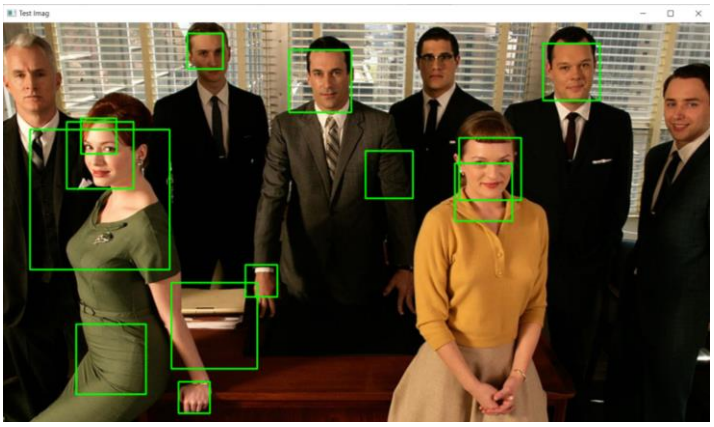


Figure 20: Trained classifier with 100 positive images, 300 negative images and 14 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12 and 13.

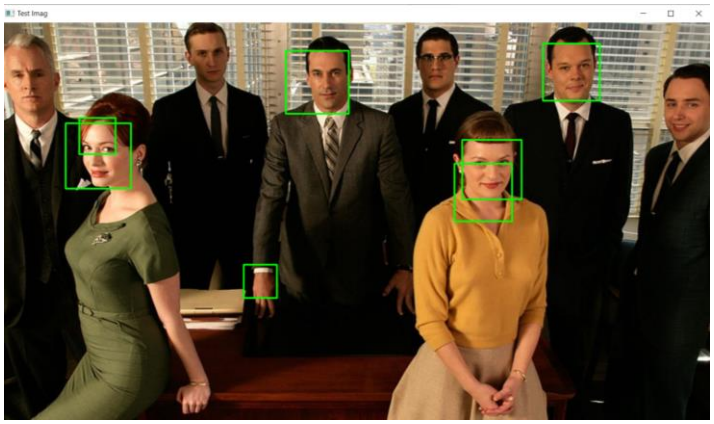


Figure 21: Trained classifier with 100 positive images, 300 negative images and 15 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12, 13 and 15.

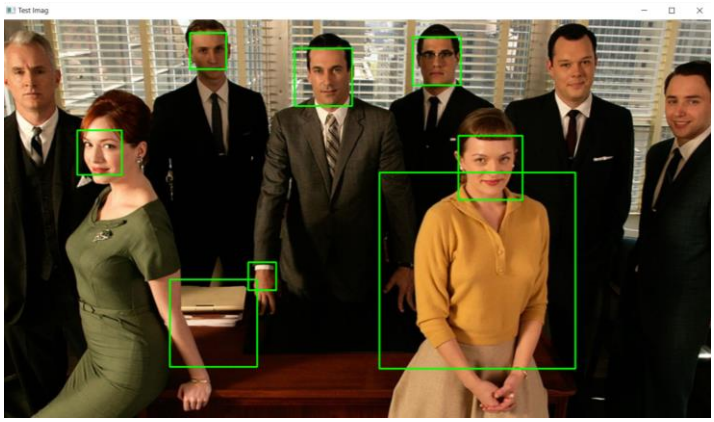


Figure 22: Trained classifier with 100 positive images, 300 negative images and 14 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12, 13 and 14.

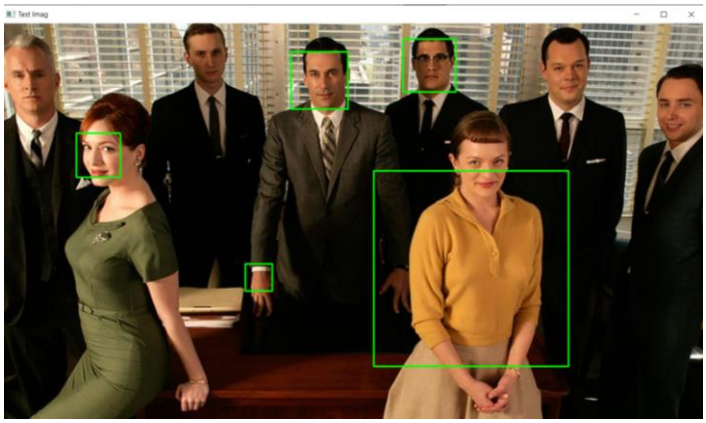


Figure 23: Trained classifier with 100 positive images, 300 negative images and 15 stages of training with maxFalseAlarmRate 0.3 and minHitRate 0.999 at stage 12, 13, 14 and 15.

From the above figures and from much more experimentation, I made some important observations. By comparing Figure 6 with Figure 10, one can see the difference that the increase in negative images for training does. Due to the increase of negative images, in Figure 10, there were less false positives generated than in Figure 6 despite all other values being the same. The more negative images, the better the cascade is able to distinguish between the characteristics of the objects that need to be detected and does that do not.

Overtraining can occur where the cascade is no longer able to generalise different images and objects to be detected and this results in it only detecting objects that are found in its training data. This can be seen mainly in Figures 12 where none of the faces where detected but it is also present in Figure 23 where most of the faces are also not detected.

I also noticed that the maxFalseAlarmRate and the minHitRate are only applied to the last training stage. This can be seen between Figure 14 and Figure 15. In Figure 14, the cascade is way more trained than that of Figure 15 despite them both being at the same training stage. This is because the cascade of Figure 14, had the maxFalseAlarmRate and minHitRate for stages 5, 7 and 9 whilst the cascade of Figure 15 only had these implemented for stage 9.

Figures 13 and 14, as well as Figures 16 and 17 show the difference that can happen with only two or one stages respectively, especially if the maxFalseAlarmRate and the minHitRate are included in the stage. This can especially be seen in Figures 12 and 19 where the cascade of Figure 12 resulted in overtraining whilst that of Figure 19 did not. The difference between these two figures was that in Figure 12, the 11^{th} stage had the maxFalseAlarmRate and the minHitRate included whilst in Figure 19, these were not included for stage 11. This can also be seen in Figure 21 and 23 where in Figure 21, the maxFalseAlarmRate and the minHitRate were not included for the 14^{th} stage but they were included for the 14^{th} stage of Figure 23.

Therefore, from my experimentation, I concluded that the training used for the cascades of Figure 11 and Figure 22 were the best. From these two cascades, I would opt for the one of Figure 22 even though it detected one less face than that of Figure 11. This is due to the fact that the cascade in Figure 11 detected many more false positives from that of Figure 22.

Artefact 2

Implementation

For Artefact 2, in order to produce a program that is able to detect faces and eyes in images, I made use of the OpenCV library and trained cascades [10]. I downloaded three frontal face detection cascades and two eye detection cascades to be able to experiment with the different cascades and see which option is the best for my project. I also used photos from [11] to test the different cascades.

In my program, I first imported the cv2 library and with the use of its **cv2.CascadeClassifier()** function, I loaded the different trained cascades that I had downloaded. Then, I created a function named **detect_eyes_faces()** which takes the eye cascade, face cascade and coloured image as parameters. Since the OpenCV cascades perform object detection on grayscale images, a copy of the given coloured image is made and it is converted to grayscale, using the **cv2.cvtColor()** function. Then, the **detectMultiScale()** function is called with the eye and face cascades individually, to perform detection on the grayscale image. For eye detection, I set the scaleFactor to be 1.3 and minNeighbors to be 6. For facial detection, I set the scaleFactor to be 1.1 and minNeighbors to be 5. I found these settings to give the most accurate results for eye and face detection. Then, the function will loop through the coordinates for each eye and face detected. In this loop, using the generated coordinates from the grayscale image, a green rectangle will be set to outline each face and a red rectangle will be set to outline each eye on the original coloured image. Finally, the coloured image with the rectangles marking the faces and eyes, will be returned. Originally, in this function, I had also included print statements to print the number of faces and eyes detected in each image. However, I commented these out as for the live webcam detection, these print statements will keep being outputted until the user closes the camera. This is because for the live webcam detection, in order to receive live correct results, the function to detect faces and eyes is called multiple times until the webcam is closed.

