Name: **Karan Bali**
UBID: **50381691**
UBID Name: **kbali**

Part 1: Face Detection in the Wild:

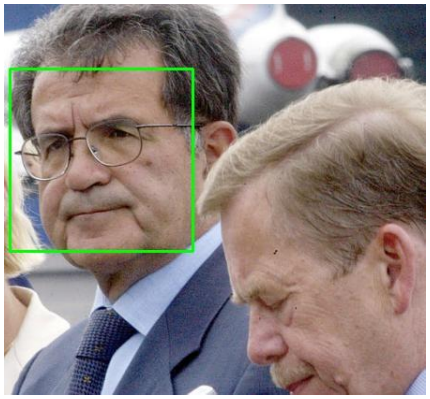
I've used an open-cv API of the haar cascade classifier(frontal face) to detect faces in an image.

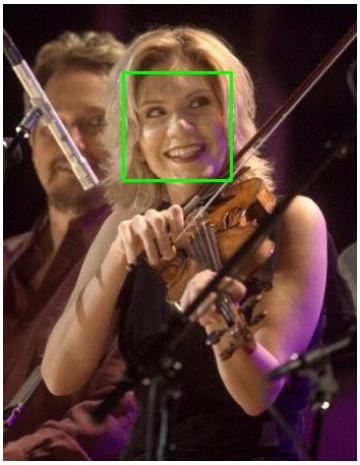
Part 2: Part A:

I've coded 'FaceDetection.py' to implement Part-A (as mentioned in the assignment).
I extract the filename passed as an argument. Calculate the number of images inside the folder (after sorting files). Thus, I sequentially read every image detect face using the classifier mentioned above. Later, I output the results of each detected face to a JSON file (in mentioned format) & saves it as "results.json". I've also coded the part to view the detected face (Although I've commented it out).

After that, you compute the FBeta score as mentioned in the assignment. I got a score of FBeta ("0.82") on testing it on "Validation Folder".

I'm pasting some of the examples from Part-A, down below.





Part 3: Part B:

I've coded 'FaceCluster.py' to implement part B (as mentioned in the assignment). I extract the filename (and 'K') passed as an argument. Calculate the number of images inside the folder (after sorting files). Thus, I sequentially read every image & repeated the same steps performed in Part-A. Later on, using the 'bbox' readings of detected faces & the "face_recognition" library I cropped the detected face & convert the same into a 128-dimensional encoding.

Then I used those encodings to implement the K-Means++ algorithm for clustering faces. K-Means++ algorithm is similar to normal K-Means, with one major difference in choosing the initial centroids to kickstart the K-means algorithm. Instead of randomly choosing the centroids, we choose the points as centroids that are farthest from each other.

Then we perform the same normal K-means algorithm, where we iterate through all data points a number of times. On each iteration, we calculate the distance between the points & centroids. Then we assign a class (of centroid) to each point. Thus, the point belongs to the class of nearest centroid. Finally, we update the centroids using the mean of all the points belonging to the same class of centroid. We repeat this process a given number of times. Finally, we get the optimal cluster of images with labels.

Finally, iterate through all clusters (& their images) to horizontally concatenate all images belonging to the same class (cluster). I save the concatenated images in the same directory ("cluster_i.jpg" with 'i' as a cluster number). I've also coded the part to view the concatenated cluster images (Although I've commented it out). I also output & save the required cluster information in the "cluster.json" (in the mentioned format).

I'm pasting the saved cluster images, down below.

cluster_0.jpg:

cluster_1.jpg:

cluster_2.jpg:

cluster_3.jpg:

cluster_4.jpg: