CSE 473 Project #3 Report

Name: Steve Glenn Joseph
UBITname: stevegle
Person #: 50218482

Description and Testing Environment

We are given a subset of the FDDB dataset consisting of about hundred images and our goal is to detect the faced contained in the images. We are allowed to use any API provided by OpenCV, So I have chosen a pre-trained cascade “haarcascade_frontalface_default.xml”. Which can be imported using:

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
```

Haar Cascade classifier is an effective object detection approach which was proposed by Paul Viola and Michael Jones in their paper, “Rapid Object Detection using a Boosted Cascade of Simple Features” in 2001. This is basically a machine learning based approach where a cascade function is trained from a lot of images both positive and negative. Based on the training it is then used to detect the objects in the other images. Here we use the pre-trained 'haarcascade_frontalface_default.xml' which contains the feature set to detect the ‘Frontal-face’.

We first Import the required modules and parse arguments :

```
import cv2
import os
import json
import argparse
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("P")
    args = parser.parse_args()
    return args
```

We then Initialize data:

```
json_list = []
dataDir = str(args.P)
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
scaleFactor = 1.1
minNeighbors = 1
FFlag = 1
images = []
```

Here we store out image file names, X,Y and Width and height for the bounding boxes in the `json_list` so we can dump it into the results.json file later. We use `dataDir` to store the directory of our test images. And `face_cascade` contains our `CascadeClassifier` with the pretrained XML file. We then initialize our parameters for the `detectMultiScale` function of the `CascadeClassifier` which are `scaleFactor` which is the parameter specifying how much the image size is reduced at each image scale and Here I have found that 1.1 has given me the best results. And `minNeighbors` which is the parameter specifying how many neighbors each candidate rectangle should have to retain it. This parameter will affect the quality of the detected faces. Here I have found that 1 works perfectly for my case. And we use `images` to store the images that I am reading.

Now we append the read test images to images:

```
for file in os.listdir(dataDir):
    file = file.lower()
    if file.endswith('.jpg'):
        images.append(dataDir + file)
```

Now we run our classifier :

```
if images:
    output = ""

    for image in images:

        img = cv2.imread(image)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray, scaleFactor,
minNeighbors)

        if (FFlag):

