# Benchmarking the LBPH Face Recognition Algorithm with OpenCV and Python

Johannes Kinzig[1], Christian von Harscher[2]
Faculty of Computer Science and Enginering
Frankfurt University of Applied Sciences
Nibelungenplatz 1
60318 Frankfurt am Main, Germany
Email: [1]kinzig@stud.fra-uas.de, [2]harscher@stud.fra-uas.de

**Abstract**

In this paper, we present an benchmark of the Local Binary Patterns Histogram (LBPH) algorithm in combination with OpenCV and Python. We wanted to show that these techniques can be used for a real-time application. The main part of this work will describe the architecture as well as the implementation of the detection and recognition application. A cascade classifier with haar-like features will be used for the detections phase. Of course the recoginition will be implemented with the LBPH algorithm. The benchmarks will be executed on different machines for generating representative results.

**Index Terms**

Real-time Systems, Face Detection, Face Recognition, Benchmark

## I. Introduction

Face detection and face recognition can be used in various real world application fields. This could be face tracking during a live broadcast of a sport event or person identification at the airport border control. Another field of application could be criminal prosecution in public places using the image data of surveillance cameras. Both examples need a fast detection and recognition algorithm because the results of detecting and recognizing a face are important for further processing. For instance, when performing a face tracking of a specific person in a video stream the operations for detecting and recognizing the face have to be as fast as the picture display rate. Otherwise the face tracking could not be performed. In this paper a benchmarking proposal of OpenCV's face detection algorithm and the face recognition algorithm (using LPBH recognizer, cf. I-B) is described and later on performed. The benchmark should give information about required hardware and wether the face detector and LBPH face recognizer can be used in realtime applications.

### A. Face Detection

Face detection describes the process of locating one or many human-like faces in a image sequence. For a fast detection of faces the haar-like feature algorithm by Viola and Jones can be used [2].

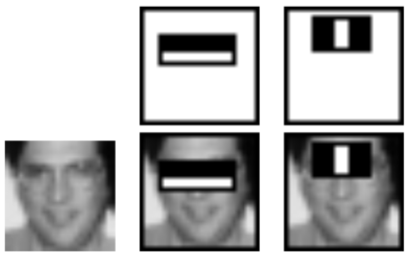

Fig. 1. Haar-like features [2, Fig. 3]

The left feature in the upper row of Fig. 1 is showing an edge. This feature can be used for detecting the forehead position of a face. The feature on the right is used for detecting a line, mostly the position of the nose of a face can be determined. The second row is showing a sample image with the applied features. A combination of these and additional features can be used for a rapid detection with a high detection rate of 95% described in [2, sect. 3.2].

The rapid detection is done by using a cascade classifier (fig. 2). The image is segmented in sub-windows and only a few features are checked at the beginning. Only if the current feature is detected, further features will be checked. This reduces the number of features that must be checked per sub-window. If the given feature set is not detected inside the sub-window then the sub-window will be rejected and no further processing will be done. Due to this processing of the feature detection, the performance of the algorithm can be increased because not all features must be checked for each sub-window.

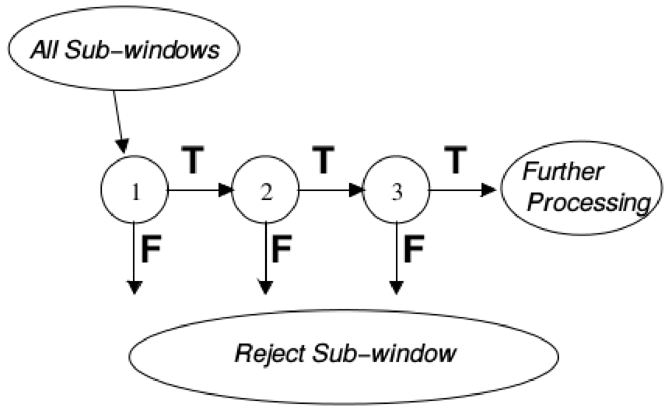

Fig. 2. Detection cascade [2, Fig. 4]
T=true; F=false

## B. Face Recognition

The next step after locating a face inside of an image is the recognition of it. The recognition is the process of identifying a face inside of a set of previously learned faces. For this purpose different algorithms like Eigenfaces, Fisherfaces or Local Binary Patterns Histograms (LBPH) can be used. In this paper the LBPH will be covered.

