EMBODY

Computer Vision Engineering Internship Assignment

Submitted By: Aagam Shah

University of Southern California

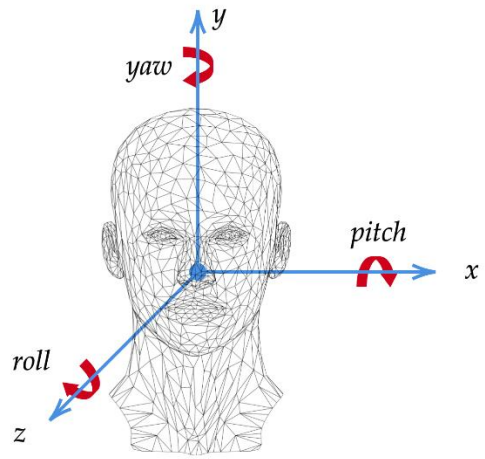
(Viterbi School of Engineering)

Submitted on 9th December 2020

Head Pose Tracker System

1. Introduction and Motivation:

Have you ever wondered that how does the computer track and determines the pose of human face? Isn't that interesting? Let us dive into it and understand the magic behind it.
In computer vision, the pose of any object basically refers to object's relative position and orientation with respect to the camera. The pose can be changed by either moving the camera with respect to the object or vice versa. Pose tracking is basically the task of estimating multi-person human poses in images or videos by estimation of human key-points which may be then useful for human action recognition, human interaction understanding, motion capture and animation.[1]
The Head Pose Tracking and Estimation basically, focuses on the prediction of the pose of a human head in an image. Technically, to be precise it is the prediction of the three Euler angles of a human head. The three Euler angles are: Yaw, Pitch and Roll. These three Euler angles helps us to determine the position as well as the orientation of an object (here head) in the 3D space.



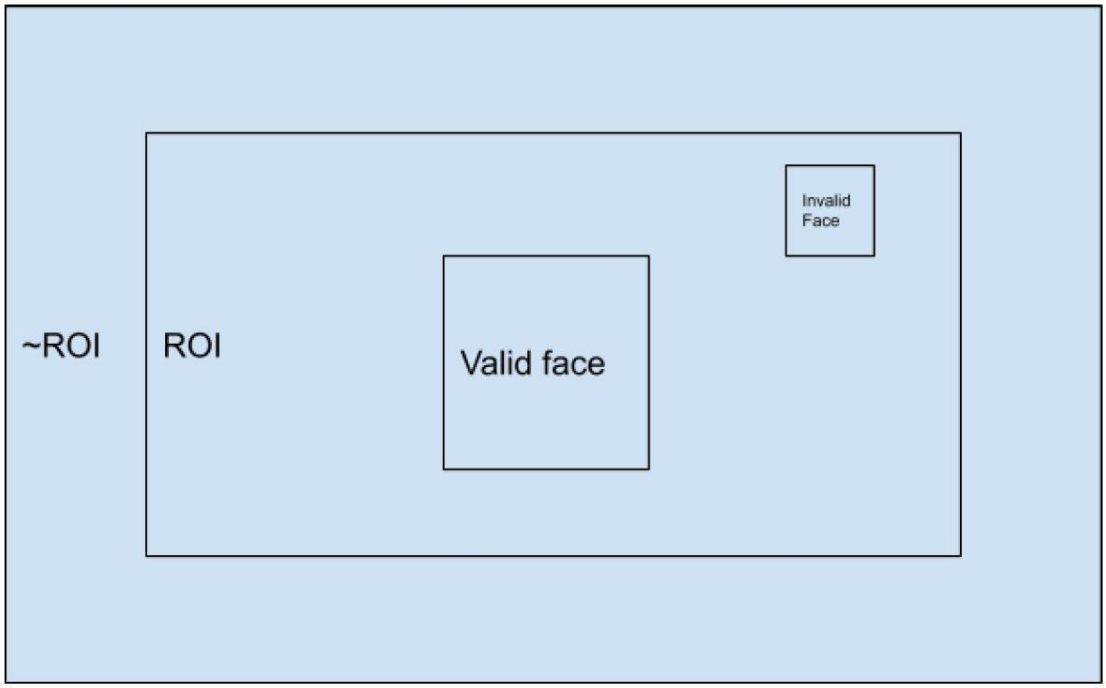
If we precisely predict these three values, then we can easily predict that in what position as well as the orientation a human head is facing. But if these three angles are predicted by the computer to determine that in which direction the human head is facing then it can serve us many useful applications. Applications can be such as mapping of 3D pose of human head similar to that of the Instagram or Snapchat filters. It can also be used in self-driving cars in order to track that whether the driver is focusing on the road or not. Now let's understand the concept and procedure behind each step we perform.

2. Approach and Procedure:

In this section I will describe in detail the approach taken, the concept behind it and the procedure followed to achieve the result. Considering each problem as the step to reach the final solution.

(i) Problem 1:

In this problem I was asked to implement a script that can detect a face in an image, assuming that the face will be within 80% of the image dimensions and ignoring the faces which will occupy less than 10% of area of the Region of Interest (ROI).



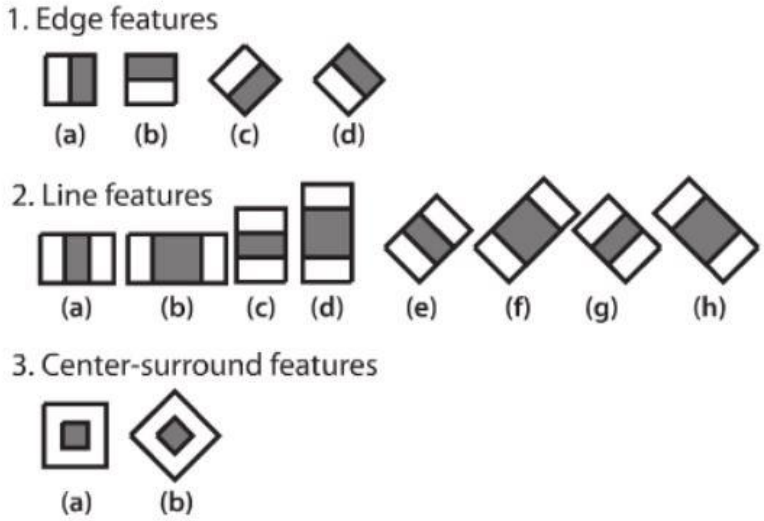
Algorithm and approach:

The main aim of this problem statement is Face Detection, that will determine the locations and sizes of human faces in various digital images. Face Detection algorithm primarily detects the facial features and ignores the rest of the information present in an image such as trees, buildings, other body parts etc.
In this problem I have used the **OpenCV's Face Detection algorithm** which is using **Haar Cascades**.
Let us first understand the basics of the Haar feature cascade classifier mainly used for object detection. This method was proposed by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001.[2]

Face Detection using Haar Cascades:

This algorithm is basically a machine learning based approach where a cascade function is trained by a large number of positive and negative images which is then later on used to detect either faces or objects present in other images.

- Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier.
- Then the features are extracted from it. These are known as Haar features. These features are nothing but similar to the convolutional kernels as shown below.

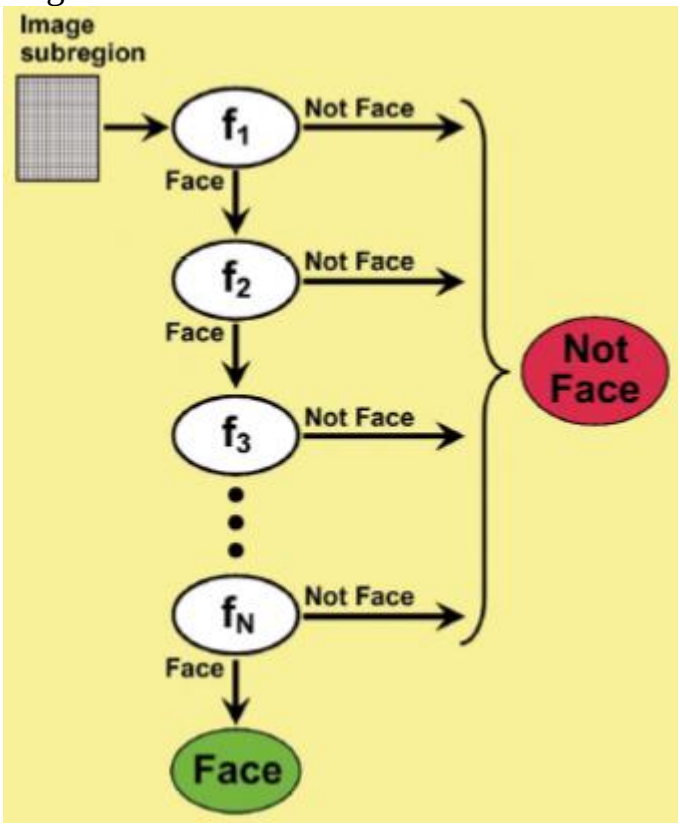


- Each feature is basically a single value obtained by subtracting the sum of the pixels under the white colored rectangle from the sum of the pixels under the black colored rectangle.
- Considering all the possible sizes and locations of each kernel, plenty of features are calculated.
- So, for each feature calculation the sum of pixels under white and black rectangle are calculated using the integral images, which basically simplifies the calculation of the sum of the pixels, by just involving four pixels at a time no matter how much ever large may be the number of pixels. This technique makes the algorithm really fast and efficient.
- But the twist is, that among all the features calculated, most of them are irrelevant. For example, consider the image shown below.

The top row shows the two good features. The first feature focuses on the property that the portion of the eyes is generally darker than the portion of the nose and the cheeks. While, the second feature selected is based on the property that the eyes are darker than the bridge of the nose. But if these same windows are applied on the other parts of the face makes it irrelevant.

- So, now the question is that how the best features are selected out of the total number of features calculated? The solution to this question is that, it is taken care of by the **Adaboost.**
- To be descriptive, each and every feature are applied to the training images and for each feature the best threshold is found which would classify the faces as either positive or negative. But surely, there is expected to be some errors or misclassifications.
- The misclassifications are handled by selecting the features with the minimum error rate, which implies that only those features which the faces or non faces with least error are taken into consideration.
- The above-mentioned process isn't that simple as stated. To handle misclassifications each image is given an equal weight initially, then after each classification step the wights of the misclassified images are increased. This process is repeated in a loop for several times and hence new error rates are calculated every time along with the new weights. This procedure is repeated until the desired accuracy or the error rate is not achieved.
- The final classifier obtained is nothing but the weighted sum of all the weak classifiers. The reason it is being called as weak is because it alone is not able to classify the face and non-face in an image, but together combined with other forms of classifier leads to the strong classifier.
- As cited by paper, even 200 features are capable of providing an accuracy of 95% [2]. The authors of the paper had 6000 features in their final setup [2].
- So, it works like, an image is taken and a 24x24 window is chosen and 6000 features are applied to it to detect that whether there is a face or not. This approach is still inefficient and time consuming. So, a better & efficient solution have been found by the authors of the paper itself.
- As most of the region in an image is non-face region, so the idea proposed by the paper is that to have a simple method to check that whether a window is a face region or not. If it isn't a face region then discard it in a single shot and not to either process it again. Instead the more preference should be given to the region where there can be a face. Through this approach more importance can be given to find a possible face region.
- In order to achieve the above-mentioned approach, the concept of **Cascade of Classifiers** was introduced. So, the basic idea is that

instead of applying all the 6000 features obtained on a single window, the features are grouped into different stages of classifiers and then they are applied one-by-one. Generally, the first few stages are bound to contain a smaller number of features. If the window fails to detect in the first stage itself, then we discard it and don't consider the remaining features on it. But if the window successfully detects, then it is applied to the second stage of features and this process is continued is a similar fashion till the window passes all the stages which will be then the face region.



- As per the paper the authors had 6000+ features with 38 stages. The first five stages had 1, 10, 25, 25 and 50 features. But the authors of the paper claim that, on an average, out of 6000+ features only around 10 features are evaluated per sub-window.

Implementation of Code:

Step 1: First of all, I load the input image using the OpenCV's inbuilt function ***cv2.imread***. The imread() function simply loads the image from the specified file in an ndarray.
Step 2: Next, in order to check if the input image is loaded properly or not, I view the image using ***cv2.imshow*** function. If the input image is too big to view, then we can use the ***cv2.resize*** function in order to resize it to the viewing window.
Step 3: Then I find the height and width of an input image using ***.shape*** function which returns a tuple in which the first element of the tuple is

height of the image and the second element of the tuple gives the width of an image. Height basically represents the number of pixel rows in the image or the number of pixels in each column of the image array and the width represents the number of pixel columns in the image or the number of pixels in each row of the image array. The third element of the tuple gives the number of channels of color used to represent an image.

Step 4: Next, as per the demand of the problem statement that the face will be within 80% of the image dimensions, so I define the ROI (Region of interest) in which the faces will be mostly located. In order to check that whether the ROI is selected properly or not I again view the ROI image using the ***cv2.imshow*** function.

Step 5: Similarly, I again find the height and width of the ROI image for further usage.

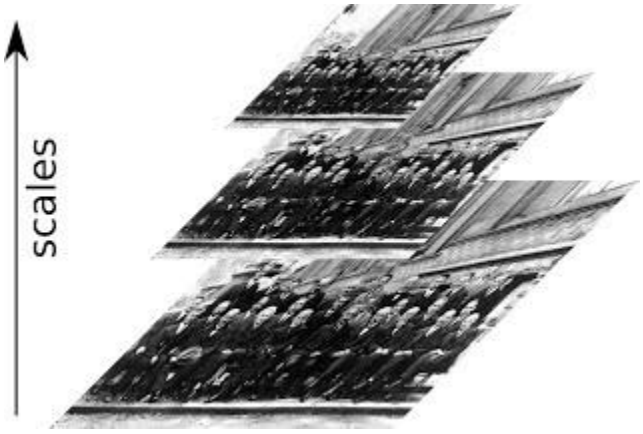
Step 6: As the input image is loaded in BGR format and the full RGB information isn't necessary for the facial detection. Also, the color image holds a lot of irrelevant information on the image which makes computationally expensive and inefficient to work with the color images. So, it is a good practice to convert the input image into gray scale image. The other good reason to work with gray scale images is that object detection yields better results in detecting luminance as opposed to the color. Grayscale images are better work with in detecting the difference in intensities of various image's area. As the OpenCV reads the images in BGR format so I changed them to grayscale images using the ***cv2.COLOR_BGR2GRAY***.

