A Study on Face Identification for Cats

Sara K., **Giacomo P.** and **Davide S.**

Sapienza University of Rome, Italy

1 Introduction

For this project we decided to try something more challenging and original than usual which, as far as our knowledge concerns, no one ever tried to approach before: instead of working on human beings as it is common to do for a recognition task involving a biometric system, we chose to study the possibility to perform a recognition operation having **cats** as subjects.

Our interest was to study whether the same algorithms that are usually employed for the recognition of human faces (*Eigenfaces*, *Fisherfaces*, *LBP*) could actually also be used to identify a different kind of living beings: animals. We chose to work with **cats** both due to the presence in *OpenCV* of some already trained face detectors for that kind of pets on the one hand and because we had a few acquaintances owning one or more of them on the other hand, the latter point also being crucial as we had to build our own *data set*.

Starting form section 2, we describe the way we structured our project, the technologies we decided to employ and we also give a brief description of the roles covered by each one of the files we developed. In section 3 we give details about our *data set*, the way we created it and the extraction of the faces from the pictures. Following, in section 4 we illustrate the techniques, such as the algorithms and possible improvements, we used for the recognition. Then in section 5 we explain the way we evaluated the performances of our system. In section 6 we show and discuss the results we obtained through the evaluation. Finally, in section 7 we draw the conclusions and discuss possible future improvements.

2 Project Structure

The project was mainly developed using the *OpenCV* library and implemented using *Python 3* thanks to *OpenCV-Python*: a library of Python bindings offering developers the possibility to write their code in a simpler and more intuitive programming language, while taking full advantage of the computational speed and efficiency of *OpenCV*'s C++ API.

2.1 External Libraries

Our project involved the use of a number of external libraries in order to achieve our goals, we provide a list of the most important ones alongside the reasons why we needed them:

- **OpenCV**: on the one hand to perform any kind of operations on images, such as reading, manipulating and saving them, on the other hand to carry out the face detection and recognition tasks.
- **numpy**: needed by *OpenCV-Python* to work properly and to facilitate or speed up some other tasks.
- **matplotlib**: to automatically plot any kind of statistic on graphs and to display them.
- **PIL**: to perform some operations on images which were not feasible or more complex to carry out using *OpenCV*.

2.2 Code Organization

We now offer a brief overview of the files in which we organized our code and the tasks they cover.

2.2.1 Detector.py

This file contains the code we developed to perform the first phase of our work: creating the *data set* to be used for the recognition operations. The main function here is `detect_cat_face`, which uses *OpenCV*'s detection models in order to identify the faces of our subjects from the raw pictures, to crop them and save them in the proper *data set* folder. The function also tries to detect the subject's eyes, which are needed to attempt to align their faces with respect to the line connecting the eyes, which is useful for the recognition phase in order to capture as much detail as possible from each face.
The other minor functions, such as those used to align faces, were either written by us or taken from the *OpenCV* face recognition tutorial and slightly modified by us.

2.2.2 Recognizer.py

This file contains the code needed to train *OpenCV*'s face recognizers using our own *data set* and to perform an identity prediction with respect to the provided probe image. The two main functions here are: `train_recongizer` and `predict`.
The first one takes care of "training" one particular model using the provided csv file containing the list of images to be used for that purpose, which in our case also corresponds to the gallery enrollment phase, due to the way *OpenCV*'s face recognizers work.
The latter function is tasked with testing a probe image against the gallery (the trained model) in order to carry out the recognition operation. The `identification` parameter allows to switch between an **identification** operation, which returns the whole list of templates stored in the gallery alongside the distance score with respect to the probe, and a **verification** operation where only the template with the lowest distance score is returned. The core of both the functions is inspired by the C++ implementation that can be found in the *OpenCV* face recognition tutorial, which was useful for us to understand the way the library works.
The file also contains a series of utility functions to run quick tests and saving/loading trained models (all of them developed by us), alongside the `norm_0_255` function, which we translated from the C++ implementation that can be found in the tutorial.

