Lab 1 & 2 - Face Recognition

Classical Face Recognition

Our objective in the classical is to compare the following methods:

- Eigenfaces
- Fisherfaces
- LBP Histograms

In order to accomplish this task we will use the Caltech dataset, which consists of 450 images of faces from 31 different people.

Task 1. Download the Caltech dataset from the following link:
https://drive.google.com/file/d/1-d64ELOaDvf-z-PqIpK5FSb0okAewgZ-/view?usp=sharing
Unzip the archive and take a moment to get familiar with the dataset structure. We are particularly interested in the following files:

- *image_nnnn.jpg* - 450 images of faces,
- *caltech_labels.csv* - information on which class the face in the picture belongs to (i.e. first row – image_0001.jpg -> class 1, second row – image_0002.jpg -> class 1, etc.),
- *ImageData.mat* - information about the location of faces in the images (please note that faces in the images are relatively small compared to the whole picture).

On the course e-learning platform you have the solution template *face_classic.py* - download and open it. As you can see, the most complicated part of this exercise is data pre-processing as the all three mentioned methods are provided in OpenCV[1] - the proper objects are created in *FRs* dictionary. We need three components:

1. Labels - function *read_labels*
2. Region Of Interest (ROI) - function *read_ROIs*
3. Paths to images - function *read_img_paths*

Task 2. Implement the *read_labels* function, which gets the path to the CSV file as a string and returns the dictionary with int keys and int values. We will use this function to read the *caltech_labels.csv* and construct the mapping image – label. Function should read a CSV file (you can use csv module) row-by-row and add records to the dictionary. Each record should

[1] They are a part of the so-called *contrib* package, which can be installed via `pip install opencv-contrib-python`. You do not have to do it in the laboratory - it is already installed as all other needed packages.

have a structure `image_number: label`, where *image_number* is the same as row in CSV file (counting starts from 1) and *label* is the content of this row in the file.

Task 3. Implement the *read_ROIs* function, which gets the path to the MAT file as a string and returns the NumPy's ndarray. We will use this function to read the *ImageData.mat* file and get the data about faces location in images - our Regions of Interest. To load data from a file you can use the SciPy's loadmat function. It returns the dictionary with some records. Among them search for the key `"SubDir_Data"` - their value is our wanted array. Please note that it has 2 dimensions and size 8 x 450 - read the dataset's README file to understand why it has such a structure.

Task 4. Implement the *read_img_paths* function, which gets the path to directory with images and returns the list of paths to files. We will use this function to get the list of paths to all images in the Caltech dataset. Therefore you must ensure that each path leads to the existing image's file, which ends with `".jpg"`. To implement this in a clean and short way, you can use Python's os module (especially listdir) with a path submodule. To check if the file has the appropriate type, the str.endswith could be helpful. At the end, each item in the return list should be a string similar to `"./caltech/image_0422.jpg"`.

After completing the above task our program will be capable of reading all necessary data from the Caltech dataset. Now we need to use this data in an appropriate way. First of all, please note that each face in the images can have a different size and location. Moreover, images are in RGB format (colorful), while face recognition methods accept the grayscale images and all of them should have the same size. Therefore we need proper image preprocessing.

Task 5. Implement the *img_preprocessing* function, which gets a path to the image and coordinates of two vertices (top left and bottom right) of the face's bounding box. It returns the pre-processed image of size *IMAGE_WIDTH* x *IMAGE_HEIGHT*. It can be achieved in 4 steps:

1. Load the image from the path - cv2.imread.
2. Convert to the grayscale - cv2.cvtColor.
3. Crop to ROI - remember that X axis is along image width (i.e. ndarray axis 1) and Y axis is along image height (i.e. ndarray axis 0).
4. Resize to proper size (*IMAGE_WIDTH* x *IMAGE_HEIGHT*) - cv2.resize.

Now we have all auxiliary functions ready and can finally deal a little more with biometrics :) First of all, since all face recognition algorithms have to be trained on some portion of data and then tested, we need to divide the dataset into two parts (approximately 75% to train subset, 25% to test). Moreover, we do not want to use faces with too few samples, as this would not be reliable.

Task 6. Create train and test subsets from the Caltech dataset based on previously defined auxiliary functions. It should be implemented in the "main" part of the script. You can do it by the following steps, iterating through all the images in filenames list:

1. Get image index from its file name - it is enough to convert a specific part of a name from str to int, str.split function could be helpful.
2. Based on the index, get the image label from labels' dictionary.
3. Check how many images with the same face are in the Caltech dataset - you can iterate through the labels dictionary and count the number of labels the same as the label obtained in the previous step.
4. If the number of faces' samples are too low (below *N_FACES_THRESHOLD*), then continue to the next image.
5. Otherwise, use the *img_preprocessing* function (prepared before) to preprocess the input image. Remember about proper arguments assignment, especially top left and bottom right points, based on the ndarray returned by *read_ROIs* and information contained in the README file. For brevity, you can use predefined variables like *X_TOP_LEFT_IDX* etc. If you are interested in what the images look like after preprocessing, you can use cv2.imshow with cv2.waitKey to display them.
6. After preprocessing, randomly append this sample to the training or test subset. Append also the proper label, which you determined before.

After preprocessing and splitting the samples, we are finally ready to test all three methods. Objects representing each of them are already declared in the *FRs* dictionary, so everything we have to do, is to use them wisely :)

Task 7. Train models of each face recognition method and then test its performance. Each object from the *FRs* dictionary has two methods: train and predict. First of them can be used on the entire train set at once, but it demands conversion of the list of labels to the numpy.array. Predict method must be invoked iteratively for each test image separately - do it in the simple *for* loop and count how many test samples are predicted correctly. Based on that, compute and print the model accuracy as "good_predictions/all_test_samples".

That's all - now you can verify which method gives the best results. You can also change the random seed to observe how this affects the final accuracy via dataset splitting.