Step 7: The most advantageous part of using OpenCV libraries is that it comes with trainer as well as detector. It contains many pretrained classifiers for face detection, eye detection, smile detection, etc. The Haar features are used for detection and they are stored as XML files. In order to access them, it can be done by using a ***cv2.data.haarcascades*** and then I add the name of the XML file for either frontal face or profile face detection. The Haar features for face detection can be chosen by adding the file path to the CascadeClassifier() constructor using ***cv2.CascadeClassifier(),*** which basically uses the pre-trained models for object detection.

Step 8: Once the classifier is loaded, now the task is to perform the detection of either the frontal or profile faces which is done by either ***faceCascade*** or ***profileCascade*** object using ***.detectMultiScale()*** method. These detectors are based on HOG (Histogram of Gradients) and Linear SVM (Support Vector Machines). This basically returns the list of rectangles for all the detected faces. The list of rectangles is nothing but a collection of pixel locations of an image. Each element of the list represents a unique face. The list contains tuples in the form of (x, y, w, h) where, (x, y) are the top-left coordinates of the rectangle & the (w, h) are the values of width & height respectively. There are few hyper parameters that needs

to be tuned as per the requirement. As here, I tuned these hyper parameters for detecting the faces present in the ROI and ignoring the faces occupying less than 10% of ROI area. The summary of the hyper parameter tuning is described as follows:

- **gray –** This parameter calls the grayscale image object which was loaded earlier.
- **scaleFactor –** This parameter is responsible for specifying the rate to reduce the image size at each image scale. Since some faces may be closer to the camera, they would appear bigger than the faces in the back. The **scale factor** compensates for this. The model must have a fixed scale during training phase, so the input images now can be scaled down for the improved detection. This scale Factor is used to create a pyramid structure as shown below.



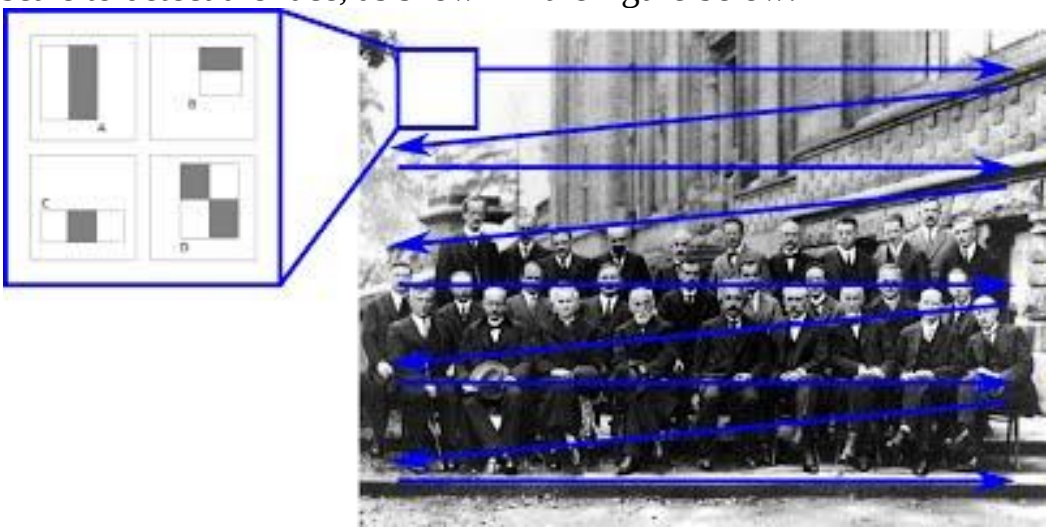
The above structure is known as image pyramid. Image Pyramid is a multi-scale representation of an image, such that the face detection can be scale-invariant, i.e., detecting large and small faces using the same detection window.

This process is terminated after reaching a particular threshold limit which is defined by the max and min Size. For example, if the scale factor is 1.5 it means that it can scale 50% down to try & match the faces in the image better. If the scale factor if increased then the serach time decreases but along with that the accuracy also decreases.

- **minNeighbors** – This particular parameter indicates that basically how many neighbors each detected rectangle should have in order to retain it. This parameter is responsible for the quality of detected faces. A higher value may result in high quality but less detections i.e., it will have less false positives, but a very high value can eliminate the true positives as well.
- **minSize –** This particular parameter is of prime importance for this problem statement atleast. This parameter defines the minimum size of the object that can be detected in pixels. It means that the objects smaller than this size are ignored or not detected. I have set this parameter accordingly, in order to ignore the faces that occupy less than 10% of ROI.

Both the **scaleFactor** & **minNeighbors** parameters are usually arbitrarily chosen and then tuned to set it experimentally as per the need of the input image. I have chosen the values that worked well for this particular setting with the tradeoff between both the hyper parameters.

The detection algorithm uses the **sliding window** approach. In this technique a detection window shifts around the whole image at each scale to detect the face, as shown in the figure below.



The sliding window shifts pixel-by-pixel. Each time the window shifts, the image region within the window will go through the cascade classifier for classification as well as through detection.

Step 9: Then in order to find out the total number of frontal or profile faces detected in an image I have used the *len* function which will calculate the length of the list which contains the unique faces detected i.e., the total number of faces detected in an image.

Step 10: Lastly, I have used the OpenCV's ***cv2.rectangle()*** method to draw a rectangle around the detected faces. In this snippet of code, I have used a ***for loop*** in order to iterate over the list of pixel locations returned from the detection step for each detected object. Even the ***rectangle*** method takes the four arguments which are described as follows:

- **image** – It basically indicates in the code to draw the rectangles on the ROI image.
- **(x, y), (x+w, y+h)** – These are basically the four-pixel locations of the detected object. The ***rectangle*** method will use these coordinates in order to locate and draw the rectangles around the detected objects in the ROI input image.
- **(0, 255, 0)** – This parameter indicates the colour of the bounding box to be drawn around the detected object. This argument is passed as tuple of BGR.
- **2 –** This parameter specifies the thickness of the line of the bounding box to be drawn in terms of pixels.

(ii) Problem 2:

In this problem I am asked to implement a script that can detect multiple faces in an image with all the specifications similar to the Problem 1 but this time I am asked mark the face closest to the centre of the image as the one under the focus with a green bounding box and the other faces with a red bounding box.
The **Algorithm & Approach** for Problem 2 remains similar to that of the Problem 1. Also, the **Implementation of Code** is almost similar to that of the Problem 1 with only one change which is described below.

Implementation of Code:

Almost all the steps of implementation remain similar to the Problem 1 with few additional steps, that are as follows:

- After getting the ROI image (**after Step 5**) i.e., 80% of the original image, I found the mid-point coordinates of the ROI image **(X,Y).**
- After finding the length of the **face list** i.e., the total number of faces detected (**after Step 9**), I declare an empty list as **list**. Then I iterate over all the list of pixel locations of the detected faces in an image and find the **Euclidean distances** between the centre of each bounding box and the centre of the ROI image and append these distances in an empty list created. This step is done basically to mark the face closest to the centre of the image as the one under the focus.
- Then I found the minimum value of the Euclidean distances present in the **list** and its corresponding **index value** which will indicate that which face is closer to the centre of the image.
- The last step of drawing bounding boxes is very much similar to the Problem 1 but this time I mark the face detected closer to the centre of the image (**index value**) with the green bounding box and the rest of the detected faces with the red bounding boxes.

Lastly the final output image is viewed in Problem 1 as well as Problem 2 using the OpenCV's ***cv2.imshow*** function.

(iii) Problem 3:

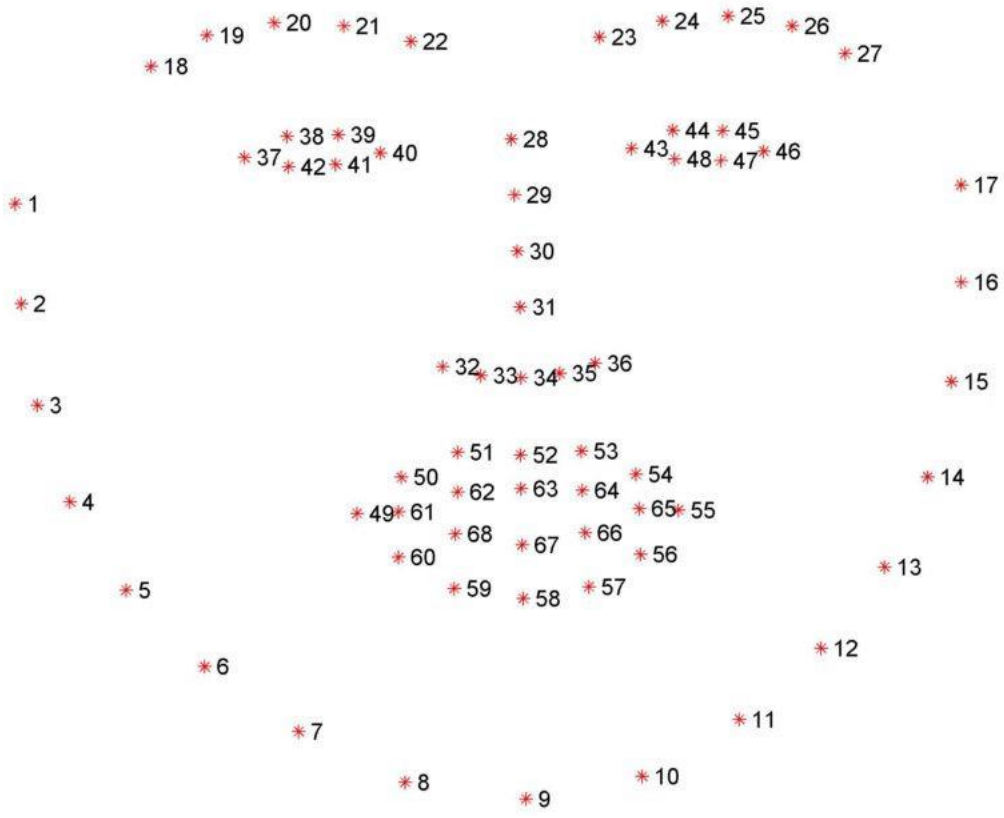
In this Problem I am asked to implement a script that can identify the most commonly known facial landmarks in the image of the detected faces. This is done for both the cases i.e., only for the face in focus and also for all the faces present in the image frame. The list of Facial Landmarks chosen is elaborated below in detail.

As from the Problem 1 and 2 I found out that the major drawback of the Haar Cascade Classifier is that it doesn't work effectively for non-frontal faces, occluded faces and also gives a lot of False predictions.

So, now in order to improve the face detection process and identification of the Facial landmarks I am now using ***dlib.***

Algorithm and Approach:

In this problem I aim to identify the Facial Landmarks of the faces detected. Dlib is a landmarks's facial detector library with the **pre-trained models.** It is basically an advanced machine learning library which is created to solve the complicated real-world problems.
I am using the HoG face detector for this problem. It is one of the widely used face detection model. This detection model is based on HoG features and SVM. To be precise this detection model is built using 5 HoG filters which are front, left, right, front but rotated left and a front but rotated right looking. This model was trained using the LFW dataset, consisting of 2825 images which was manually annotated.
Human face has several features that can be identified, like eyes, nose, mouth, etc. Dlib algorithm is used to detect these features and estimate the location of 68 coordinates (x,y) also known as the landmarks which will map the facial points on the face as given below.



These facial coordinates are obtained from the pre-trained model based on

the iBUG300W dataset. I have chosen the below mentioned landmarks for the face detection as these are the prominent and sufficient landmarks required to detect a human face.

- Jaw Line = 1-17
- Right eyebrow = 18-22
- Left eyebrow = 23-27
- Nose points = 28-36
- Right eye = 37-42
- Left eye = 43-48
- Mouth = 49-60
- Lips = 61-68

These features are the most optimum features which are enough to detect a human face.
So here, the goal is to detect the important facial landmarks on the face which is then use for shape prediction methods.
Facial landmarks detection is therefore a twostep process:

1. **Face Detection:** This step is responsible for the localization of the human face which will return the value in (x, y, w, h) which is a rectangle.
2. **Face Landmark:** After the face localization step is performed and the location of face in an image is obtained, then the key facial points are detected on the face ROI.

❖ The first step i.e., the **Face Detection,** is achieved by **HOG + Linear SVM object detector** approach mainly for the detection of the face. If briefly explained it can be as follows:[3]
Step 1: P positive samples are sampled from the training data of the object or face we want to detect, and the HOG descriptors are extracted from these samples.
Step 2: N negative samples are sampled from the negative training set which would not contain any of the objects to be detected and then the HOG descriptors are extracted from these samples as well. Generally, the negative samples are much larger than the positive samples (N >> P).
Step 3: Now the positive and negative samples are trained using a Linear Support Vector Machine.
Step 4: Now, for each image and at each possible scale of the image in the negative training dataset, the sliding window technique is applied, and the window is slided across the image. For each window HOG descriptors are computed and applied to the classifier. If the classifier fails to classify the object or face in a given window, then it will lead to the false positives. The feature vector associated with the false positive is recorded along with the probability of the classification. This approach is basically known as the **hard-negative mining.**

Step 5: The false positive samples obtained during the hard-negative-mining stage are taken and then sorted according to their confidence (probability) and then using these hard-negative samples the classifier is re-trained.

Step 6: Once the classifier is trained, then it can be applied to the test dataset. Similar to the step 4, for each image in the test dataset and for each scale of the image, the sliding window technique is applied. Also, for each window the HOG descriptors are extracted and applied to the classifier. If the classifier is able to detect an object or face with large probability, then the bounding box is recorded of that window.

Step 7: Once the scanning of the entire image frame is completed, then the NMS (non-maxima suppression) is applied to remove the redundant and the multiple overlapping boxes for the same object or face.

- For the second step, i.e., the **Facial Landmark** detection on the face region, the facial landmarks mainly chosen are mouth, right eyebrow, left eyebrow, right eye, left eye, nose and jawline.
  The Facial landmark detector of the Dlib library is based on the ***One millisecond face Alignment with an Ensemble of Regression Trees*** [4].
  The algorithm behind this approach can be explained in two steps as described below:
  1. The facial landmarks are labelled manually on the images of the training dataset. The (x, y) coordinates of the regions surrounding each facial structure are specified.
  2. The probability of the distance between pairs of input pixels, priors to be more specific, are taken into consideration.

  Now on the training data, an ensemble of regression trees is trained in order to estimate the facial landmark locations directly from the intensities of the pixels. So basically, no feature extraction is performed in this approach. At last, the Facial landmark detection is performed using the facial landmark detector in real time as well as standalone videos with high quality predictions.