In fig. 3 an example image is given in *a*. This image is already screened in sections. Our application for benchmarking will display a colored image sequence but analyze only the desaturated grayscale image. After this each section will be weighted with a grayscale value, representing the dissimilarity to the neighbor sections. In paper [3, sect. 4] the black sections indicate a value of 0.0 and the white sections are weighted with a value of 4.0. The analyzed image is shown in *b* of fig. 3. After this dissimilarity analysis the weighted sections represent a shape and texture pattern that can be compared to the existing samples of a person.

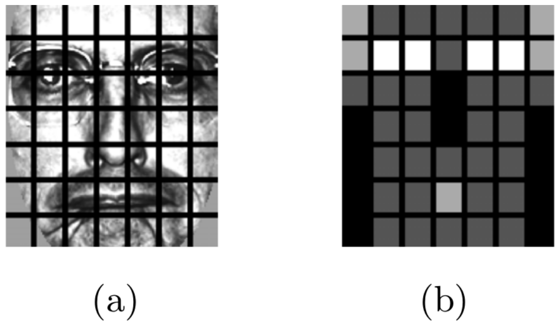

Fig. 3. LBPH analysis [3, Fig. 5]

## II. Algorithm Implementation

Before a benchmarking of the OpenCV algorithms could be performed, a software had to be developed which implements the face detection and recognition algorithm. Implementing the recognition algorithm requires to also implement OpenCV's face trainer.

This lead to a complete software which had to be developed with the following features:

1) Read in the video stream from a camera which is connected to the computer
2) Detect a face in the current frame of the image sequence
3) Allow to store the face and link a label to it (label = face name)
4) Recognize the stored face in another video stream and show the correct label

A graphical representation of the implemented features and software architecture can be seen in fig. 4.

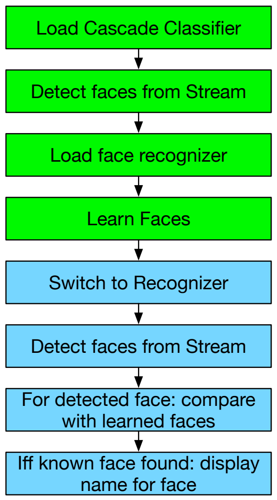

Fig. 4. Software Architecture

### A. Architecture

The overall system architecture is shown as a use case diagram in fig. 5. It shows the detection and recognition process. Subject A and B are positioned in front of the webcam. The detector will then load the cascade classifier for detecting the haar-like features of the image sequence. After detecting the faces of Subject A and B inside of an image the detector will forward the coordinates of the detected faces to the recognizer. Then the recognizer will loop through the learned samples of each subject and tries to recognize the subject. If the recognition process was successful the recognizer will highlight the face of a subject and display the name over the previously drawn rectangle. For this whole process the face samples need to be learned first.

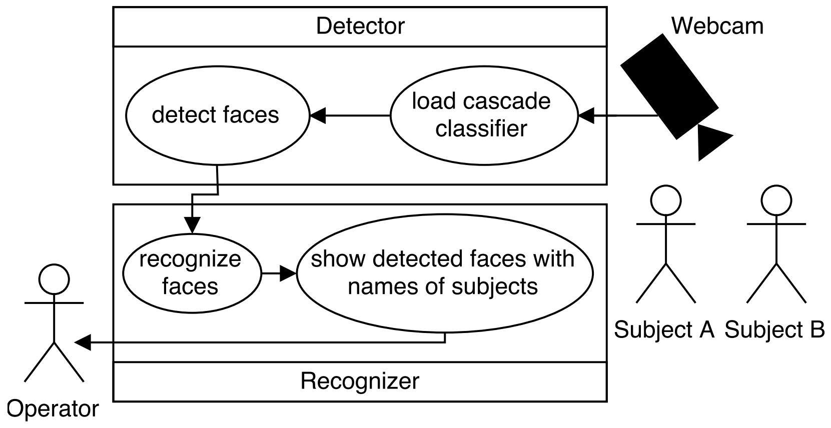

Fig. 5. Use Case Diagram

In the learning phase the face detector is used for detecting the faces positions. After that the face can then be named by the operator and saved by the system. A typical set of samples can be found in fig. 6. The first row represents one person and the second row represents another person. The system will not save the images as files but interpret the features of each face and saves them into main memory.

Fig. 6. Example samples as images

### B. Implementation details regarding execution speed

The complete software was written in Python and executed in a Python V.2.7.11 environment. The OpenCV algorithms are written in C++ and were called using the Python application. The timing overhead which occurs when calling the libraries from within the Python application can be neglect, additionally all the algorithms which are benchmarked are written in C++ and were compiled to binaries for the specific system. Basically this means that the algorithms are executed directly and abstain from a virtual machine layer like the Python VM or Java VM. This leads us to the assumption that the results are reliable.

Fig. 7 shows the face recognizer window. The video stream was captured from the camera and the face recognizer compares each frame with the faces contained in the training set. To prove wether the recognition was done correctly a rectangle is drawn around every detected face and labeled with the stored label (label = name for the face).

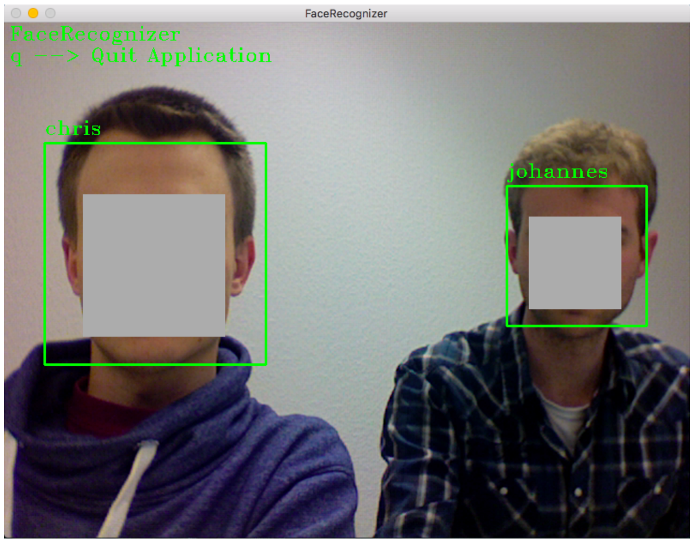

Fig. 7. Face recognizer application developed to benchmark the face detection and recognition process

## C. *Face Detection*

Generally developing applications which detect a face is very easy when using OpenCV. The detection itself is just one line of code, the ”configuration” of the face detector is also quite lightweight. In line two the file with the haar-cascades is loaded into the ramdisk. This is an XML based file which holds pre-trained information necessary to detect a human face. Loading another cascade classifier leads to detecting other objects, for instance a dogs’s face. Line five just connects to the camera which is connected to the computer (using USB). For detecting faces and to let it look like a video-stream in line seven an endless loop is started. In line nine the current frame is taken from the video stream and is stored as ”frame”. The the other value which is stored as ”ret” is a boolean value which is ”True” if the frame was captured successfully, otherwise it is ”False”. Because the LBPH algorithm can only take gray frames for processing in line 12 the picture is converted to gray (cv2.COLOR_BGR2GRAY). In line 15 the face detection is performed. The method takes the gray frame as an input (including some other parameters) and returns detected faces as the output, these are stored in ”faces” (cf. [1]).

```
1   # loading the cascade classifier into ramdisk
2   self.faceCascade = cv2.CascadeClassifier(cascPath)
3
4   # start capturing the stream from the camera
5   self.videoStream = cv2.VideoCapture(self.videodevice)
6
7   while True:
8            # Capture frame-by-frame
9            self.ret, self.frame = self.videoStream.read()
10
11           # convert image to gray, because recognition is done in gray
12           self.grayframe = cv2.cvtColor(self.frame, cv2.COLOR_BGR2GRAY)
13
14           # holds the face data
15           self.faces = self.faceCascade.detectMultiScale(self.grayframe, ... ,
                ↪ flags=cv2.cv.CV_HAAR_SCALE_IMAGE)
```

## D. Face Recognition

The face recognition process is similar to the detection process. The most important task is to train the recognizer correctly. In line two an instance from the LBPH face recognizer is generated, in line five its method *train* is called. This method trains the face recognizer with the face - label pairs. As arguments it takes the faces (features) which were detected by the face detector (cf. II-C) and the labels for the faces. The labels are passed as an numpy array. Numpy is an extension module for Python which allows fast computing of large matrices and large arrays. The numpy array is internally organized as an array in C++. As described in section II-B OpenCV is written in C++ and therefore the face trainer expects the labels as an supported data structure. This requires to pass the labels as an numpy array.

```
1  # For face recognition the LBPHF recognizer will be used
2  self.recognizer = cv2.createLBPHFaceRecognizer()
3
4  # now train the recognizer, pass the faces and the corresponding labels
5  self.recognizer.train(features, numpy.array(labels))
```