2.2.3 Recognition_Tests.py

This file contains the functions needed to evaluate the performances of our system.
All the code that can be found in this file was entirely written by us following the description of the algorithms we studied during the course.
The `k_fold_cross_validation` function is tasked with generating the *k fold cross validation* subsets to be used to evaluate our system.
The `compute_distance_matrix` function deals with the creation of an $\text{All}_{\text{PROBE}}$-against-$\text{All}_{\text{GALLERY}}$ distance matrix, needed to compute the rates to measure the performances of the system.
The `evaluate_avg_performances` function takes the result of `k_fold_cross_validation` and computes the average rates produced by the `evaluate_performances` function, which is the one actually measuring them for each <training, testing> subsets couples.

2.2.4 Eyes_Recognizer.py

This file contains the code we used to perform the identity prediction considering the color of the probe's eyes from the image.
The function that analyzes all the possible cases that could occur with the eyes detection is `detected_cat_eyes`. In order to have a feedback of the cutting step (the eyes are cut from the probe image given in input) and in particular to be sure that everything was done correctly by our system, the eyes of the subject in the probe image are saved in the *eyes* folder.
The idea of improving our system with the possibility of reducing the number of subjects that are considered for the final prediction, leads us to work on the extraction of the color of the eyes for subjects that are cats. We could not apply methods used in general for humans, because the pupil in cats has a different shape, which is similar to an

ellipse, when there is much light in the ambient. The analysis of the color of each pixel in the eyes, cut from the face of the cat, was the method we decided to adopt in order to obtain a color for each detected eye.

2.2.5 utils.py

This file contains several utility functions we used throughout the whole project, some of them developed by us (like those needed for plotting the performances of our system on graphs or those to resize or display images) or taken from the tutorial and modified to suit our needs (like the ones to create and read the csv files used by *OpenCV*'s face recognition models).

2.2.6 intersection.py

The code in this file was taken from a repository we found on GitHub, which is released under *MIT* license. The purpose of the main function, named `intersection`, is to detect where two curves intersect.
The implementation makes use of the **numpy** library and was useful for us to detect the point where the **FAR** and **FRR** curves intersect in order to highlight the **ERR** value in our graphs.

2.3 Main Design Choices

Here we list and motivate some of the choices we had to take while developing our project.

The recognition operation Due to the nature of the subjects, we decided to perform an *identification open set*, as it is unlikely for a cat to be able to issue an identity claim, making a classic *verification* operation unfeasible. We are aware that we could also have opted for some kind of *implicit verification* (think about some kind of "smart cat flap" in which the only enrolled entity is the one allowed to enter the residence, where the identity claim is therefore implicit as the system would try to recognize the only enrolled entity), nevertheless we decided to go on with the idea of an *identification* operation as we thought it could cover a wider range purposes.

OpenCV face recognition We chose to use *OpenCV*'s face recognizers as they do not demand a proper prior training like most machine learning approaches, which require great amounts of data to work properly (as our *data set* was created by us and has a limited size). Instead *OpenCV*'s recognizers are capable of giving good results by just applying the face recognition algorithms on the same images that end up being the templates of the enrolled users.

Performance evaluation As *OpenCV*'s training actually corresponds to the enrollment operation, the best way we could evaluate the performances of our system was using the $\text{All}_{\text{PROBE}}$-against-$\text{All}_{\text{GALLERY}}$ technique.

3 Data Set Creation

3.1 Pictures Retrieval

As we mentioned in the Introduction[1], we had to build our own *data set* for this task due to the lack of any around the Internet. As we had to perform a recognition operation, we had to collect sets of pictures all belonging to the same subjects. Some were retrieved by those among us having a cat in their own families, while some others were kindly provided by our acquaintances (relatives, friends, etc.). We had to make sure about a set of properties those images had to comply with in order to carry out a truthful, unbiased and reliable evaluation of the performances of our system. An example of the mentioned properties is the following one:

- They had to have a minimum **quality** level so that faces could be detected and recognized from them.
- They had to be taken in **different time periods** (moments of the same day, different days, etc...).
- They had to be taken under **different circumstances** (illumination, slight pose changes, etc...).
- The subject had to be in a **frontal pose** to make it possible for the face detectors to work properly.