Implementation of Code:

Step 1: First of all, at the start of the code I have initialised and loaded the dlib function i.e., ***get_frontal_face_detector*** which will return a detector function that can be used to retrieve the information of the faces. Each face in the image is an object which contains the points where the image can be found. This function will perform the task of face detection in an image.

Step 2: I also declared the predictor model at the start of the code which is responsible for the identification of face features i.e., ***shape_predictor.***

Dlib has pre-defined predictor model in **.dat** file containing the 68 face landmarks in the ***shape_predictor_68_face_landmarks.dat***.

Step 3: I load the input image using the OpenCV's inbuilt function ***cv2.imread***. The imread() function simply loads the image from the specified file in an ndarray. Next, in order to check if the input image is loaded properly or not, I view the image using ***cv2.imshow*** function.

Step 4: Then I found the height and width of an input image using ***.shape*** function which returns a tuple in which the first element of the tuple is height of the image and the second element of the tuple gives the width of an image.

Step 5: Then as mentioned in the problem statement that the faces would be mostly in the 80% of the image dimensions which would be our region of interest (ROI). So, I define the ROI (Region of interest) in which the faces will be mostly located. In order to check that whether the ROI is selected properly or not I again view the ROI image using the ***cv2.imshow*** function.

Step 6: I again find the height and width of the ROI image for further usage.

Step 7: As the input image is loaded in BGR format and the full RGB information isn't necessary for the facial detection and the color image holds a lot of irrelevant information on the image which makes computationally expensive and inefficient to work with the color images. So, it is a good practice to convert the input image into gray scale image. Grayscale images are better work with in detecting the difference in intensities of various image's area. As the OpenCV reads the images in BGR format so I changed them to grayscale images using the ***cv2.COLOR_BGR2GRAY***.

Step 8*:* Now I used the detector function which was declared at the start of the code to detect and retrieve the face information and detect the faces present in an image. The grayscale image is passed through this detector function which will return the list of rectangles for all the detected faces. The list of rectangles is nothing but a collection of pixel locations of an image.

Step 9: Then in order to find out the total number of frontal faces detected in an image I have used the *len* function which will calculate the length of the list which contains the unique faces detected i.e., the total number of faces detected in an image.

Step 10: Then in order to draw the bounding boxes around the faces found in an image I iterate it, in *for* loop over all the faces detected in an image. This is done by using the four coordinates obtained in the list of the **faces** which is in the form of tuple having the (x, y) coordinates of the top left corner and the bottom right corner of the rectangle bounding box.

Step 11: Now in order to plot the facial landmarks for the faces found in the bounding boxes, I have used the **predictor** object which was also declared at the start of the code which will take the grayscale image and

the list of **face** as the input argument and plots the 68 landmarks for the faces detected in the previous step. Each landmark point is marked as a circle with (x, y) coordinates of each landmark point as the center and radius 2 of thickness -1. The color of the circle can be set as any one of the colors as either R, G or B. Here, I have chosen it as green color.
Step 12: Lastly the final output image is viewed using the OpenCV's ***cv2.imshow*** function.

I have implemented this problem considering both the cases i.e., facial landmark detection for the only face under focus as well as for all the faces detected in an image.
The algorithm of implementing the code for the facial landmark detection for the only face under focus which is present at or near to the centre of the image is similar to the one used for the **Problem 2**. Rest everything remains same as described above.

(iv) Problem 4:

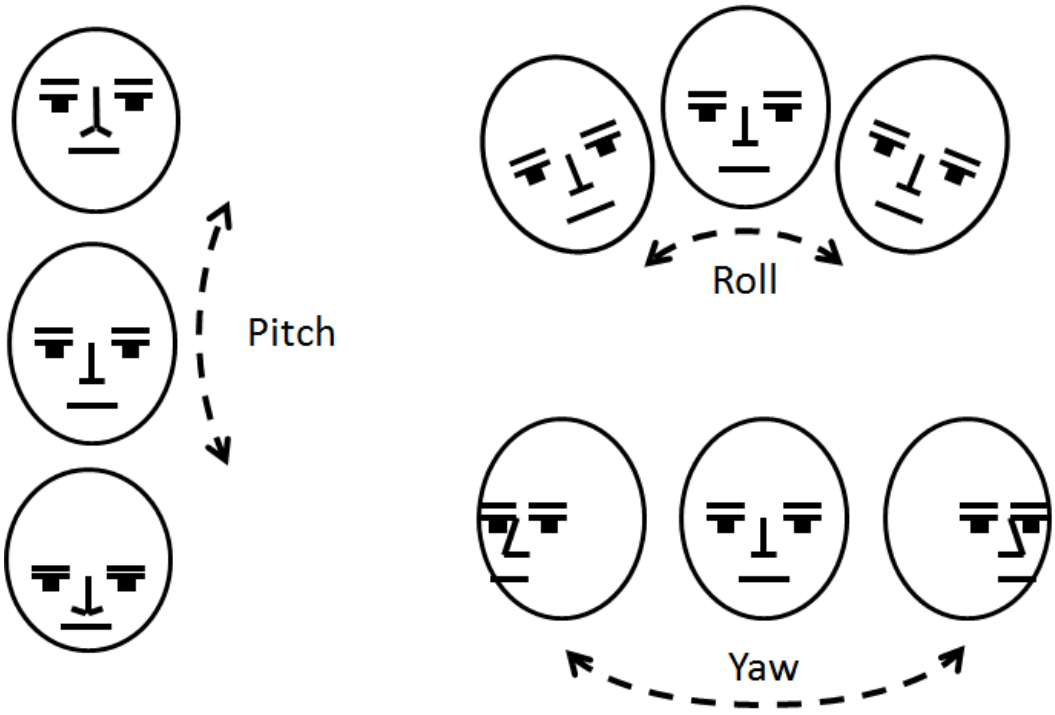
In this I am asked to implement a script that can identify the **head pose** by either using the landmarks or any other algorithm. Also, I am asked to mark the direction in which the person might be looking at, with a vector (theta, phi) which would be centred at the tip of the nose of the person. Again, in this Problem too, this is done for both the cases i.e., only for the face in focus as well as for all the faces present in the image frame.

Even for this Problem I am using Dlib library for the face detection as well as prediction.

Algorithm and Approach:

The main aim of this problem is the Head Pose identification. There are several applications of this technique such as in virtual reality, one can make use of the pose of the head to render the view of the scene. Then in driver assistance system, a camera looking at the face of the driver in vehicle can use the estimation of head pose to make sure that the driver is paying attention to the road or not. It also has applications in the field of gesture-controlled games, gaze estimation, etc.

As we know that face of a human is a 3D object, which means that any object in 3-dimensional space can be rotated over all the three axes. The movements around these 3 axes are known as **yaw, pitch and roll.** It can be effectively visualized as shown in the following figure.



Estimation of these head poses is useful for the **liveliness detection systems.**
I would be explaining in detail the entire concept of pose estimation in the following section.

What's Pose Estimation?

The pose of a human face/object generally refers to the position and the orientation with respect to the camera. The pose can be changed either by moving camera with respect to the human face/object or by moving the human face/object with respect to the camera.
Pose Estimation is usually referred as **Perspective-n-point (PnP)** problem. So, basically the goal here is to find the pose of a human face/object provided we have the 3D location of human face/object, calibrated camera and the corresponding 2D projections in the image.

Mathematical representation of the Camera Motion

Ass it is known that any 3D object will persist only 2 kinds of motion with respect to the camera i.e., **translational and rotational.**

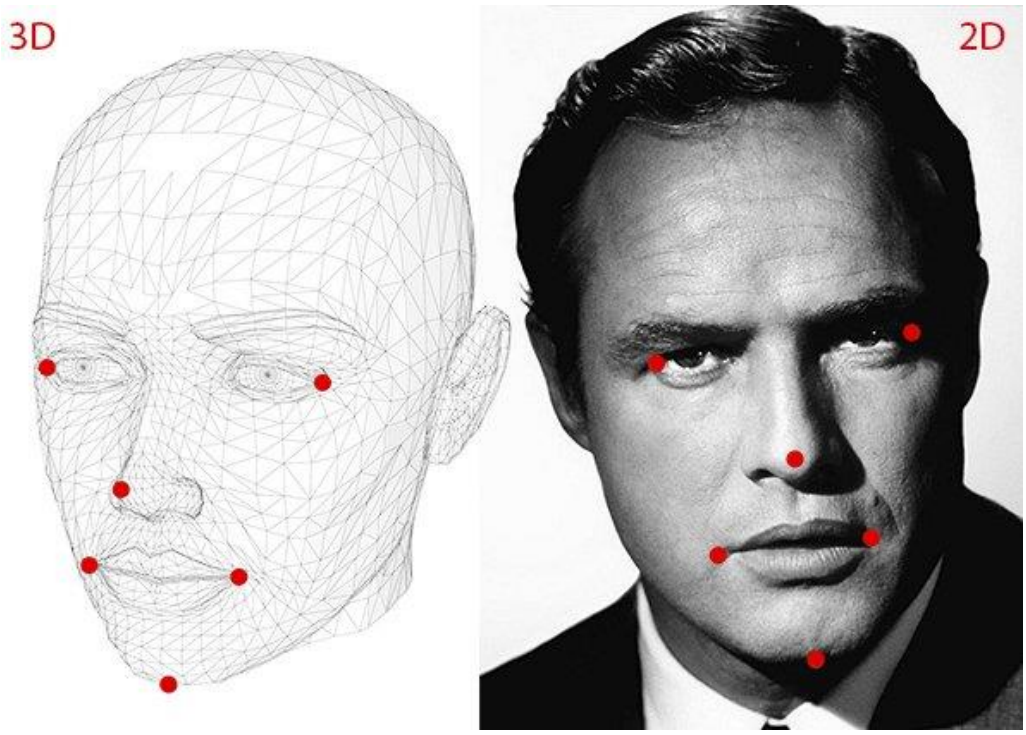
1. **Translational:** The term translation can be defined as the movement of camera from its current 3D location i.e., (X, Y, Z) to the new 3D location (X', Y', Z'). Translational motion consists of 3 degrees of freedom (3 DOF) which means it can be moved in any of the three X, Y and Z direction. Translational motion is generally represented by a vector ***t*** which is equivalent to (X-X', Y-Y', Z-Z').

2. **Rotational:** The camera can also be rotated about the three, X, Y and Z axes. Therefore, the rotational motion also has 3 degrees of freedom (3 DOF). The rotational motion can be represented by making use of the Euler angles as discussed at the start i.e., (Yaw, Pitch and Roll) or by a 3x3 rotation matrix.

   So, in order to conclude that estimating a pose of 3D object indicates the finding of 6 degrees of freedom of an object (3 for translational motion and 3 for rotational motion).

Requirements of Pose Estimation

Let us consider the visual perception of the concept as follows:



To estimate the 3D, pose of a human face/object, the following information would be needed:

1. **2D coordinates of facial landmarks:** The 2D locations of the Facial Landmarks of the detected face in an image is needed. As discussed in the above section, these facial landmarks can be obtained by using the detector and predictor function of the Dlib's facial landmark detector API.
2. **3D locations of facial landmarks:** Along with the 2D coordinates of the facial landmarks, their 3D positions are also required. So, we can say that ideally, we require the 3D locations of the facial landmarks in order to predict the 3D model. So, here for pose estimation the 3D locations of the facial landmarks used are as follows:

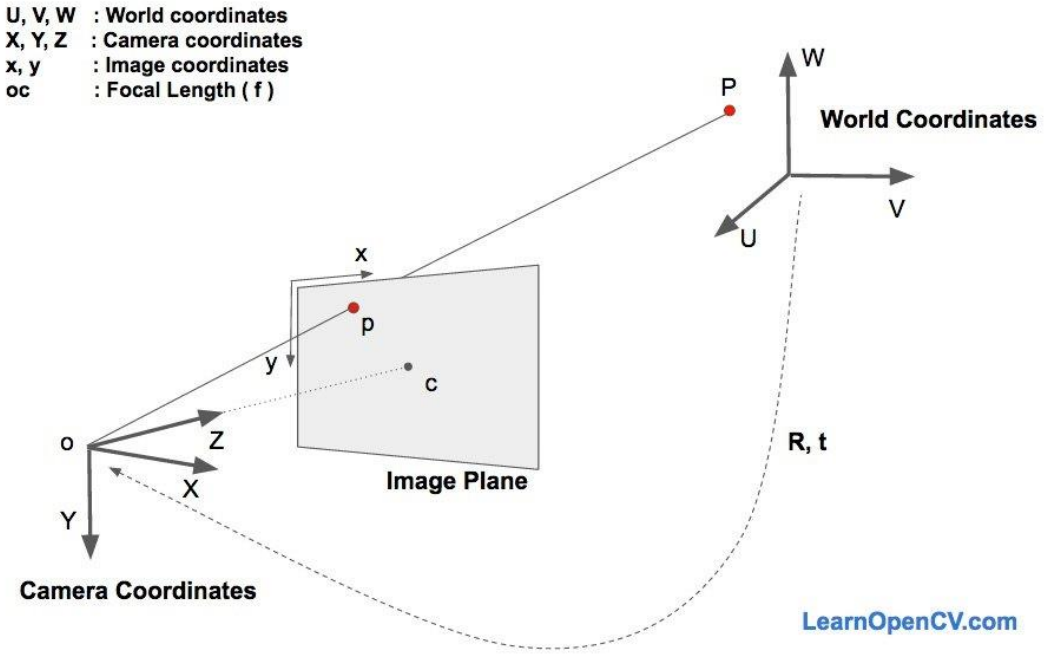
- Nose tip: [0.0, 0.0, 0.0]
- Chin/Jaw: [0.0, -330.0, -65.0]
- Left corner of the left eye: [-225.0, 170.0, -135.0]
- Right corner of the right eye: [225.0, 170.0, -135.0]
- Left corner of the mouth: [-150.0, -150.0, -125.0]
- Right corner of the mouth: [150.0, -150.0, -125.0]

These locations are with respect to the **World Coordinates reference frame.**

3. **Camera Intrinsic parameters:** In order to calibrate the camera, we would require the camera intrinsic parameters in order to tune them. So, basically the **focal length**, its **optical center** and the **radial distortions** are needed. The **optical center** is almost approximated as the image center, the **focal length** can be obtained by the width of the image pixels and the **radial distortions** can be assumed to be null.

Working of the Pose Estimation Algorithm:

As we have the 3D coordinates of the facial landmarks in the World coordinate system. Now if the translational and the rotational information is also known then we could transform the 3D points present in the **world coordinate** system to the **camera coordinate** system. After obtaining the 3D points in the camera coordinate system, these points can be projected onto the image plane which is nothing but the **image coordinate** system. This is achieved by using the camera intrinsic parameters. The entire working of the algorithm can be shown as given in the figure below:



The image formation equations which are used to understand the working of the above coordinate system can be explained as follows.
If carefully observed in the above image, ***o*** is the camera center as well as the center of the image plane. Now the equations which govern the projection ***p*** of the 3D point ***P*** onto the image plane can be found as follows.
Let's consider that the 3D point **P** in the world coordinate system is defined by the location (U, V, W). If the rotation matrix, **R** (3x3 matrix) and the translational matrix, **t** (3x3 matrix) are known of the world coordinate system with respect to the camera coordinates, then the location of point P(X, Y, Z) can be calculated in the camera coordinate system with the aid of the following equation:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{R} \begin{bmatrix} U \\ V \\ W \end{bmatrix} + \mathbf{t}$$

$$\Rightarrow \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = [\mathbf{R} \mid \mathbf{t}] \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

If the above equation is expanded, then it can be shown as:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \end{bmatrix} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

<u>Direct Linear Transformation:</u>

Now many 3D points of the model i.e., (U, V, W) is known but the point (X, Y, Z) is not known. Also, only the location of the 2D point (x, y, z) is known. So, assuming that there is no lens or radial distortion, the coordinates (x, y) of the point ***p*** of the image coordinates can be given as follows:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = s \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Where, $\boldsymbol{f_x}$ and $\boldsymbol{f_y}$ are the focal lengths in the *x* & *y* directions & ($\boldsymbol{c_x}$, $\boldsymbol{c_y}$) is the optical center. The term ***s*** is the scale factor, responsible for the depth or disparity in any image. If the point **P** which is the 3D center is joined to the camera center, **o** and the point **p,** then the line which will intersect the image plane is nothing but the image of point **P.** AS all the points joining with the center of the camera and the point P will lead to the same image, which means that point (X, Y, Z) can be obtained only up to scale factor **s.** So, now the equation will look like,

$$s\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \end{bmatrix} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

The above equation can be solved by Direct Linear Transformation (DLT).

In OpenCV the function **solvePnP** can be used directly to estimate the pose of a human face/object. Various algorithms can be used to solve this problem using the OpenCV's function, which can be selected using the flag parameter. Generally, the **SOLVEPNP_ITERATIVE** which is the DLT solution followed by the Levenberg-Marquardt optimization is used.

Implementation of the code:

Step 1: First of all, at the start of the code I have initialised and loaded the dlib function i.e., ***get_frontal_face_detector*** which will return a detector function that can be used to retrieve the information of the faces. Each face in the image is an object which contains the points where the image can be found. This function will perform the task of face detection in an image.

Step 2: I also declared the predictor model at the start of the code which is responsible for the identification of face features i.e., ***shape_predictor.*** Dlib has pre-defined predictor model in **.dat** file containing the 68 face landmarks in the ***shape_predictor_68_face_landmarks.dat***.

Step 3: Now, for all True condition, the following steps are executed. I load the input image using the OpenCV's inbuilt function ***cv2.imread***. The imread() function simply loads the image from the specified file in an ndarray. Next, in order to check if the input image is loaded properly or not, I view the image using ***cv2.imshow*** function.

Step 4: Then I found the height and width of an input image using ***.shape*** function which returns a tuple in which the first element of the tuple is height of the image and the second element of the tuple gives the width of an image.

Step 5: Then as mentioned in the problem statement that the faces would be mostly in the 80% of the image dimensions which would be our region of interest (ROI). So, I define the ROI (Region of interest) in which the faces will be mostly located. In order to check that whether the ROI is selected properly or not I again view the ROI image using the ***cv2.imshow*** function.

Step 6: I again find the height and width of the ROI image for further usage.

Step 7: As the input image is loaded in BGR format and the full RGB information isn't necessary for the facial detection and the color image holds a lot of irrelevant information on the image which makes computationally expensive and inefficient to work with the color images.

So, it is a good practice to convert the input image into gray scale image. Grayscale images are better work with in detecting the difference in intensities of various image's area. As the OpenCV reads the images in BGR format so I changed them to grayscale images using the ***cv2.COLOR_BGR2GRAY***.

Step 8*:* Now I used the detector function which was declared at the start of the code to detect and retrieve the face information and detect the faces present in an image. The grayscale image is passed through this detector function which will return the list of rectangles for all the detected faces. The list of rectangles is nothing but a collection of pixel locations of an image.

Step 9: Then in order to find out the total number of frontal faces detected in an image I have used the *len* function which will calculate the length of the list which contains the unique faces detected i.e., the total number of faces detected in an image.

Step 10: Now, the 3D locations of the 2D facial landmark points are found using the function **ref3Dmodel** which is custom defined function.

Step 11: The Head pose of the faces detected in an image is found using the facial landmarks. For each face in the image I have done the following steps.

Step 12: In order to find the bounding boxes around the faces found in an image I iterate it, in *for* loop over all the faces detected in an image. This is done by using the four coordinates obtained in the list of the **faces** which is in the form of tuple having the (x, y) coordinates of the top left corner and the bottom right corner of the rectangle bounding box.

Step 11: Now in order to plot the facial landmarks for the faces found in the bounding boxes, I have used the **predictor** object which was also declared at the start of the code which will take the grayscale image and the list of **face** as the input argument and plots the 68 landmarks for the faces detected in the previous step. Each landmark point is connected with another landmark point using the OpenCV polyline function.

Step 12: In this step, the 2D locations of the facial landmarks if obtained using the **ref2dImagePoints.**

Step 13: Next I found the camera matrix using the intrinsic parameters, such as the **focal length and optical center,** assuming the **radial distortion** as zero.

Step 14: Now the **ref3Dmodel, ref2dImagePoints,** camera matrix and the **radial distortion,** these parameters are passed through the OpenCV's **solvePnP** function which will calculate the rotational and translational matrices.

Step 15: After getting the rotational and translational matrices, I can now project the 3D points on to the 2D image plane using ***cv2.projectPoints*** function which takes rotational matrix, translational matrix, camera matrix, radial distortion and 3D position of the optical center which is at nose tip,

as the input arguments and returns the 2D position of the nose tip as well as jacobian matrix.

Step 16: Next, the Yaw and Pitch angles (theta and phi) are found using the 2D reference image points and the 2D nose end points i.e., p1 and p2. Now in order to get the points to estimate the sideways posing of the head, the **head_pose_points** custom defined function is used which will in return the 3D points presented as 2D points to draw the annotation box around the detected face.

Step 17: The p1 and p2 points as well as the annotation boxes are denoted using ***cv2.line*** function.

Step 18: Then the **gaze angles** are calculated with the aid of OpenCV's ***cv2.Rodrigues*** function which takes input as rotation matrix and returns rotational vector for all the 3 axes separately. Using RQ decomposition I found the tilt angles in the x, y and z axes respectively.

Step 19: Lastly, I set the threshold for yaw and pitch angles in order to determine the gazing direction and orientation of the faces detected in an image.

Step 20: For the better understanding of the orientation as well as the direction of head pose, I put text on top of the image indicating the Yaw as well as the Pitch direction using ***cv2.putText.***

Step 21: Lastly, the final output image is viewed using ***cv2.imshow*** function.

I have implemented this problem as well considering both the cases i.e., head pose estimation for the only face under focus as well as for all the faces detected in an image.

The algorithm of implementing the code for the head pose estimation for the only face under focus which is present at or near to the centre of the image is similar to the one used for the **Problem 2**. Rest everything remains same as described above.

(v) Problem 5:

In this problem I am asked to combine all the solutions found in the above problems into one to create a head pose tracker system which can work with standalone videos as well as for the real-time webcam live feed.

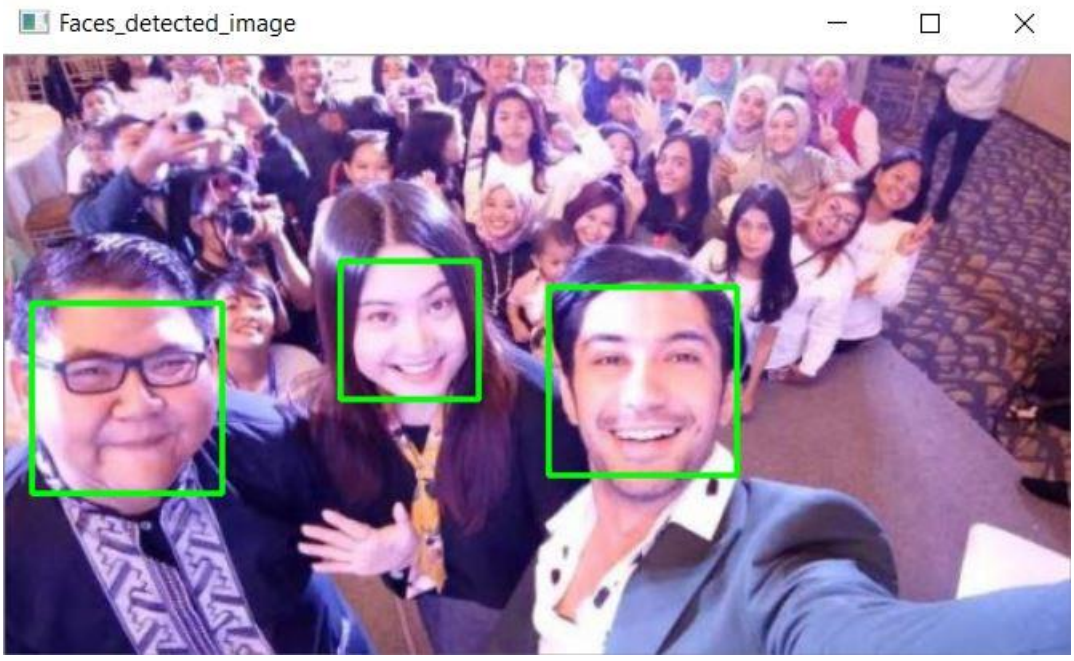
The **algorithm & approach** remains exactly same as described for the Problem 4. Also, the **implementation of the code** part remains similar to the Problem 4 with just a minor difference that the input fed to the system now, will be either standalone videos or real time webcam feed instead of images. So, now the processing would be done on each image frame of the video which would get sampled into multiple image frames and each image frame would be treated as an individual image.

3. Results:

Problem 1:

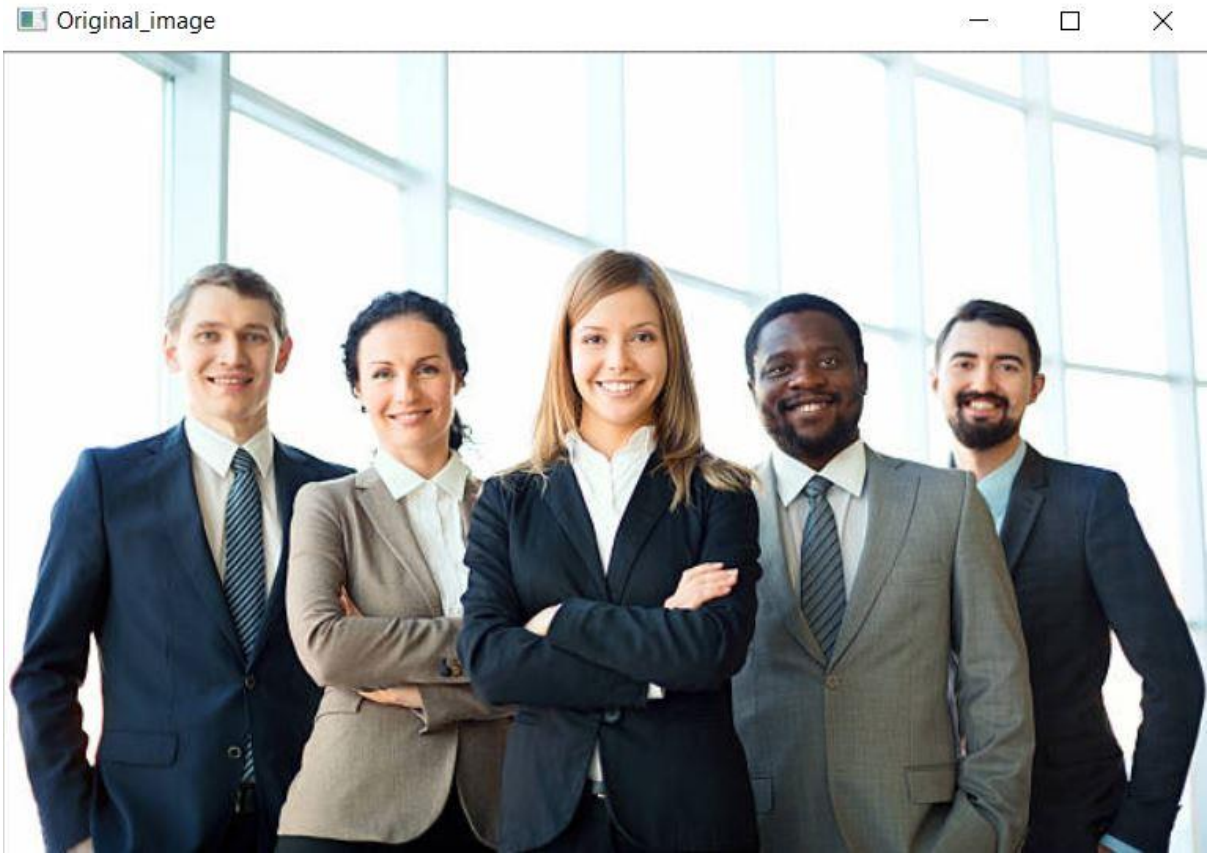
Original Image

80% ROI Image

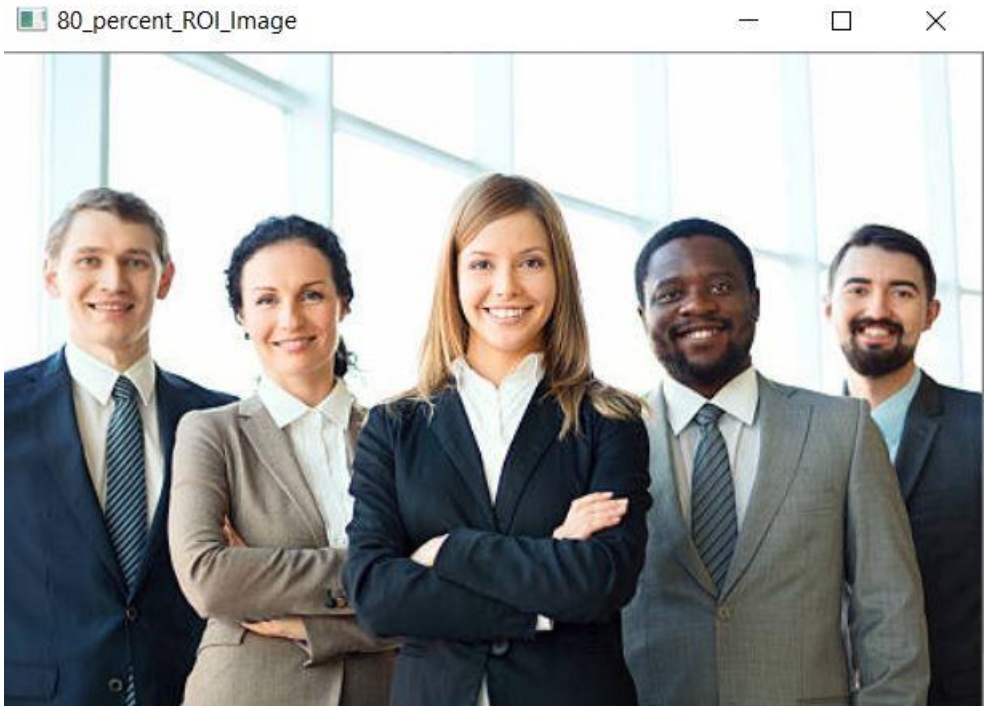


Faces detected Image

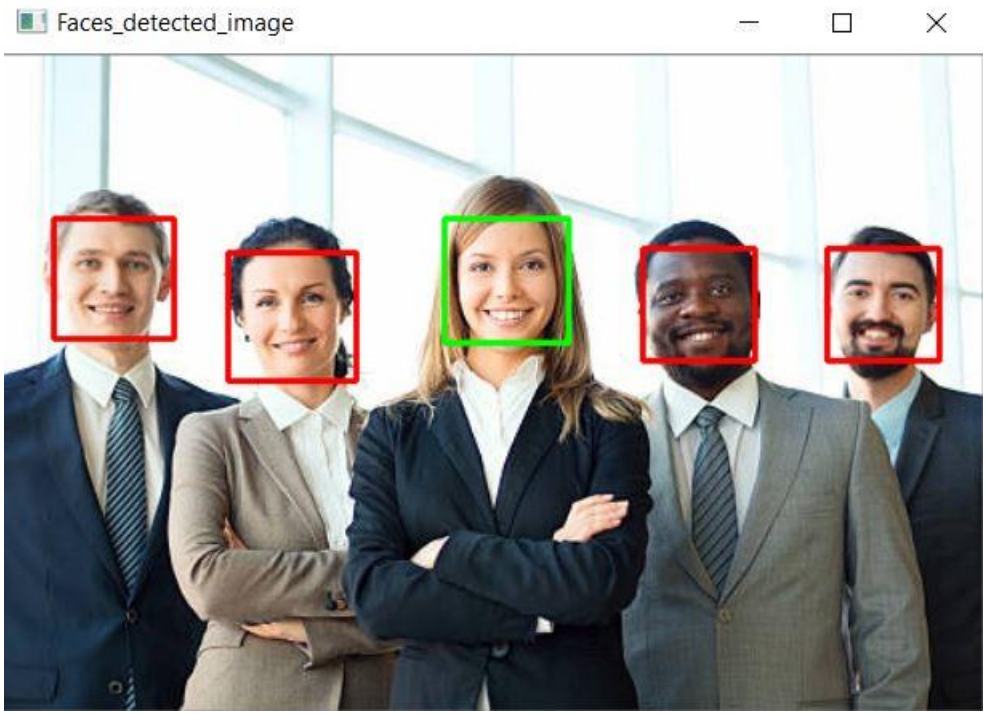
Problem 2:



Original Image



80% ROI Image



Faces detected Image (under Focus)

Original Image

80% ROI Image

Faces detected Image (under Focus)

Problem 3a:

Facial Landmarks detected Image (all Faces)

Problem 3b:

Facial Landmarks detected Image (Face under Focus)

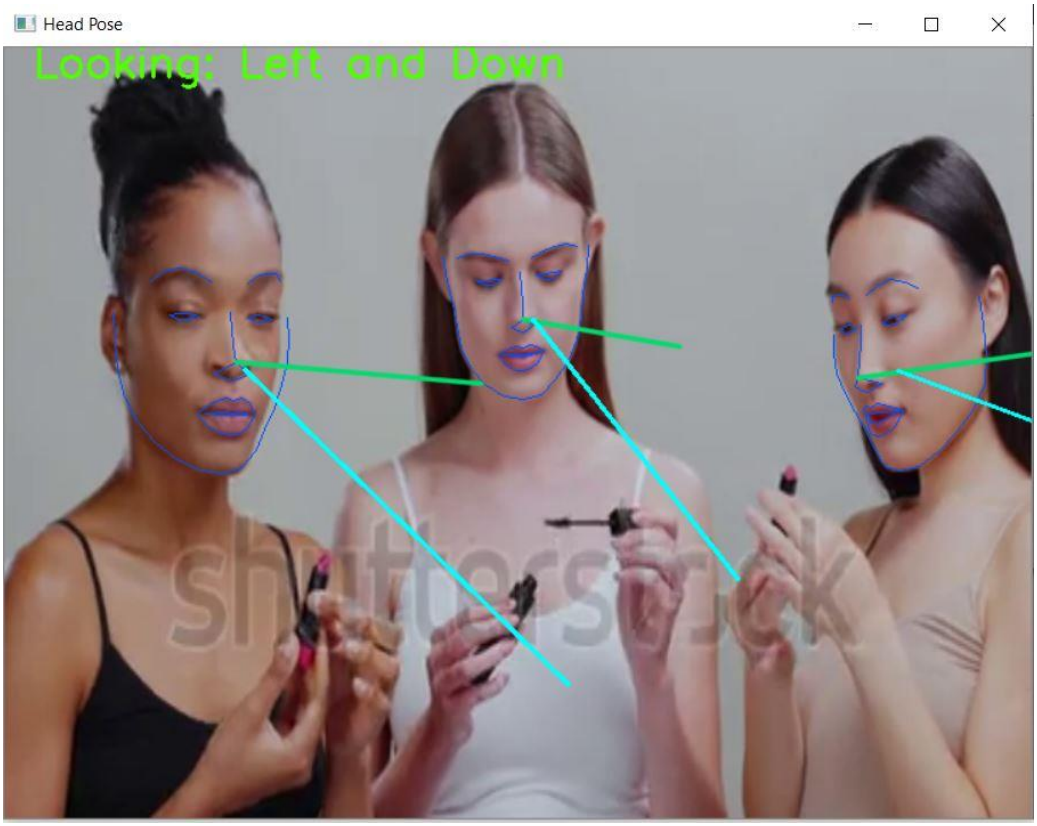
Problem 4a:

Head Pose Identification for all the faces in an image

Problem 4b:

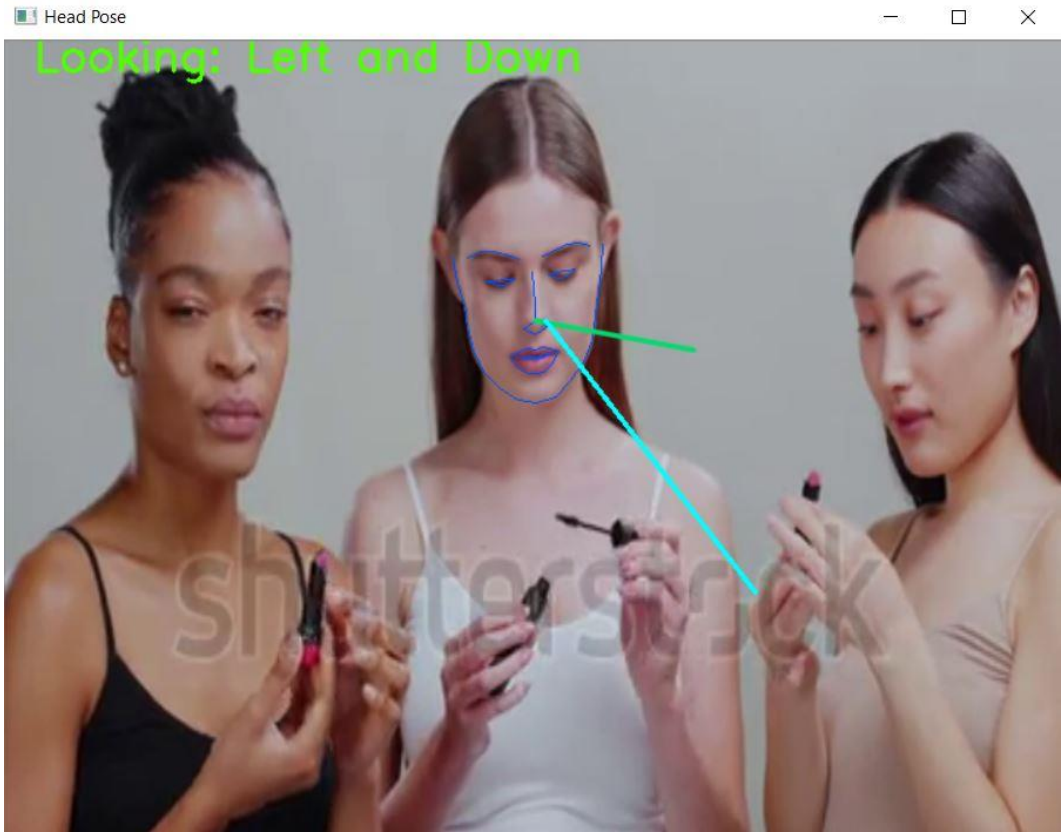
Head Pose Identification for the faces under focus

Problem 5a:



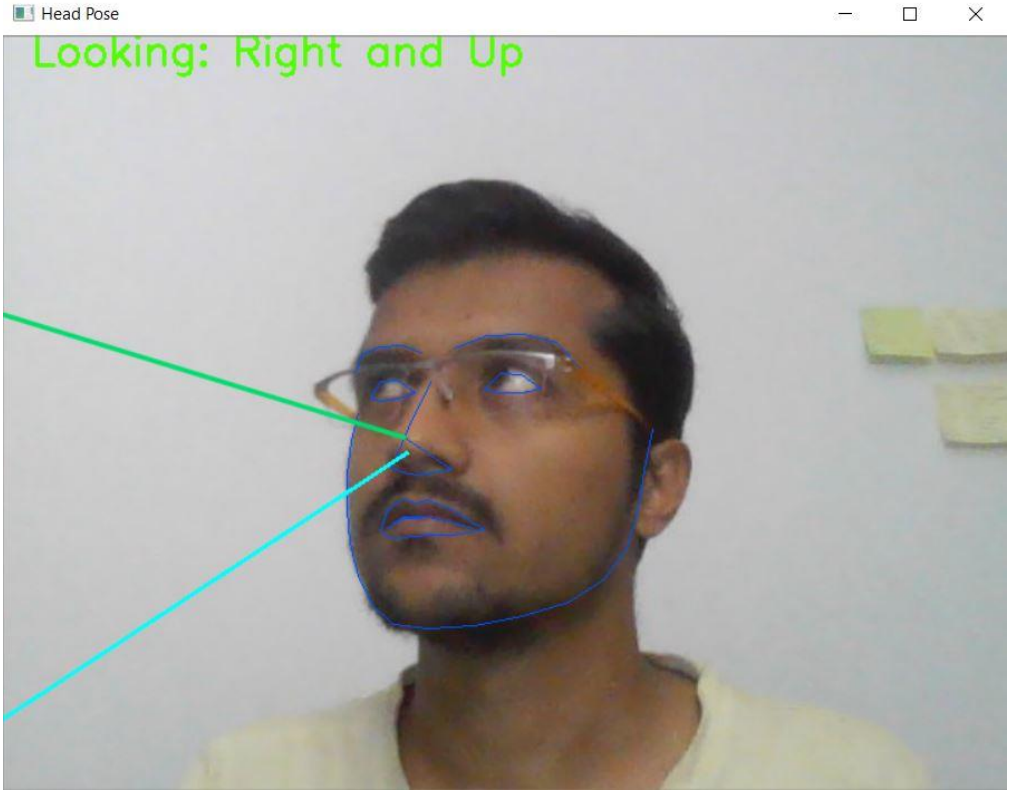
Head Pose Tracker for one frame of standalone video for all faces

Problem 5b:



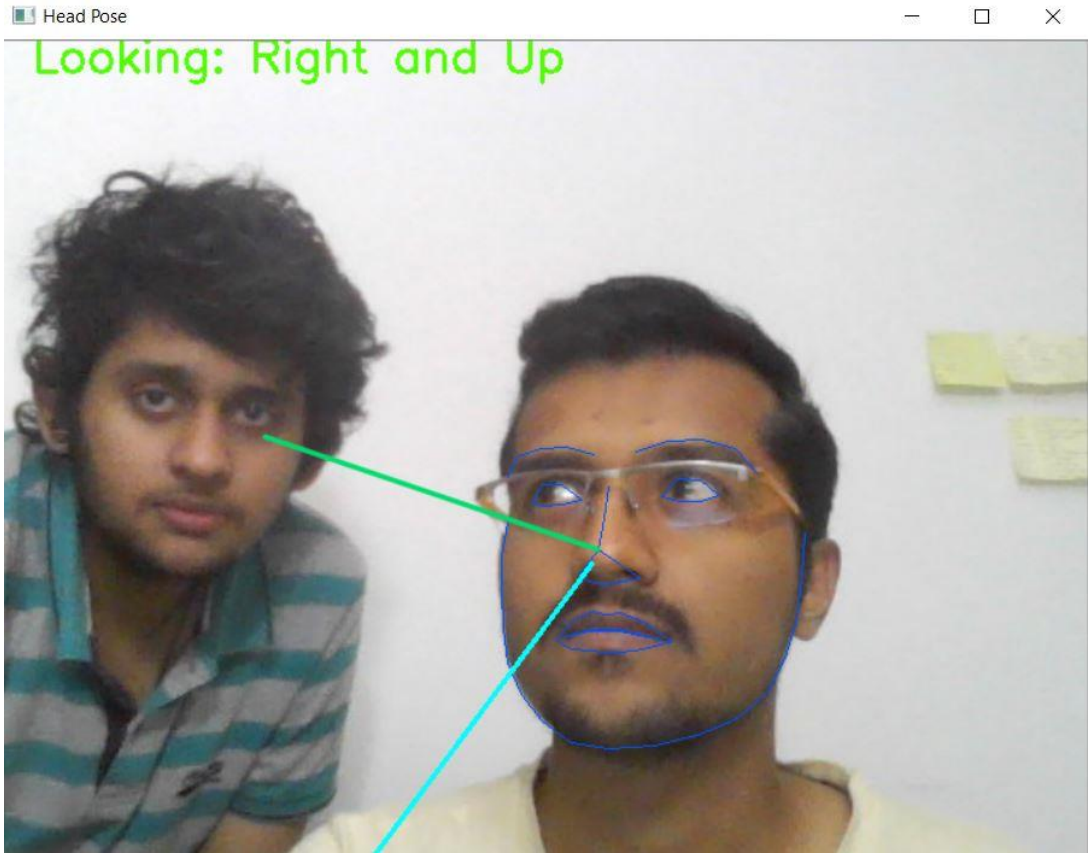
Head Pose Tracker for one frame of standalone video for face under focus

Problem 5c:



Head Pose Tracker for one frame of real time video for all faces

Problem 5d:



Head Pose Tracker for one frame of real time video for face under focus

4. Conclusion:

I would like to conclude that after rigorous testing of the code in all kind of environments it works really good as a head pose tracker or estimation system. I have also addressed all the **Additional Problems** as well by taking all those test cases into consideration while designing of the system.