The face recognition itself again starts as the detection. Inside the endless loop frame after frame is captured from the video stream. Then again the face detector is started and stores detected faces into an array. Afterwards the face recognizer is called for every face which was detected by the face detector (face was stored inside array). This saves time because the face recognizer is only started when a face was detected, otherwise the recognizer would get a ”faceless” frame as an input.

```
while True:
    # holds the face data
    self.faces = self.faceCascade.detectMultiScale(self.grayframe, ... , flags=cv2.
        ↪ cv.CV_HAAR_SCALE_IMAGE)

    # for every face detected above
    for (x, y, w, h) in self.faces:

        # run the face recognizer
        self.predicted, self.conf = self.recognizer.predict(self.grayframe[y: y + h,
            ↪  x: x + w])
```

The face recognizer begins recognizing a face when its *predict* method is called. The predict method takes a frame as input. In this scenario this frame definitely contains a face because otherwise the for loop would not run (because the array where to iterate over would be empty).

The recognizer then delivers two output values *predicted* and *conf*. *Predicted* holds the label of the face which was recognized and *conf* contains the according confidence of the recognition. The higher *conf* is the less is the recognizer confident about the recognition. A perfect recognition is obvious when the recognizer returns 0.0 for the *conf* value (cf. [1]).

## E. Benchmarking

As already described in section II-D the face detection and also the face recognition is basically performed with just one command. Therefore the benchmarking consists of measuring the execution time of the specific command which starts the detection and recognition.

Measuring the execution time can programmatically be performed in simple steps:

1) Get current time (by asking the OS) and store it as start time
2) Run the function or method whose execution time has to be determined
3) Get the current time and subtract the previously stored start time

For this specific task a Python class was written which does the time measurement. This has the advantage that in the main application only two methods have to be called: *startBenchmark()* for starting the measurement and *stopBenchmark()* for stopping the measurement and returning the measured values. The *stopBenchmark()* method returns two double values, representing the execution time in seconds for the wall clock time and CPU time. This described class also takes care for writing the determined values to a CSV file.

Measuring the time with Python is quite fast forward, the *time* module has to be imported. With the command *time.clock()* the current CPU time is returned as a floating point number expressed in seconds on Unix. With the command *time.time()* the current wall clock time is returned as a floating point number in seconds since the start of the epoch in UTC (cf. [4])

The following listing shows a sample method to measure time in Python with the class which was developed to benchmark the OpenCV algorithms (same is described in [5, p 191, 192]):

```
# TimeMeasurement instance with name "Measurement1"
measurement1 = TimeMeter("Measurement1")

# measure cpu and wall clock time
measurement1.startBenchmark()
time.sleep(1) # method or function call which timing has to be determined
wallclock, cpuclock = measurement1.stopBenchmark()
measurement1.writeToFile(str(wallclock) + ",\t" + str(cpuclock) + "\n")
```

The CPU execution time describes how long the algorithm was processed by the CPU, apparently how much CPU cycles were needed for the execution. The wall clock time represents the real execution time of this algorithm. This means that all other processes which are running on the system and are inserted in the execution because of a higher priority (or any other scheduling policy) are respected for the time measurement. This is the reason why the real time is alway higher than the CPU time.

Determining the CPU time when benchmarking an algorithm is usually the first way to go because this shows the behavior independent from the other processes. This allows to compare the algorithm performance among different CPUs with different speed. The CPU time also offers valuable clues to the multi core or multi CPU behavior of an algorithm. If the wall clock time is lower than the CPU time, it is assumed that the algorithm is supporting multiple cores. The explanation is that for the same amount of time more CPU cycles are available than on a system with less CPU cores.

Fig. 8 shows that the execution measured using the wall clock time takes less time than the CPU time. This shows that the face detection algorithm provided by OpenCV supports multiple cores and increases in performance with the number of CPU cores.

The wall clock time is interesting when getting information about how fluent an application is running on a specific system because the real time is returned. This information is also important for this kind of benchmark because the output of the software execution is visible. For instance when doing a live broadcast of an sports event and the stage direction wants the camera to follow a hockey player the real execution time (wall clock time) of detecting and recognizing the face has to be known beforehand because otherwise the complete system is not working. In case the player runs faster out of the frame than the algorithm can detect and recognize this face, the camera is not able to follow the face anymore.

