# University at Buffalo

# Department of Computer Science and Engineering

# CSE 473/573 - Computer Vision and Image Processing

**Spring 2022**
**Project #3**
**Due Date: May 11th, 2022, 11:59PM**

## Table of Contents

## LIST OF FIGURES

# 1. Part A: Face detection.

## 1.1 Performing face detection using different algorithms and comparing the results.

### 1.1.1 Haar cascade:

Description of how the algorithm is implemented:

- Haar Cascade classifiers are an effective way for object detection. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images.
- It is then used to detect objects in other images, here we will work with face detection.
- For implementing the algorithm in my code, first we need to import cv2 library and then I made a list of all the images with another list of filenames corresponding to the image index.
- Then I made a detector using 'cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')' which is the Haar cascade detector available in cv2 library.
- After that I'm converting all the images from color to grayscale and using the gray scale images to detect the faces using the detector.
- I'm using 'detector.**detectMultiScale(gray, 1.2, 4)**' to find faces in image which has 6 parameters as image, scaleFactor, minNeighbors, flags, minSize, maxSize. I'm just using 3 parameters.
- Image- which is the image for which we want to find the faces, scaleFactor- Parameter specifying how much the image size is reduced at each image scale, minNeighbors - Parameter specifying how many neighbors each candidate rectangle should have to retain it.
- After the '. **detectMultiScale'** we get the [[x,y,w,h]] that is bbox elements which I'm appending in my result list along with the corresponding filename. If multiple images are detected the output is in following format [[x1, y1, w1, h1] [x2, y2, w2, h2] ….].

Result and implementation challenges:

- So, after getting the results.json file, the F1 score for this method is: '0.8113207547169811'
- This method gives me the second best F1 score among the algorithms I tried.
- Challenges:
    - While implementing the algorithm the major challenge I faced was to tune the parameters of '. **detectMultiScale'.**
    - I had to tune the scaleFactor and the minNeighbors multiple times to get the best possible F1 score I could achieve with this algorithm.

### 1.1.2 DNN Model:

Description of how the algorithm is implemented:

- For this algorithm I'm using a DNN model with pretrained weights. There are two files which I'm using for the implementation.
- The first file is "opencv_face_detector_uint8.pb" which define the model architecture.
- The second file is "opencv_face_detector.pbtxt" which contains the weights for the actual layers.
- We use the above two files and make a DNN network from them 'net = cv2.dnn.readNetFromTensorflow(model, config)'.
- After I get my network, I'm looping through all the images, for all the images first I extract the height and width of the images.
- For correct predictions from the DNN, first we need to preprocess the images, for that purpose I'm using 'cv2.dnn.blobFromImage' which will preprocess the images for the DNN.
- The parameters which 'cv2.dnn.blobFromImage' has are image, scalefactor, size, mean, swapRB. For my implementation I'm tuning 3 parameters: image- the image which we are using, scalefactor- providing a scalefactor of '0.8', size- the spatial size '227×227'.
- After preprocessing the image I'm putting the image in the DNN and getting the output of the DNN.
- The output of the model gives us the confidence of the predictions and we choose the predictions which have confidence above a threshold level.
- After obtaining the confident predictions, we use those predictions to get the bbox parameters [x, y, w, h] which I append in the result list along with the corresponding filename.

Result and implementation challenges:

- So, after getting the results.json file, the F1 score for this method is: '0.909090909090909'.
- This method gives me the best F1 score among the algorithms I tried. As it gave me the best F1 score, I'm using this algorithm for my face detection part.
- Challenges:
    - While implementing the algorithm the major challenge was to tune the parameters for the preprocessing part.
    - I tuned the scalefactor and the spatial size multiple times before getting the best F1 score for the algorithm.
    - Another challenge was to set the threshold level for the confident predictions. After tuning multiple times, I set it to '0.45'.

### 1.1.3 HOG (Histogram of Oriented Gradients):

For HOG algorithm, there are two libraries which can implement the algorithm, I've implemented using both the libraries.

Description of how the algorithm is implemented (First method):

- The first method was implemented using the 'face_recognition' library.
- I looped through all the images and converted them to grayscale. After converting to grayscale, I used the 'face_recognition.face_locations(img)' which detects the faces in the images and gives the faces location as the output.
- The output is given as [(top1, right1, bottom1, left1), (top2, right2, bottom2, left2), ….] through which we can extract our bbox parameters [x, y, w, h] by simple addition operation for w, h while for x and y they are left and top respectively.
- After getting the bbox parameters, I'm just appending them in the result list according to their corresponding image.

Result and implementation challenges:

- So, after getting the results.json file, the F1 score for this method is: '0.7648902821316614'.
- This method gives me third best F1 score among the algorithms I tried.

Description of how the algorithm is implemented (Second method):

- The second method was implemented using the 'dlib' library.
- First, I'm converting all the images from BGR to RGB channel ordering (dlib expects RGB images).
- After getting the RGB images, I get my detector using 'dlib.get_frontal_face_detector()' which recognizes the faces in the images and gives the output in 'rectangles' object.
- We get the bbox parameters from the 'rectangles' which gives the (top, right, bottom, left) corners of the faces and by using some operations we get the [x, y, w, h] for faces in the images.
- After getting the bbox parameters, I'm just appending them in the result list according to their corresponding image.

Result and implementation challenges:

- So, after getting the results.json file, the F1 score for this method is: '0.7361563517915309'
- This method gives me the least F1 score among the algorithms I tried.

## 2. Part B: Face clustering

### 2.1 Performing face clustering using Kmeans:

Description of how the algorithm is implemented:

- For the part B of the assignment I'm using Kmeans clustering algorithm by using the 'KMeans' library from 'sklearn.cluster'.
- First, we need to get the bbox parameters for the faces in images, for that I'm using the face detection method from part A.
- After getting the bbox parameters for the faces, I'm cropping the faces from the images which I'll be using further to plot the face clusters.
- By using the bbox parameters from each image, I'm using the function 'face_ recognition.face_encodings(img, boxes)' to get 128 dimensional vector for each cropped face.
- By using the computed face vectors, I'm using the 'KMeans' function for clustering the faces which are of the same person. The no of clusters is provided in the code as 'K'.
- After clustering, we get the cluster labels for each cropped face from 'kmeans.labels_'. After getting the labels for each cropped face, I'm making two list in which I'm grouping all the same clustered images as one group and their corresponding file names as one group in each element of the two list respectively.
- After grouping them, I'm appending the filename groups in the result list.

Results:

- After clustering is done, I'm showing the faces in the same cluster by using matplotlib. I'm using my cropped face images in the plot for better visualization. Below I'll be showing the plots for different clusters.

    (i) Cluster 0:

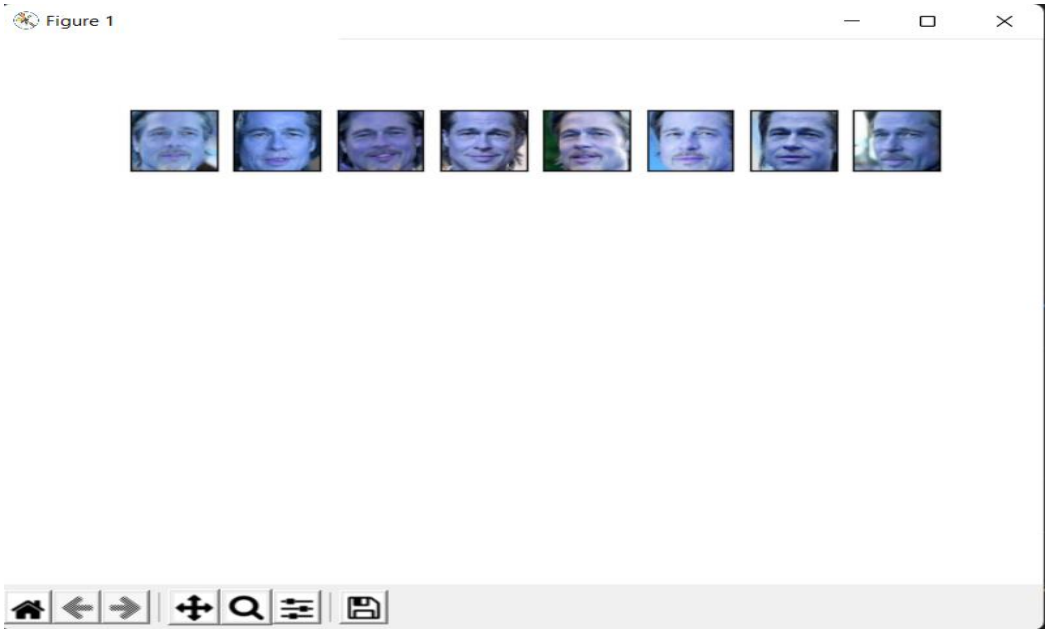


**Figure 1:** *Cluster 0*

(ii) Cluster 1:

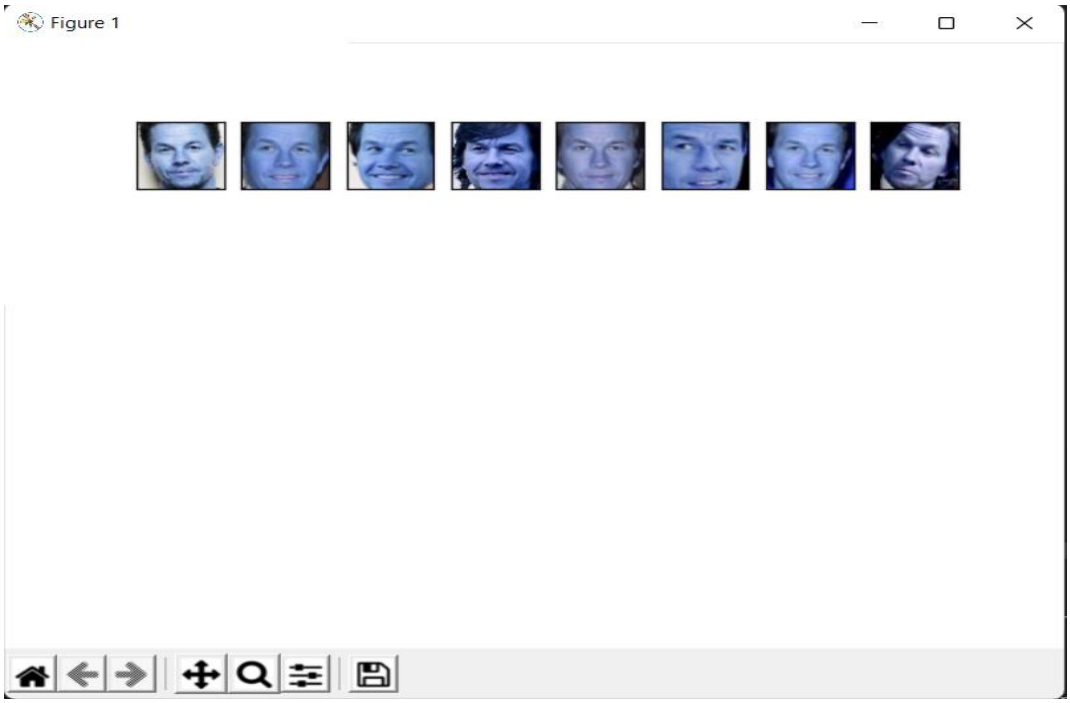


**Figure 2:** *Cluster 1*

(iii) Cluster 2:

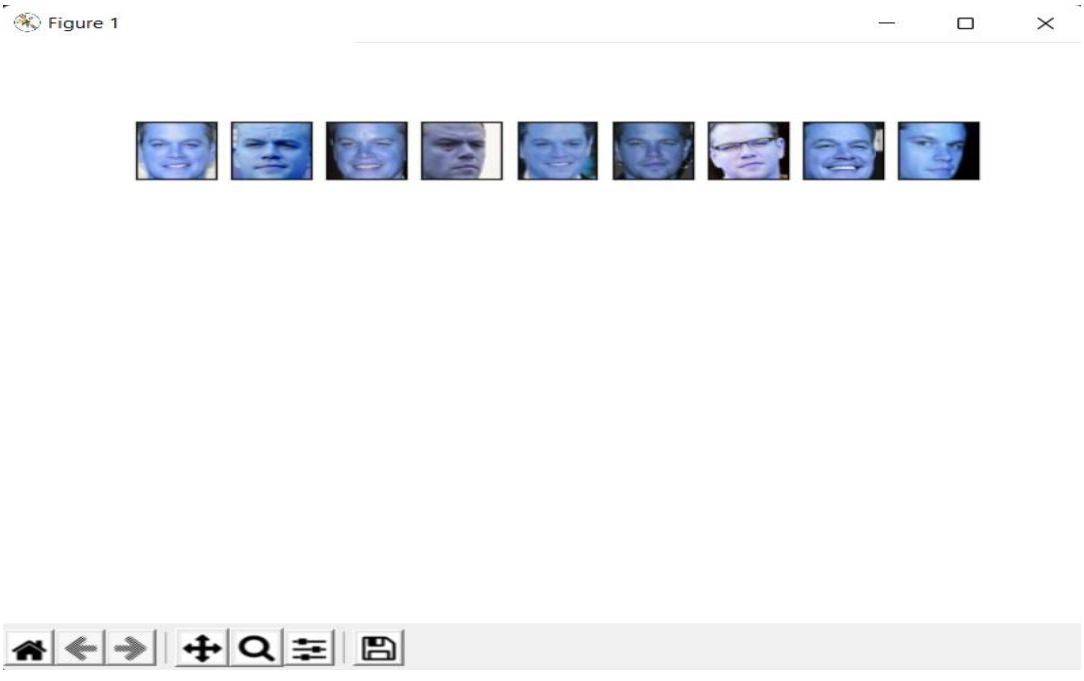


**Figure 3:** *Cluster 2*

(iv) Cluster 3:

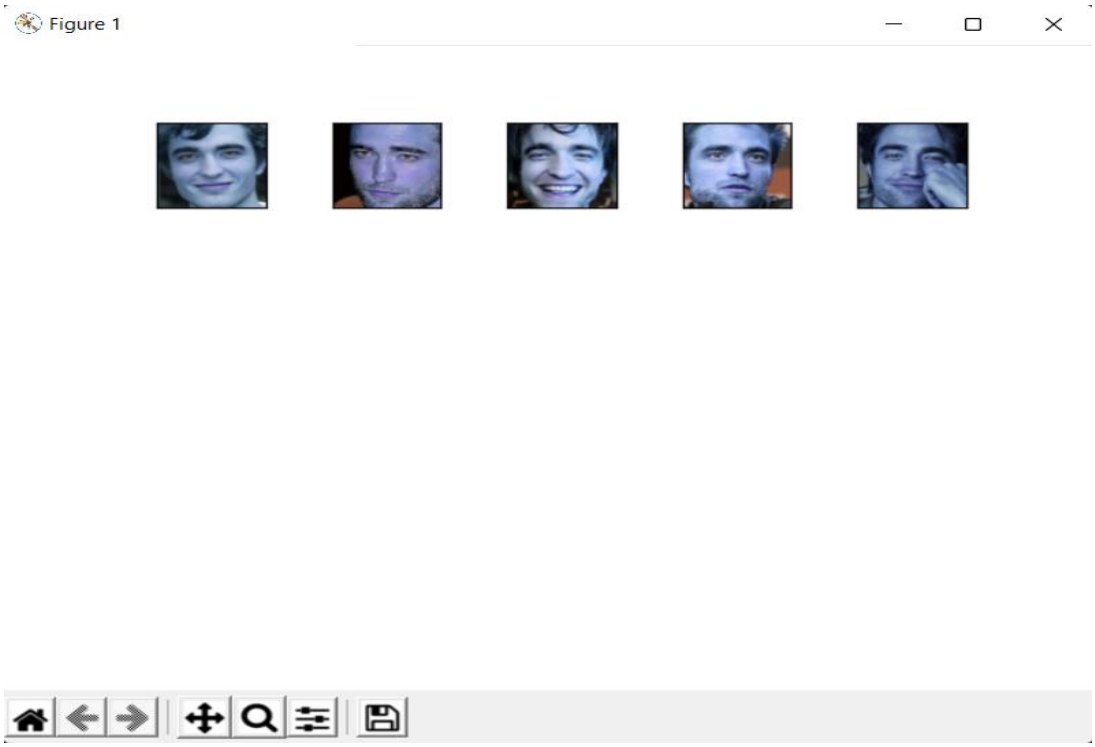


**Figure 4:** *Cluster 3*

(v) Cluster 4:

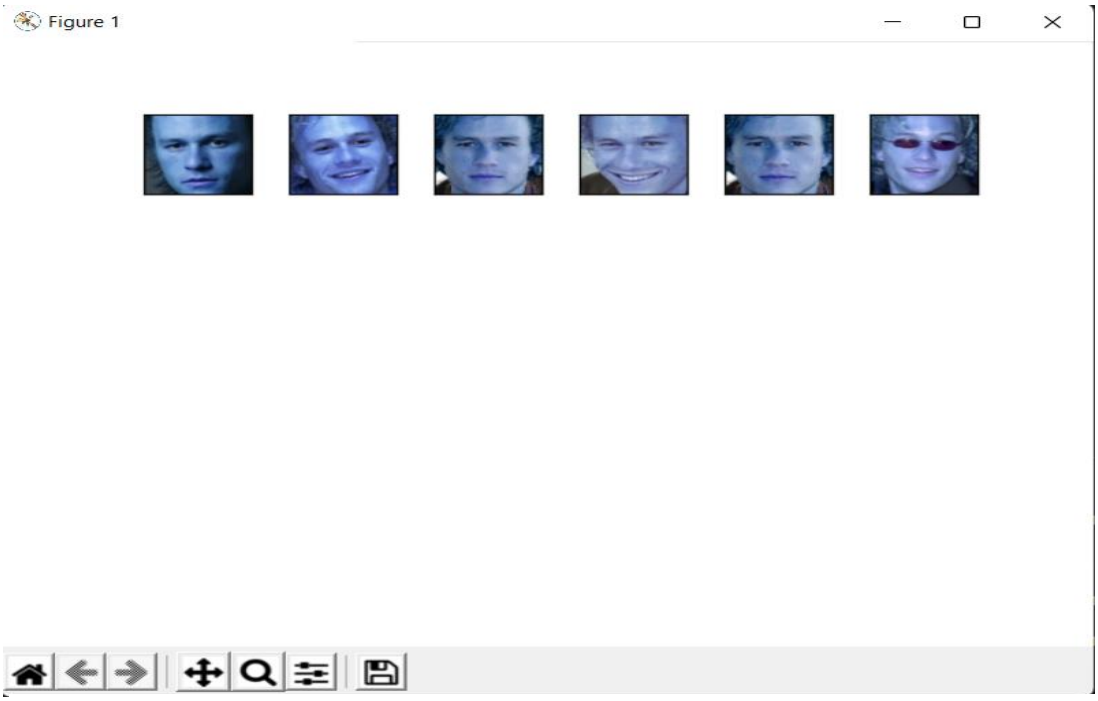


**Figure 5:** *Cluster 4*

- As we can see from the above plots, I’m successfully getting the clusters of the same faces in each plot. I’ve also manually checked from the clusters.json file that in each cluster the same faces are grouped together.
- For part B, as I was successfully able to implement face clustering with very good results, I’m using ‘Kmeans’ clustering for face clustering.

## REFERENCES


[1] https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html

[2] https://www.datasciencelearner.com/face-detection-recognition-python-hog-tutorial/

[3] https://www.codegrepper.com/code-examples/python/faceModel+%3D+%22opencv_face_detector_uint8.pb%22

[4] https://docs.opencv.org/4.x/d6/d0f/group__dnn.html#ga29f34df9376379a603acd8df581ac8d7

[5] There was lot of information in the 'CSE_473_573_Project_3.pdf' for the project provided on piazza by Professor.