In order to test this function, I imported some test images using the **cv2.imread()** function and called the **detect_eyes_faces()** function with the imported image and chosen eye and face cascades. Then after facial and eye detection was performed on the given image, I called the **cv2.imshow()** function to display the final image in a window with the name of the eye and face cascades used. With the use of the **cv2.waitKey()** and **cv2.destroyAllWindows()** functions, the window will remain open until the 0 button is pressed. When the 0 button is pressed, the window will be destroyed. I did the above multiple times with different images and calling different cascades in order to test the cascades and find the most optimal ones.

Then, in order to test the **detect_eyes_faces()** function with the live webcam feed, I made use of the **cv2.VideoCapture()** function with 0 passed in as a parameter in order to open the primary camera. If a camera is found and launched, then, with the use of the **read()** function, each frame is returned as well as a Boolean value to show if a frame was found or not. The **detect_eyes_faces()** function is then called on every frame of the live webcam feed and then,

the frame with the rectangles marking the eyes and faces is displayed to the user with the use of the **cv2.imshow()** function. The above will continue to loop until the ‘e’ button is pressed. With the use of the **cv2.waitKey()** function, when the ‘e’ button is pressed, the function will stop, the camera will be closed using the **release()** function and the window where the frames where being displayed will be destroyed using the **cv2.destroyAllWindows()** function.

Experiments and Evaluation

For the facial detection, I tried out three different cascades; the haarcascade_frontalface_alt.xml cascade, haarcascade_frontalface_default.xml cascade and haarcascade_frontalface_alt_tree.xml cascade.

In Figure 24, detection was performed using the haarcascade_frontalface_alt.xml cascade whilst in Figure 25, detection was performed using the haarcascade_frontalface_default.xml cascade and in Figure 26, detection was performed using the haarcascade_frontalface_alt_tree.xml cascade. As can be seen in Figure 24, the haarcascade_frontalface_alt.xml cascade correctly identified all the faces with no false positives. In Figure 25, the haarcascade_frontalface_defualt.xml cascade correctly identified all the faces, however it also detected one false positive. Meanwhile, in Figure 26, the haarcascade_frontalface_alt_tree.xml cascade failed to detect any of the three faces.