This is the reason why this paper describes the approach for taking both values into account, the wall clock time and the CPU time.

## III. Performing the benchmark

### A. Preconditions

When performing the benchmark several important preconditions had to be ensured:

- Illumination conditions have to be the same for training the recognizer and for running the recognizer
- Training the recognizer was done from live camera stream
- Recognition of faces should be done in the same environment as training the recognizer

The light and illumination conditions should be the same for training the face recognizer and for running the recognizer. Occurring shadows in the faces because of different light and illumination conditions, especially when the light beam comes from another direction may lead to a bad recognition rate or to a faulty recognition. This had to be avoided because additionally measuring the recognition rate would have increased the effort drastically. This lead to the idea to learn the faces directly from within the situation in where the faces should be recognized in the second step. This ensured a comparatively high recognition rate which allowed to concentrate on the time measurement.

While doing the benchmarking all faces were recognized correctly and the illumination condition was the same throughout all tests.

### B. Benchmarking on devices

To get an impression about the execution time of the face detector and recognizer the idea was to run the same tests on several devices. This shows wether the algorithms support multi core systems (”increasing number of cores, the faster the execution becomes”) and offers valuable clues about the systems which are needed to use the algorithm appropriately for different fields of application.

As described in section I the benchmark should give information wether the OpenCV algorithms are suited for real time applications, three different devices were chosen (table I). One older computer, the MacBook (Modell late 2009), a newer Mac (MacBook Air) with average performance and an Lenovo ThinkPad with comparatively high performance.

TABLE I
MACHINES USED FOR BENCHMARKING

| Machine | Operating System | Processor | Main Memory |
|---|---|---|---|
| MacBook | OS X 10.11.3 64-bit | Intel Core 2 Duo @2 GHz | 4 GB Ram |
| MacBook Air | OS X 10.11.3 64-bit | Intel Core i5 @1.4 GHz | 4GB Ram |
| ThinkPad | Linux Mint 17.3 64-bit | Intel Core i5-4200M @2.5 GHz | 7.5GB Ram |

## IV. Results

The benchmarking of the application was done separately for the detection as well as for the recognition phase. For all recognizer test cases in table II the camera resolution was specified to 800*600 pixels and the samples of two persons were used (fig. 7). The same camera resolution was used for the face detector.

TABLE II
Recognizer test cases

| Test case no. | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Machine | MacBook | MacBook | MacBook | ThinkPad | ThinkPad | MacBook Air | MacBook Air |
| Samples per person | 2 | 7 | 12 | 2 | 7 | 2 | 7 |
| AVG(wallclock time) [sec] | 0.00838 | 0.01155 | 0.01170 | 0.00321 | 0.00468 | 0.00637 | 0.00442 |
| AVG(CPU time) [sec] | 0.00945 | 0.01252 | 0.01338 | 0.00500 | 0.00677 | 0.00653 | 0.00466 |

The benchmark of the detection phase in fig. 8 shows that the ThinkPad is the fastest of the machines followed by the MacBook Air and MacBook.

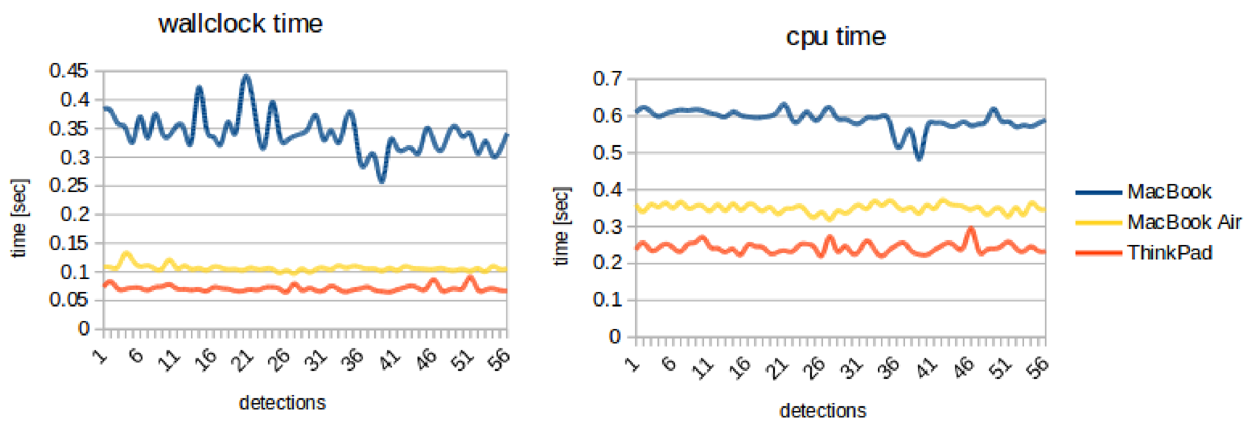

Fig. 8. Face detector benchmark

As described in section *Local Binary Patterns Histograms* of [1] it is necessary to use at least 8±1 samples per person for a high recognition rate. The same applies for the speed of the used LBPH algorithm. The comparison between 2, 7 and 12 samples is made in fig. 9. The average wallclock time of these three test cases in table II imply that there is a significant difference between test case 1 (2 samples) and 2 (7 samples). On the other hand there is no significant difference between the average wallclock time of test case 2 (7 samples) and 3 (12 samples). So slightly increasing the number of samples from 7 to 12 does not affect the speed of the used LBPH algorithm.

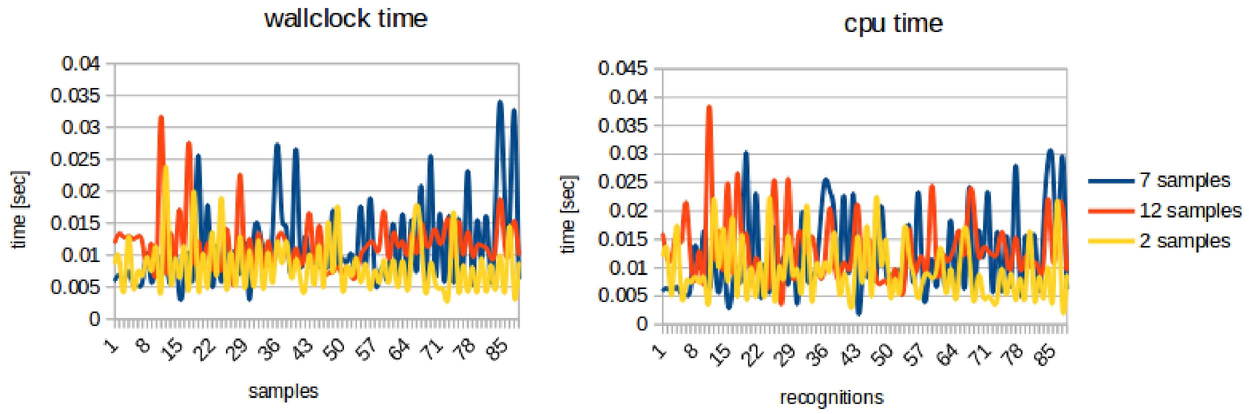

Fig. 9. Face recognizer benchmark with different number of samples on MacBook (test cases: 1, 2, 3)

For comparing the runtimes with different number of samples per person (2 and 7 samples) the benchmarks were visualized in fig. 10 and 11. The execution of the algorithm does not necessarily take longer but in the most cases it does because of the increased number of samples. The MacBook Air shows a low-peak in fig. 10 whereas the ThinkPad performes constantly. The MacBook as the slowest execution time. The anomaly described in V pt. *2)*, could be caused by other processes running either on the ThinkPad during the test case 5 or on the MacBook Air during test case 6.