            for (x,y,w,h) in faces:
                element_1 = {"iname": image.replace('Test
Folder/images/',''), "bbox": [int(x), int(y), int(w), int(h)]}
                json_list.append(element_1)
```

Here, for all valid `images`, we first convert the image to greyscale. The reason for this is gray channel is easy to process and is computationally less intensive as it contains only 1-channel of black-white. We then run the `detectMultiScale` function on the greyscale images. And now this function returns 4 values — x-coordinate, y-coordinate, width(w) and height(h) of the detected feature of the face. We will now take these returned values and store them in `element_1` which we then append into the `json_list`.

We then dump the `json_list` to result.json:

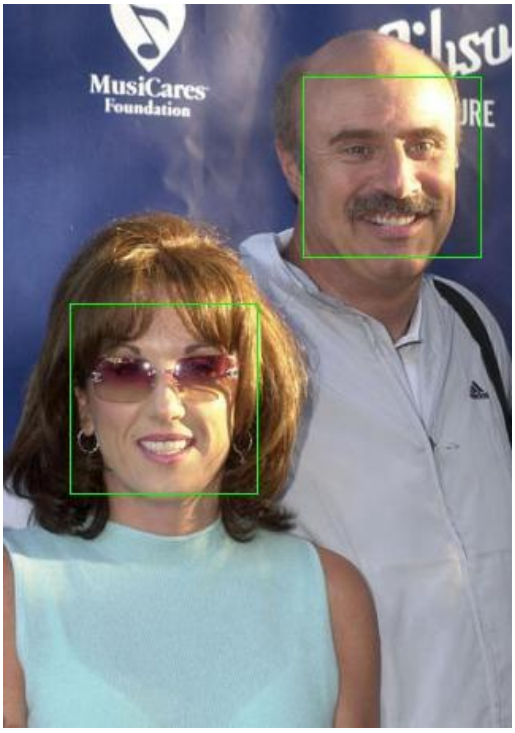
```
output_json = dataDir+"results.json"
#dump json_list to result.json
with open(output_json, 'w') as f:
    json.dump(json_list, f)
```

Results and Conclusion

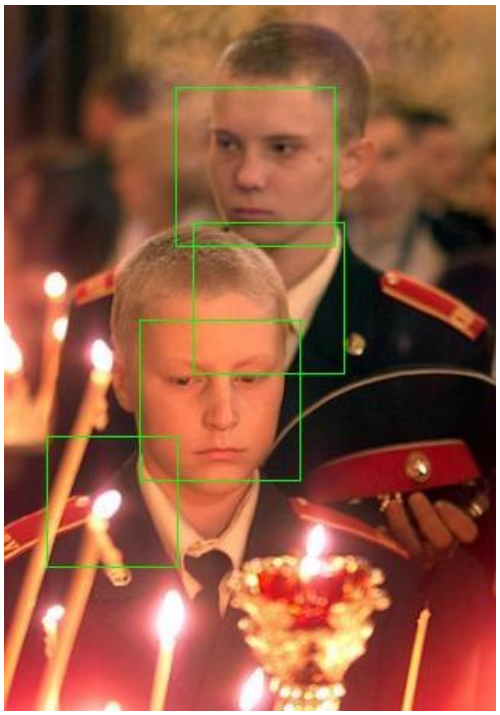
After running the model on the Validation images we get a F1 score of 0.67669:

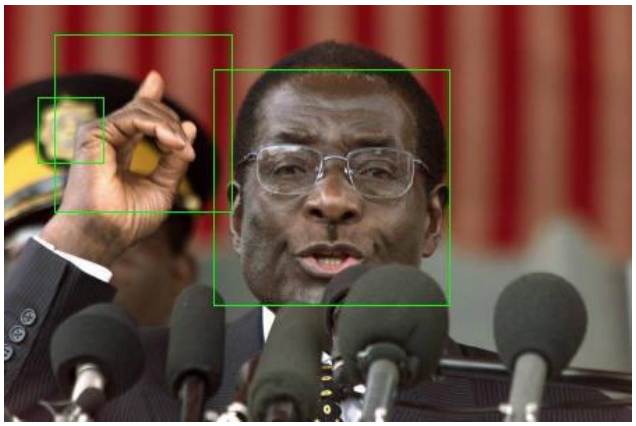
```
(base) C:\Users\steve>python D:\473\ComputeFBeta\ComputeFbeta.py
D:\473\ComputeFBeta\results.json D:\473\ComputeFBeta\ground-truth.json
0.6766917293233082
```

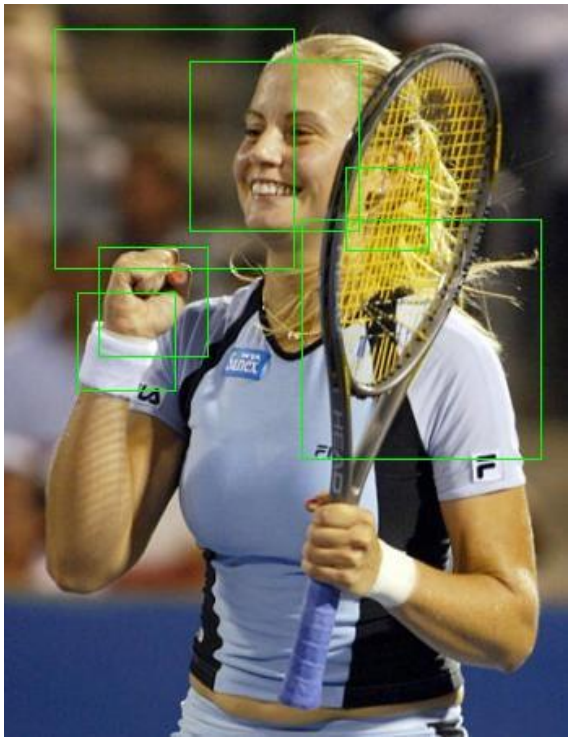
And we get annotated detection images (Sample of correctly detected images):



But we also face some issues when we try to detect faces in more complex scenes:







We can see that this implementation sometimes detects “faces” in areas that ever so slightly resemble a face and in some cases would omit faces as we see in the last picture. This can be possibly mitigated by tuning the parameters that we pass to the `detectMultiScale` function.