(i) If the head goes out of the focus and then if it comes back in, my algorithm can start tracking it again as soon as the face gets detected again.

(ii) I am finding the center of all the bounding boxes which detects faces as well as the center of the entire ROI image and then I find the Euclidean distance between each bounding box and the center of the ROI image and store them in the list. Then the minimum value in the list and its index value is fetched which will indicate that which bounding box is closer to the center of the image which in turn means that which face is under focus.

(iii) Yes, I am able to track the faces, facial landmarks as well as pose estimation with foreign objects such as headphones, spectacles and nose ring but not able to detect with mask as it blocks the detection of the

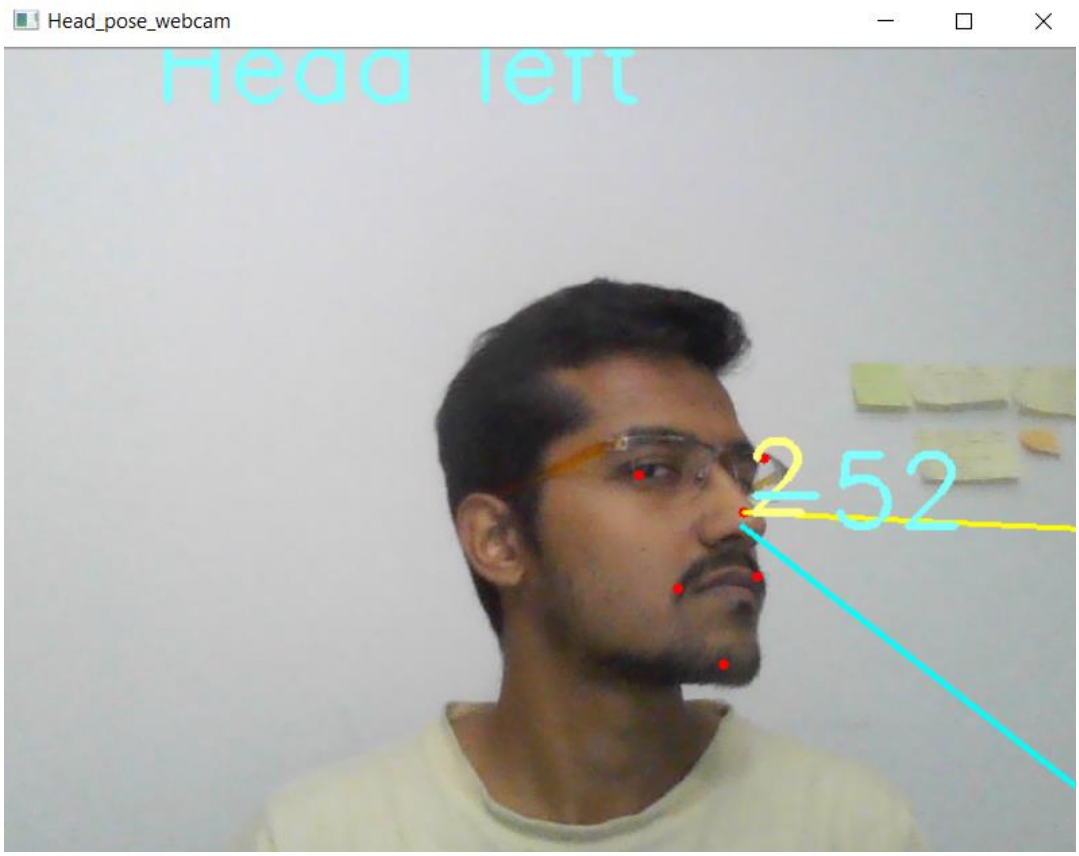
facial landmarks and in turn the pose estimation while there is no such problem with the other foreign objects.

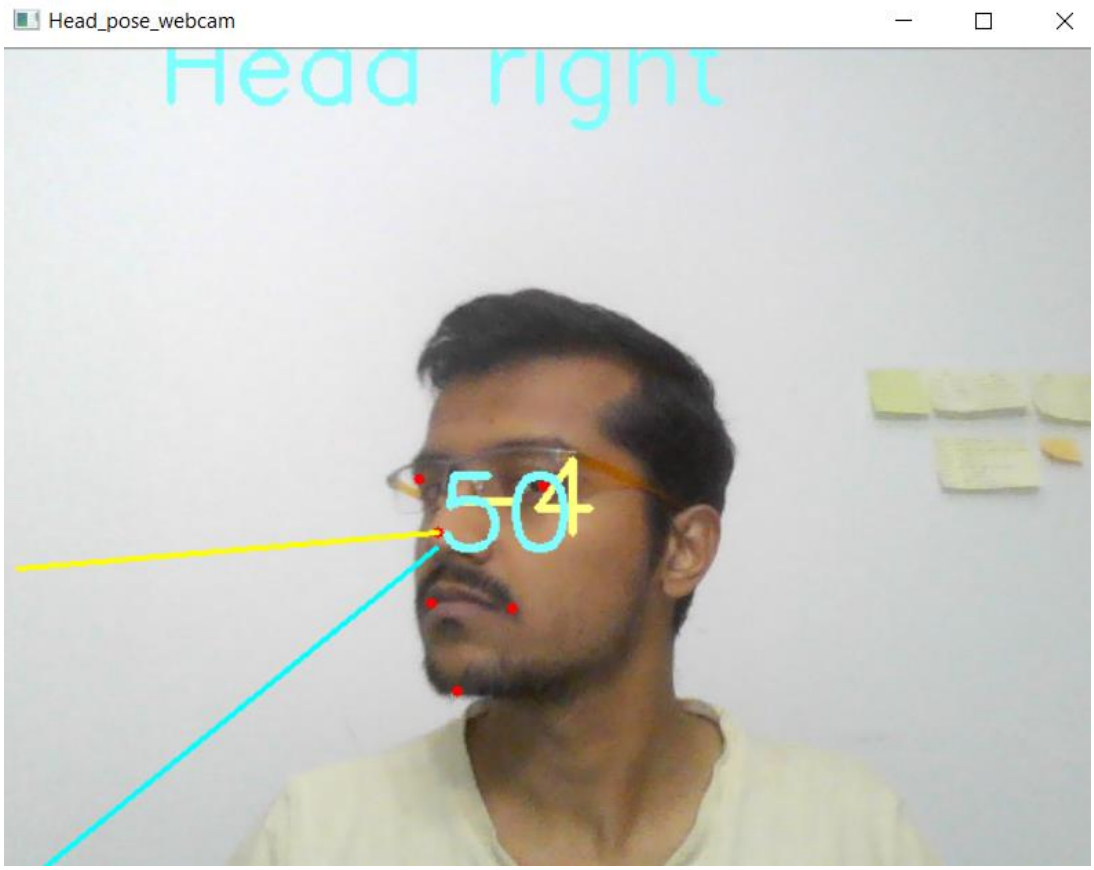
(iv) Yes, if there is an occlusion (for example: if someone is drinking water from a bottle) the tracking will stop as the model fails to fetch the facial landmarks which gets occluded when you are drinking water from bottle or by any other such activities which blocks the detection of facial landmarks the model fails to estimate the pose of the head in such position. But this issue is addressed by the DNN approach described below.

(v) So, as I am keenly interested and very much passionate to work in this domain, I experimented with other algorithm and technique which is based on **Deep Neural Network.**

- So, for this task I used Caffe Model of OpenCV's DNN module for detection of facial landmarks. The network is loaded using ***cv2.dnn.readNetFromCaffe*** and the model's layers and weights are passed as its arguments.
- Also, I found that the Dlib facial landmark detector does not provide extremely good accuracy, so to make an improvement upon that I used TensorFlow's CNN training method trained on 5 datasets provided by Yin Guobing which gives the most accurate 68 landmarks for facial landmark detection. So, instead of performing the entire training step, I used the **Transfer Learning approach** and used the pretrained model which really saves the computational time and complexity. [6]
- The article published by the researcher Yin Guobing extensively describes about the data pre-processing, extraction of faces and stability of facial landmarks even in videos as compared to the OpenCV and Dlib library.
- Also, the importance of the specific loss function when used in model and its effect on the training phase is explained. At first the ***tf.losses.mean_pairwise_squared_error*** was used which is based on the relationship between points as the basis for optimization when minimizing loss but couldn't generalize it properly, but when ***tf.losses.mean_squared_error*** was employed it worked perfectly fine.
- So, in the given pre-trained model, the model takes the square boxes of size 128x128 which would consist of faces and then return the 68 facial landmarks and 3D annotation boxes are drawn for the same using ***draw_annotation_box()*** custom function.
- The **Implementation of the Code** part remains almost similar to Problem 4 and Problem 5 with just the **detector and landmark model** custom written using TensorFlow.
- The only advantage of this approach better accuracy, high sensitivity along-with less computational time and complexity as well as cost effective.

- Results for the above implementation is as follows:

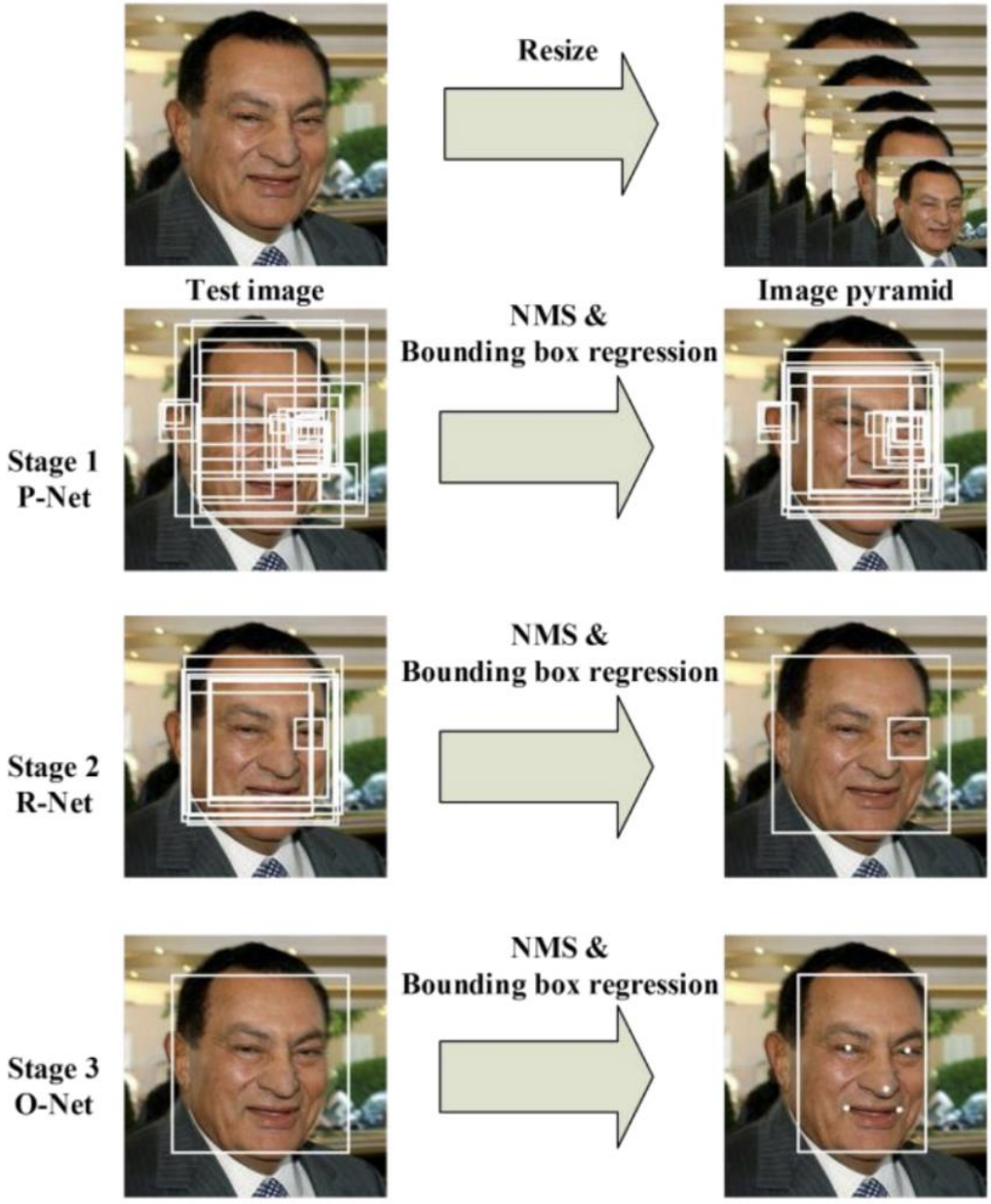



Head_pose_webcam
Head right
504

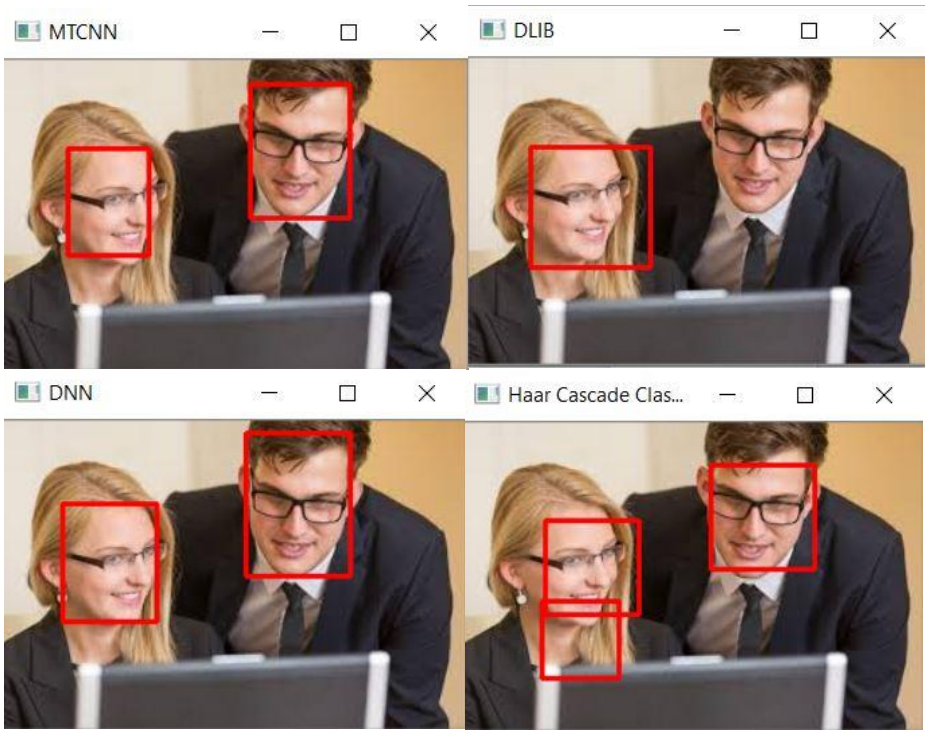

Head_pose_webcam
Head down
417

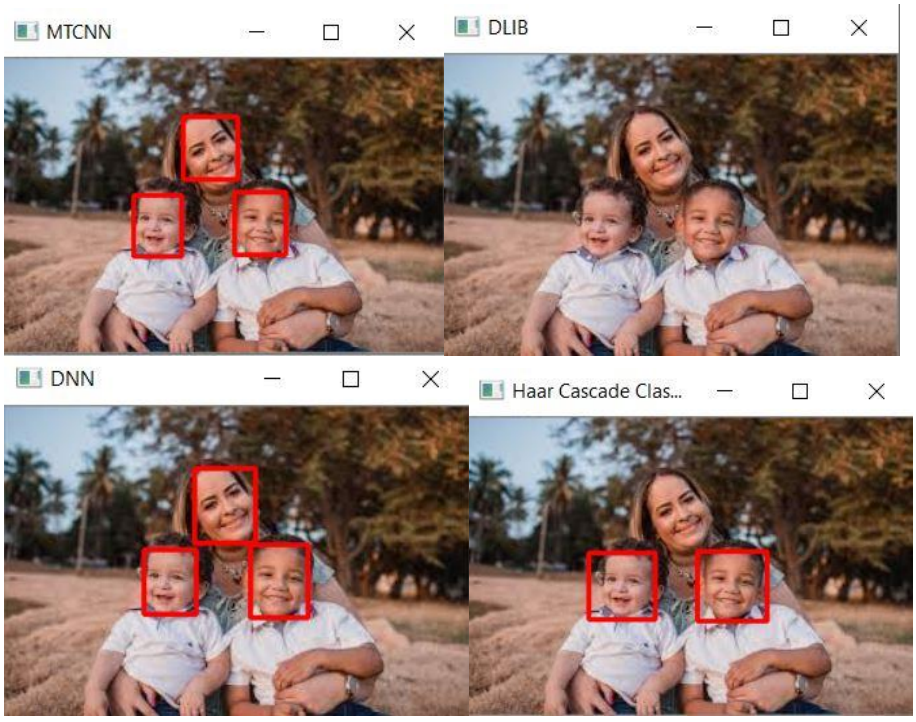
(vi) Finally, I thought of doing comparison just for image data among the algorithms and techniques is experimented with i.e., Haar Cascade, Dlib, DNN and lastly MTCNN (Multi-Task Cascaded Neural Network). It was not possible to do comparison for all the algorithms using the video data as some of them required GPU for better processing.

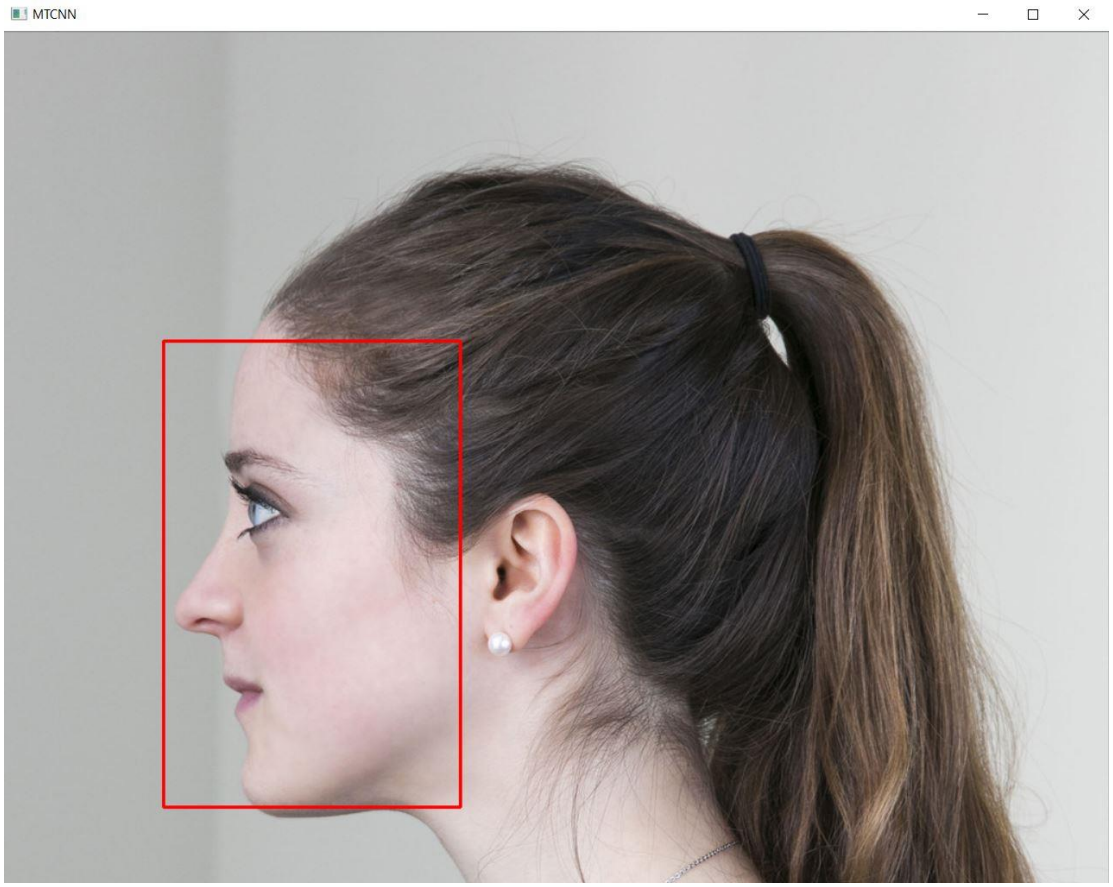
MTCNN is nothing but a network which is based upon the usage of cascaded structure of three networks, firstly the input image is rescaled to a range of different sizes know as the **Image Pyramid** structure, the secondly, in the prior part **Proposal Network/P-net** proposes the candidate's facial regions and then the second model i.e., **Refine Network/R-net** filters out the bounding boxes, lastly, the third model known as the **Output Network/O-net** proposes the facial landmarks.[7] The image from the paper can be shown as follows which gives a summary of the three stages right from the top-bottom approach with the output of each stage going from left to right direction.



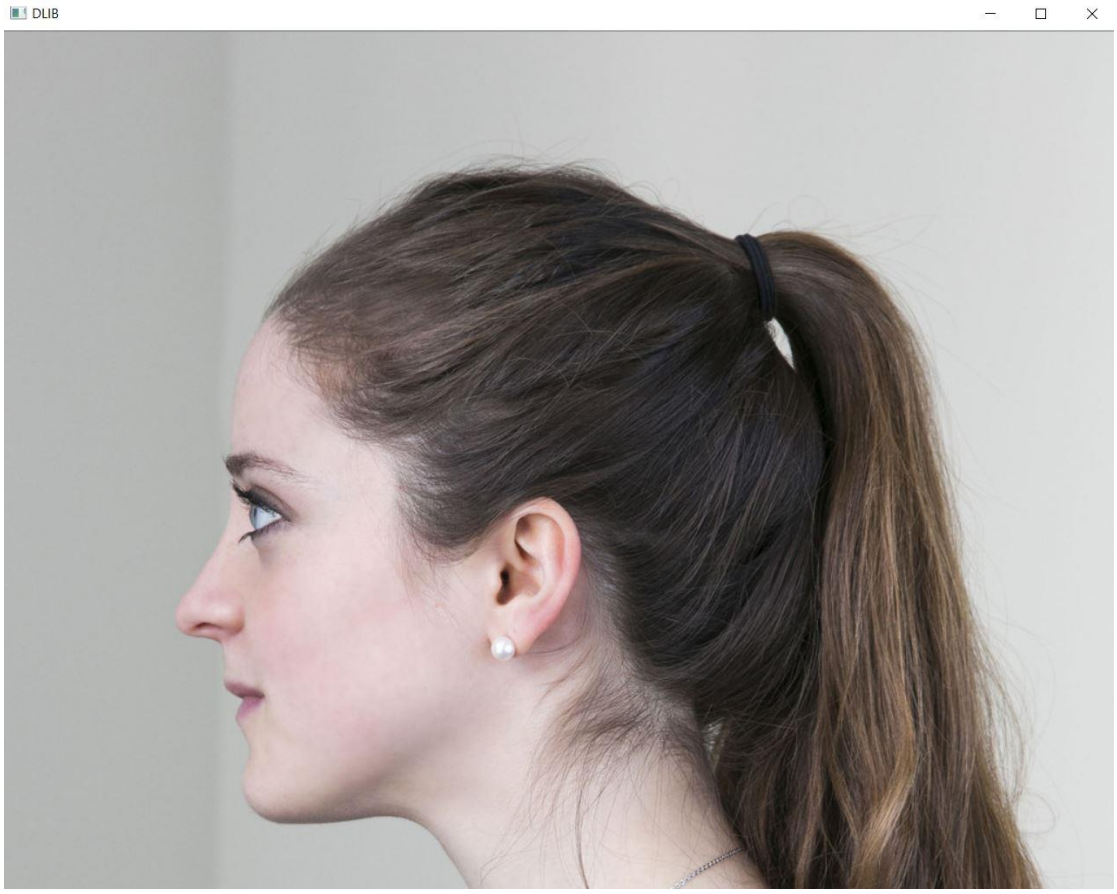
This model basically preforms the three types of predictions i.e., face classification, bounding box regression and facial landmark detection. All the three models aren't connected directly to each other, its just that the output of the previous stage is fed as input to the next stage. This adds one more thing to the processing part i.e., Non-maxima suppression (NMS) in order to filter the cascade bounding boxes which are proposed by the first stage P-net prior to providing them to the second stage i.e., R-net.
Few of the results of comparison can be shown as follows:



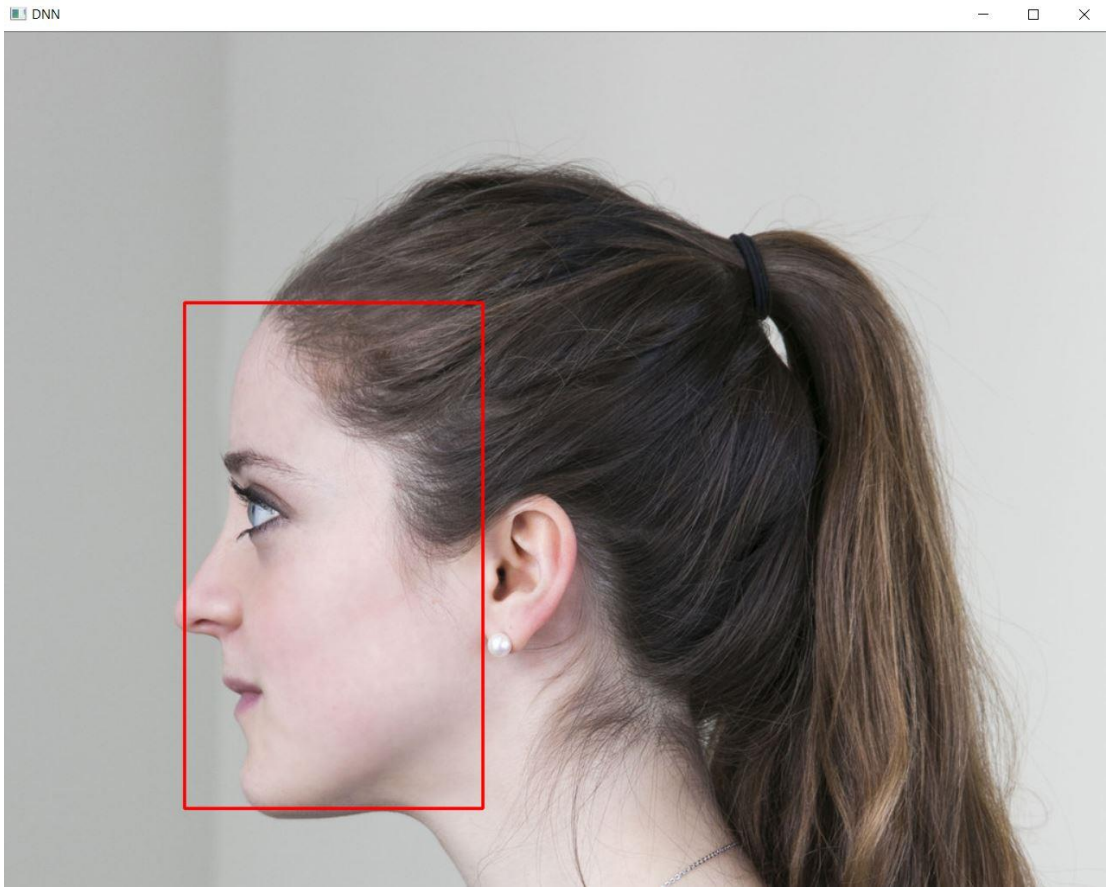




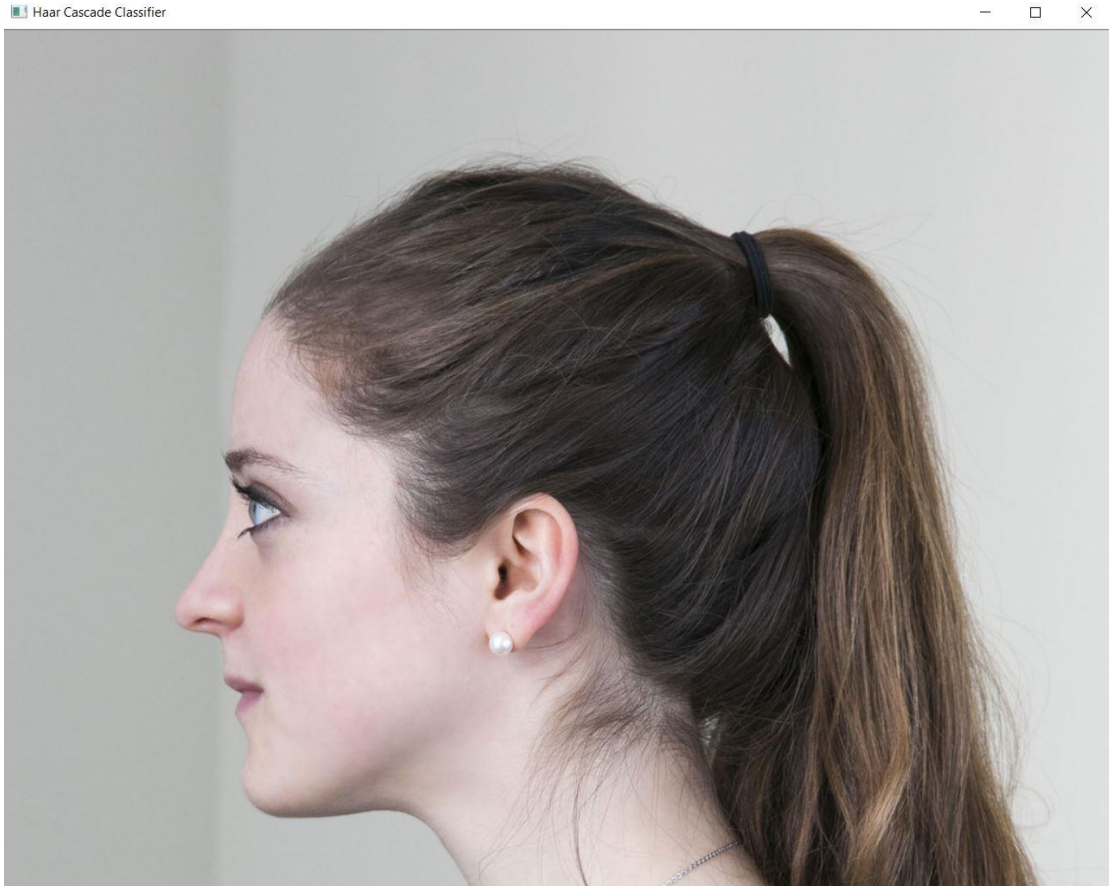
MTCNN



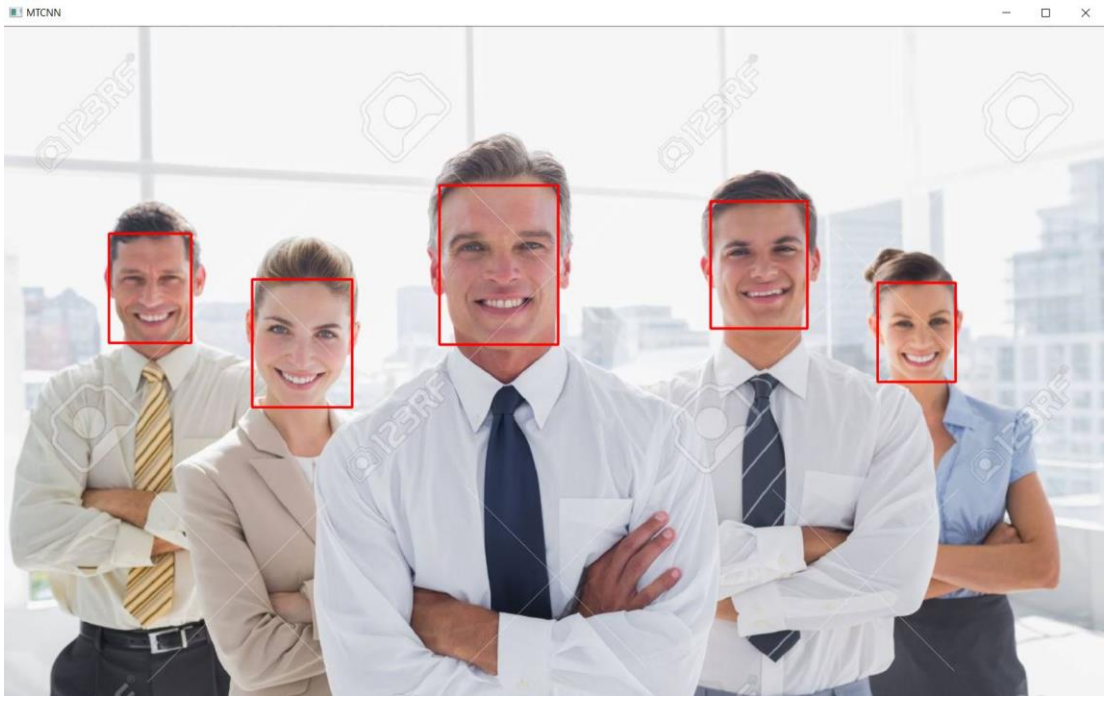
DLIB



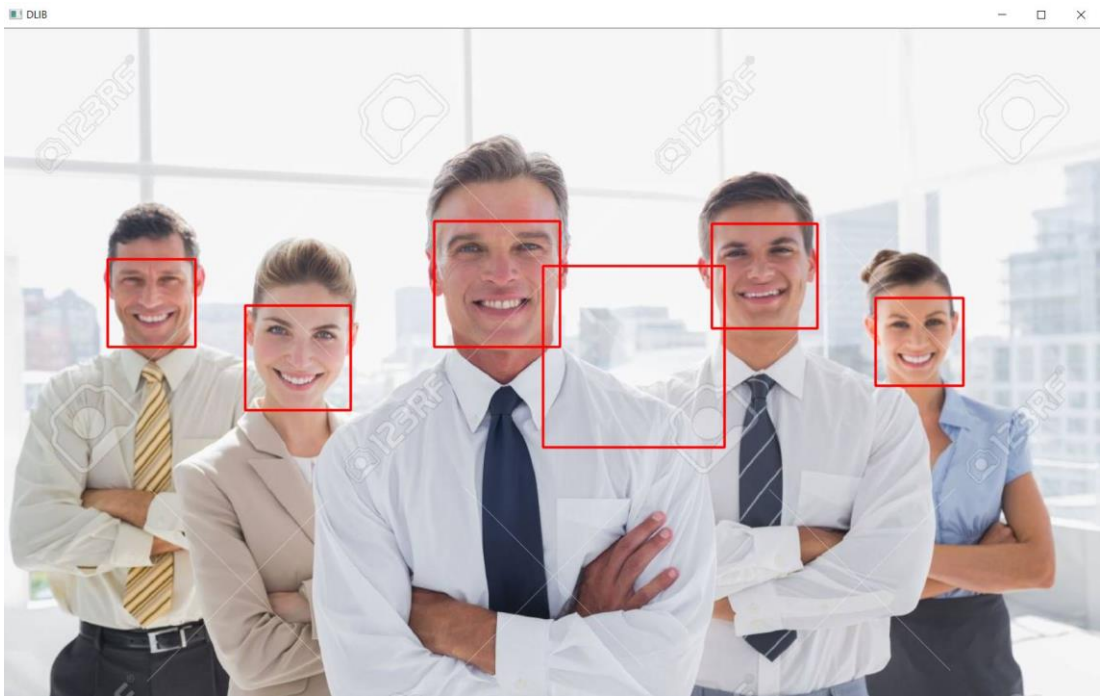
DNN



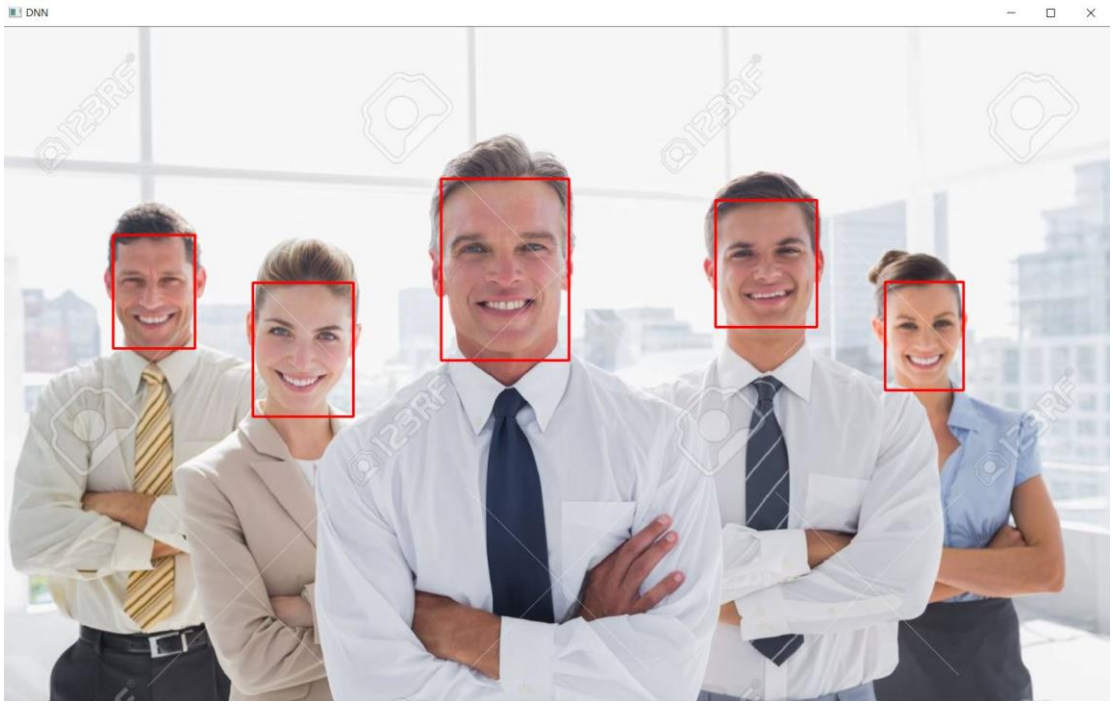
Haar Cascade Classifier



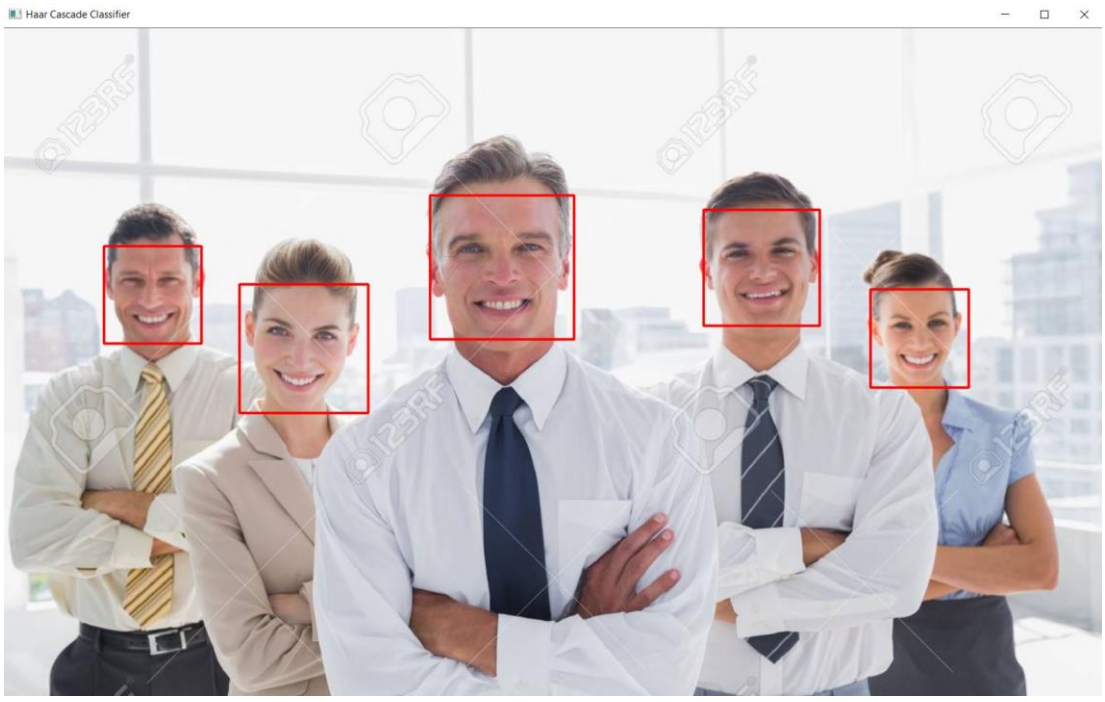
MTCNN



DLIB



DNN



Haar Cascade Classifer

From the above results of comparison between various algorithms I found out that none of the algorithm can be said to be the best or the worst as it completely depends on the application and the environment in which it will be deployed. Each algorithm has its own advantages as well as disadvantages. Its just a matter of fact of choosing the algorithm wisely as per the need of requirement.

5. References:

1. Guanghan Ning, Heng Huang. LightTrack: A Generic Framework for Online Top-Down Human Pose Tracking. arXiv:1905.02822v1 [cs.CV] 7 May 2019.
2. Paul Viola and Michael Jones. Rapid Object Detection using a Boosted Cascade of Simple Features, 2001.
3. https://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/
4. One millisecond face Alignment with an Ensemble of Regression Trees**,** by Kazemi and Suvilian, 2004
5. https://docs.opencv.org/4.1.1/index.html
6. https://yinguobing.com/facial-landmark-localization-by-deep-learning-background/
7. https://arxiv.org/ftp/arxiv/papers/1604/1604.02878.pdf