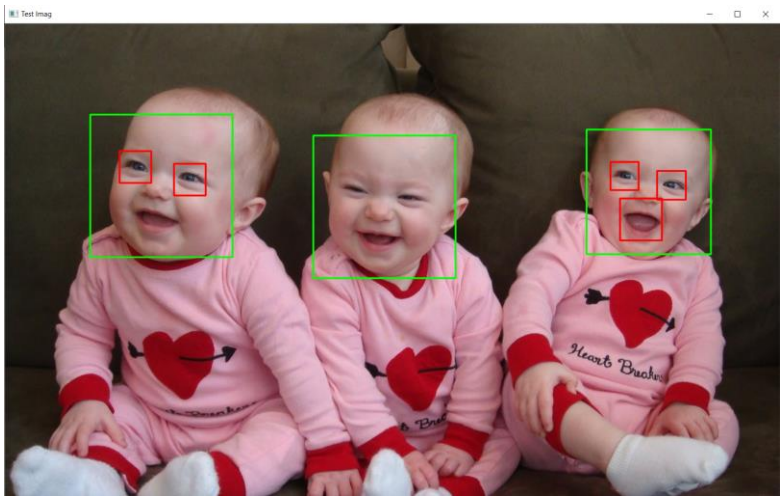


Figure 24: Facial detection using haarcascade_frontalface_alt.xml cascade

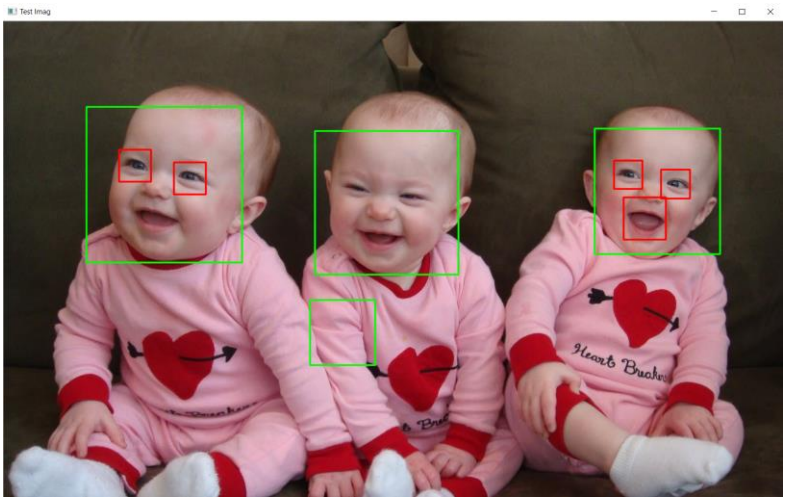


Figure 25: Facial detection using haarcascade_frontalface_default.xml cascade

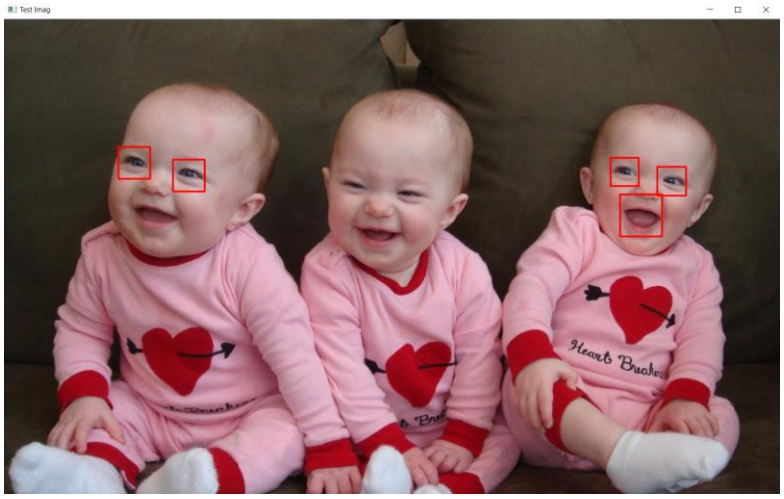


Figure 26: Facial detection using haarcascade_frontalface_alt_tree.xml cascade

Similar to the above results, both the haarcascade_frontalface_alt.xml cascade and the haarcascade_frontalface_default.xml cascade correctly identified all the faces with no false positives. These results can be seen in Figure 8 and Figure 9 respectively. In Figure 29, the haarcascade_frontalface_alt_tree.xml only detected four out of eight faces with no false positives.

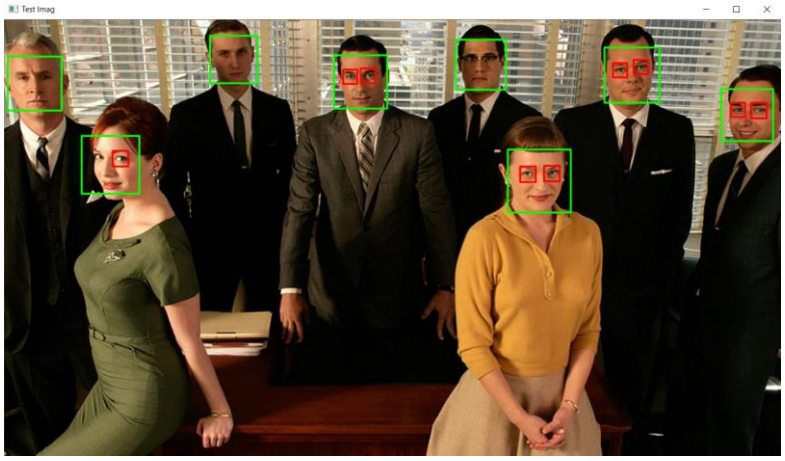


Figure 27: Facial detection using haarcascade_frontalface_alt.xml cascade

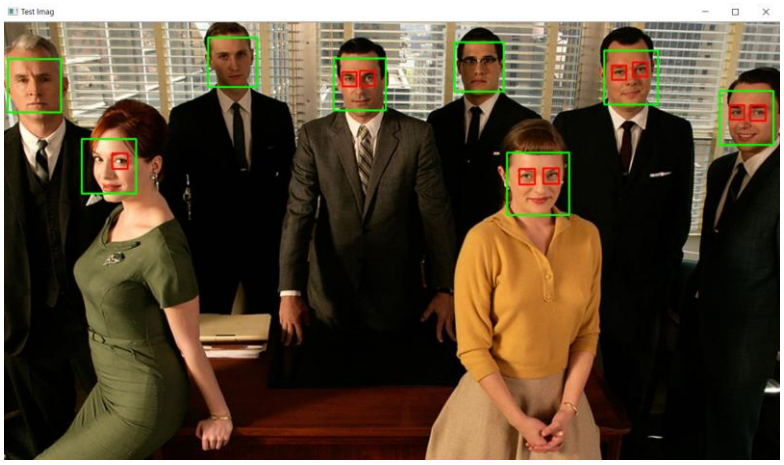


Figure 28: Facial detection using haarcascade_frontalface_default.xml cascade

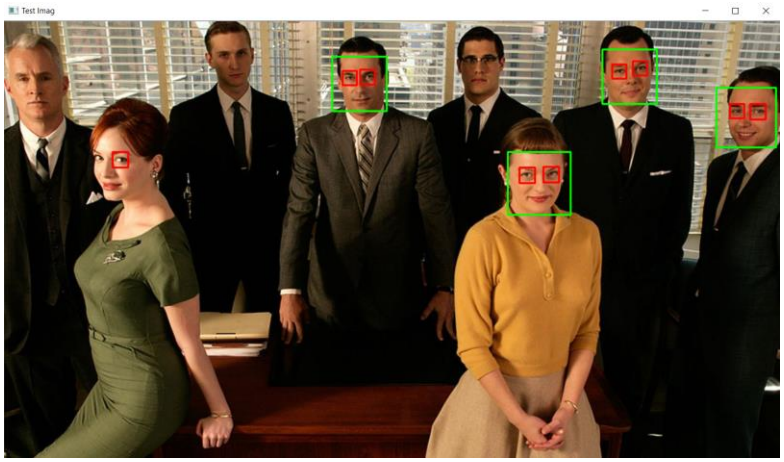


Figure 29: Facial detection using haarcascade_frontalface_alt_tree.xml cascade

Hence, I concluded that the haarcascade_frontalface_alt.xml cascade was the most accurate at frontal face detection. In photos, the haarcascade_frontalface_alt.xml cascade was correctly detecting all the faces. During live webcam feed, the haarcascade_frontalface_alt.xml cascade was also correctly detecting all the faces however it sometimes detected false positive. This could have happened due to different lighting.

For eye detection, I experimented with two cascades, the haarcascade_eye.xml cascade and the haarcascade_eye_tree_eyeglasses.xml cascade.

During live webcam feed, the haarcascade_eye_tree_eyeglasses.xml cascade sometimes gave some false positives however it was more accurate than the haarcascade_eye.xml cascade when it came to detecting eyes with glasses in front of them. However, the haarcascade_eye.xml cascade was overall more able to correctly detect eyes. This is because, unlike the haarcascade_eye_tree_eyeglasses.xml cascade, it was also more accurately detecting eyes when the face was tilted. The haarcascade_eye.xml cascade, returned less false positives and was more accurate in being able to detect eyes without glasses than the haarcascade_eye_tree_eyeglasses.xml cascade. Both cascades have their strengths and weaknesses, however, during live webcam feed, the difference between these two cascades was minor.

Meanwhile, for photos, the haarcascade_eye.xml cascade was much better at detecting eyes than the haarcascade_eye_tree_eyeglasses.xml cascade. Both cascades were failing to detect some eyes, however, the haarcascade_eye_tree_eyeglasses.xml cascade was failing to detect more eyes than the haarcascade_eye.xml cascade. The haarcascade_eye.xml cascade was also sometimes detecting false positives.

In Figure 30, eye detection was performed using the haarcascade_eye.xml cascade whilst in Figure 31, eye detection was performed using the haarcascade_eye_tree_eyeglasses.xml cascade. One can see the difference in these two cascades. In Figure 30, the haarcascade_eye.xml cascade failed to detect 2 eyes whilst detecting 1 false positive and 4

true positives. Meanwhile, in Figure 31, the haarcascade_eye_tree_eyeglasses.xml cascade completely failed at detecting any eyes.

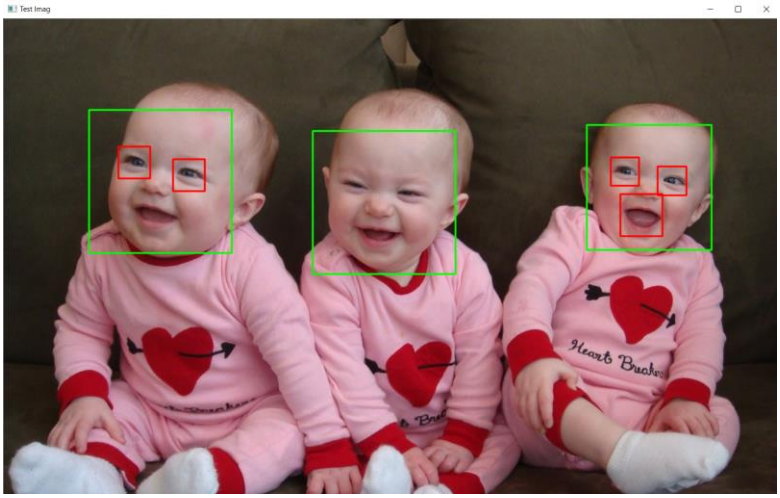


Figure 30: Detection with haarcascade_eye.xml cascade

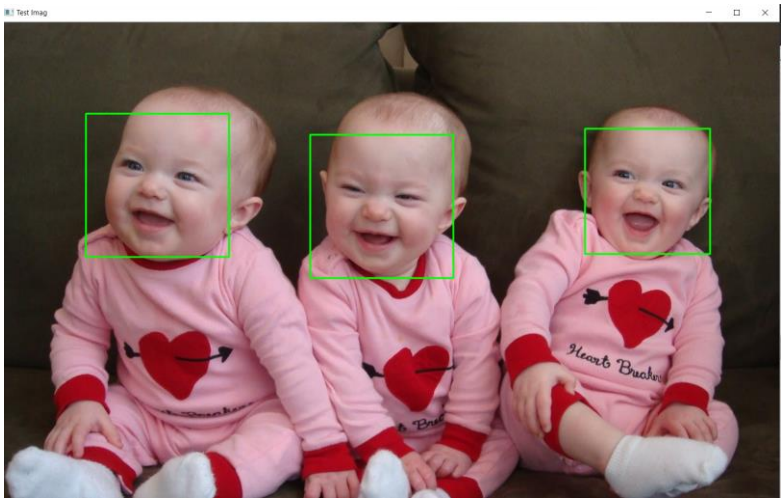


Figure 31: Detection with haarcascade_eye_tree_eyeglasses.xml cascade

Similarly to Figure 30 and Figure 31, in Figure 32, detection was performed using the haarcascade_eye.xml cascade whilst in Figure 33, detection was performed using the haarcascade_eye_tree_eyeglasses.xml cascade. In Figure 32, the haarcascade_eye.xml cascade failed to detect 7 eyes whilst detecting 9 true positive. Meanwhile, in Figure 33, the haarcascade_eye_tree_eyeglasses.xml cascade only detected 1 true positive and failed to detect 15 eyes.