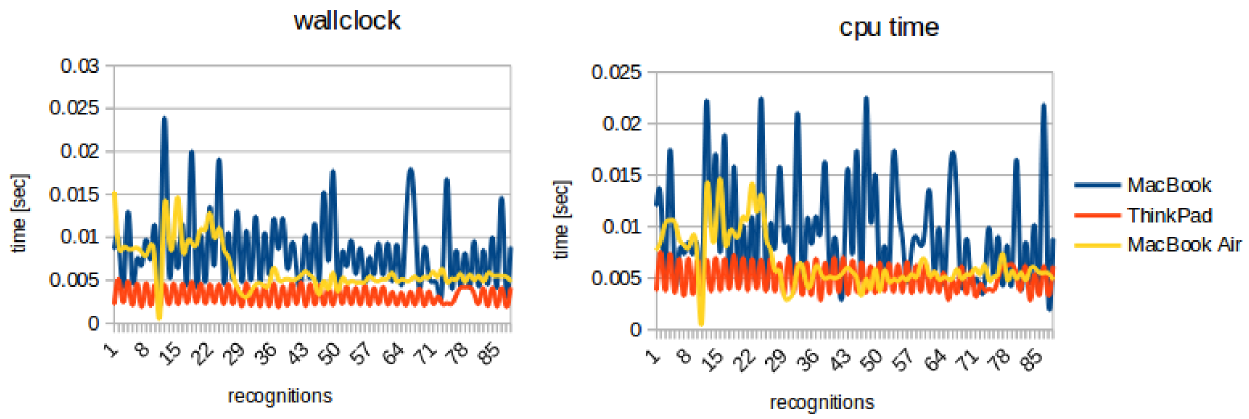

Fig. 10. Face recognizer benchmark, with 2 samples on each device (test cases: 1, 4, 6)

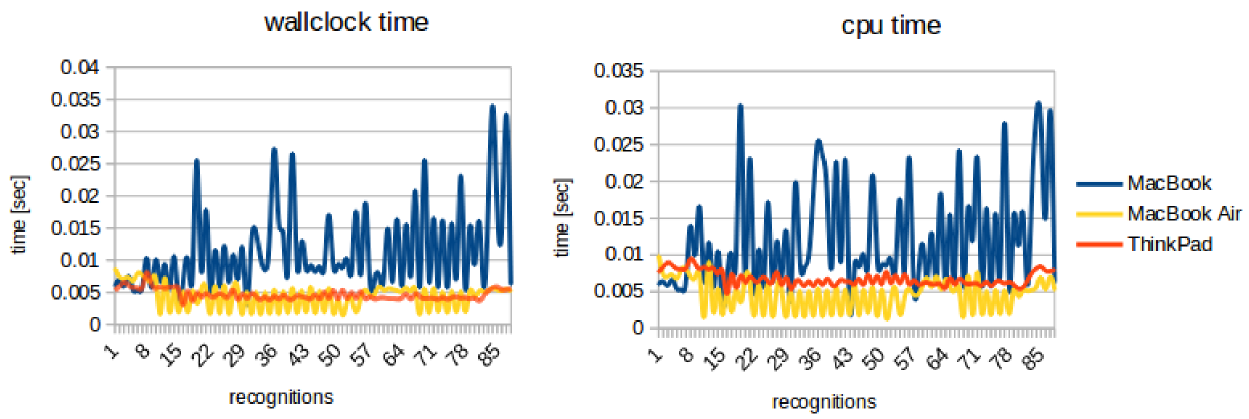

Fig. 11. Face recognizer benchmark, with 7 samples on each device (test cases: 2, 5, 7)

## V. Discussion

Section IV shows that both, the face detector and the face recognizer are suited for real time applications; this even on older systems such as the MacBook. When doing the benchmarks, on all three systems the video stream was displayed fluently, there were no noticeable delays in the stream. This is valid for the detection and also for the recognition of the faces. The proof for the detector can be seen in fig. 8 where the average time taken is about 0.08 seconds (wall clock) for the fastest system. Table II shows the proof for the face recognizer where the average time taken is about 0.00468 seconds (wall clock) for the Lenovo ThinkPad.

When interpreting the results, additional conspicuousness was recognized. Further explications to the following circumstances will not be given because this is not part of the actual work and may be researched in a following project.

1) When looking at fig. 11 and 8 it becomes obvious that the detection itself takes longer than the recognition.
2) When comparing fig. 10 and 11 it becomes obvious that the MacBook Air is performing faster than the ThinkPad when the training set consists of seven samples. When the training set consists of two samples, the ThinkPad is faster.

## References

[1] OPENCV DEV TEAM, Face Recognition with OpenCV, Mar 09, 2016 - accessed: 09.03.2016 *http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html*

[2] VIOLA, JONES: Rapid Object Detection using a Boosted Cascade of Simple Features, 2001 *https://www.cs.cmu.edu/ efros/courses/LBMV07/Papers/viola-cvpr-01.pdf*

[3] AHONEN, HADID, et al., Face Recognition with Local Binary Patterns, ECCV 2004 *http://uran.donetsk.ua/ masters/2011/frt/dyrul/library/article8.pdf*

[4] The Python 2.7.11 Standard Library: Generic Operating System Services *https://docs.python.org/2/library/time.html*

[5] David M. Beazley, Python Essential Reference, EAN: 9780768687026, Editor: Pearson Education