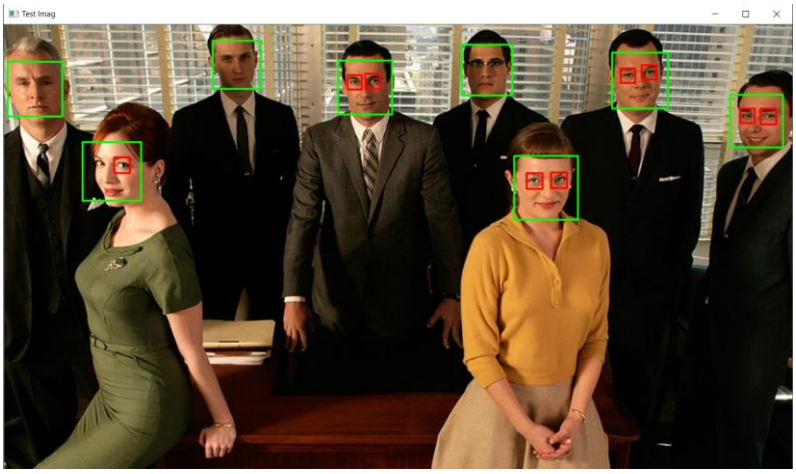


Figure 32: Detection with haarcascade_eye.xml cascade

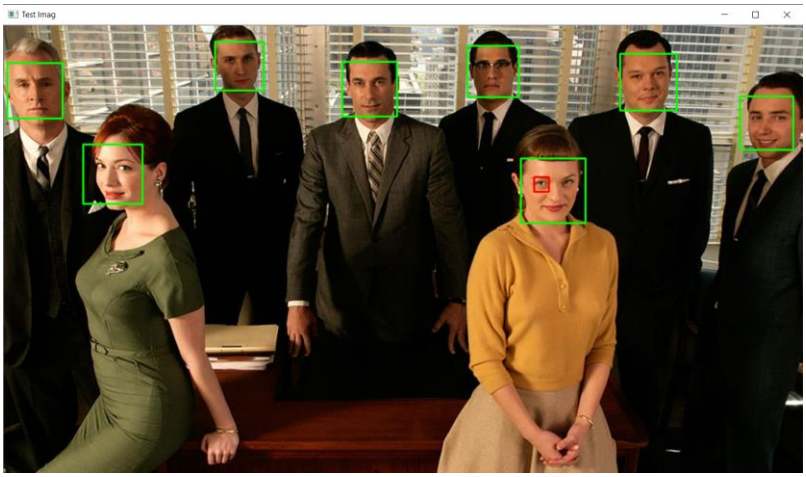


Figure 33: Detection with haarcascade_eye_tree_eyeglasses.xml cascade

Therefore, I reasoned that it is better to have some false positives than to fail to detect some of the required objects. Hence, with this reasoning, I concluded that the haarcascade_eye.xml cascade was overall better than the haarcascade_eye_tree_eyeglasses.xml cascade to perform accurate eye detection in both live webcam feed and photos.

Alternative Approaches

An alternative approach to Viola-Jones for face and eye detection would be the MTCNN. MTCNN stands for MultiTask Cascaded Convolutional Neural Network which is a 3-stage neural network detector used for face detection. It is very accurate as it can also detect faces with different lighting and sizes as well as with strong rotations.

The first stage includes resizing of the image multiple times in order to be able to detect faces of different sizes. Then the proposal network will scan the image to perform first detection. In this stage, many false positives are detected purposely as it has a low threshold for detection. Then, the Refine network filters out these detections in order to obtain precise bounding boxes. This would be the second stage. The third and final stage is the Output network which executes the final refinement of the bounding boxes. Therefore, the bounding boxes as well as the faces detected would be very precise. During the face detection process, MTCNN also detects facial features which can therefore, be used for multiple applications such as face alignment with little to no extra cost.

When compared to Viola-Jones, the MTCNN is slower. However, MTCNN is more accurate and robust from Viola-Jones. MTCNN is able to detect faces even if they are partially obscured or tilted whilst Viola-Jones is not as capable as it only accurately detects full front facing faces. Unlike Viola-Jones which analyses grayscale images, MTCNN uses coloured images in order to perform detection. MTCNN can make use of a GPU if it is available whilst Viola-Jones can only make use of a GPU in certain implementations. Overall, both Viola-Jones and MTCNN are very good at facial recognition and it depends on the individual person's goal to choose between one or the other. For example, for live webcam detection, Viola-Jones would be better since it is faster. [14]

Statement Of Completion

Artefact 1:	Completed
Artefact 2:	Completed
Artefact 2 – Real time face/eye tracking:	Completed
Experiments and evaluation:	Completed
Technical discussion on Viola-Jones method:	Completed
Alternative Methods:	Completed

References

[1] R. Gupta. “Breaking Down Facial Recognition: The Viola-Jones Algorithm” towardsdatascience.com. https://towardsdatascience.com/the-intuition-behind-facial-detection-the-viola-jones-algorithm-29d9106b6999 (Accessed Dec. 12, 2021)

[2] Great Learning Team. “Face Detection using Viola Jones Algorithm” mygreatlearning.com. https://www.mygreatlearning.com/blog/viola-jones-algorithm/#sh2 (Accessed Dec. 12, 2021)

[3] https://tse3.mm.bing.net/th?id=OIP.5L7w08N7mVMm2xRR5p58zwHaCz&pid=Api (Accessed Jan. 6, 2022)

[4] “Releases” opencv.org. https://opencv.org/releases/ (Accessed Dec. 17, 2021)

[5] “Python 3.7.0” python.org. https://www.python.org/downloads/release/python-370/ (Accessed Dec. 17, 2021)

[6] NVlabs. “ffhq-dataset” github.com. https://github.com/NVlabs/ffhq-dataset (Accessed Dec. 17, 2021)

[7] “Wallpapers and Cool Pictures for 1024x1024” vividscreen.info. https://vividscreen.info/for-1024x1024 (Accessed Dec. 20, 2021)

[8] “Image Map Generator” image-map.net. https://www.image-map.net/ (Accessed Dec. 20, 2021)

[9] M. Zalwert. “OpenCV tutorial—train your custom cascade/classifier/detector” maciejzalwert.medium.com. https://maciejzalwert.medium.com/opencv-tutorial-train-your-custom-classifier-e6f12b274296 (Accessed Dec. 17, 2021)

[10] opencv. “opencv/data/haarcascades” github.com. https://github.com/opencv/opencv/tree/master/data/haarcascades (Accessed Dec. 6, 2021)

[11] informramiz. “Face-Detection_OpenCV” github.com. https://github.com/informramiz/Face-Detection-OpenCV/tree/master/data (Accessed Dec. 6, 2021)

[12] SuperDataScience Team. “Face detection using OpenCV and Python: A beginner's guide” superdatascience.com. https://www.superdatascience.com/blogs/opencv-face-detection (Accessed Dec. 6, 2021)

[13] P. Wife. “Read, Write Images and Videos with OpenCV” pythonwife.com. https://pythonwife.com/read-write-images-and-videos-with-opencv/ (Accessed Dec. 7, 2021)

[14] J. Adamczyk. “Robust face detection with MTCNN” towardsdatascience.com. https://towardsdatascience.com/robust-face-detection-with-mtcnn-400fa81adc2e (Accessed Jan. 4, 2022)