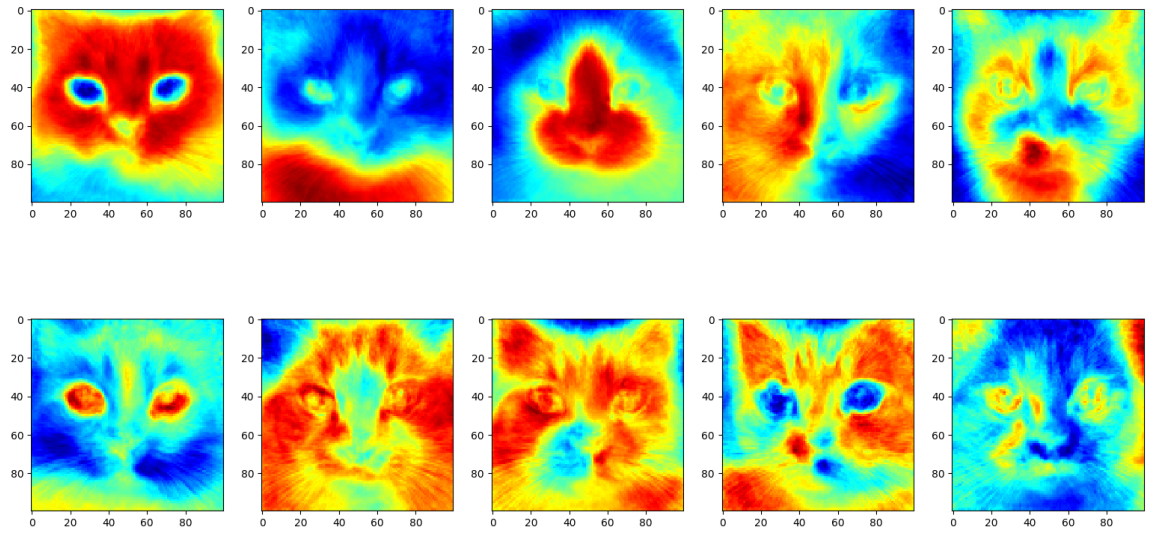

Figure 1: An example of Eigenfaces generated by our system.

Another most important source we used to expand our *data set* is the famous *social network* Instagram, where we looked for and downloaded a very good deal of high quality pictures belonging to the same cats. All the images we took of course had to have the same properties mentioned above.
We managed to collect images of **23 subjects** with **at least 12 pictures** for each one of them, for a total of **350** pictures.

3.2 Faces Detection and Cropping

After building the *data set*, we worked on actually detecting the faces of our subjects in order to apply *OpenCV*'s face detection algorithms on their faces and eventually carrying a thorough performance evaluation using an $\text{All}_{\text{PROBE}}$-against-$\text{All}_{\text{GALLERY}}$ distance matrix approach.

Since the pictures we used came from different sources and were taken in completely uncontrolled settings (i.e. no definite distance from the camera, no uniform background, different image quality - as different capture devices were used - and often not on a completely frontal pose), our intervention was often required to help *OpenCV*'s face detectors identify the subjects' faces by changing the various parameters and selecting the rectangles which actually contained the faces before cropping them.
At the end of this phase we finally managed to crop and align the faces we used for the recognition phase.

4 Recognition

For this operation we employed all of the *OpenCV*'s available face recognizers, as we intended to compare them and decide which one best suited the task of recognizing cat faces. The library provides an implementation of the three following face recognition algorithms: **Eigenfaces**, based on the PCA technique (figure 1), **Fisherfaces**, based on the LDA technique (figure 2), and **LBP Histograms**, based on the LBP technique. We used the same *data set* with each one of them and selected a few parameter configurations to try out, in order to find out any improvements in the recognition.

4.1 Addition: Filtering through Eye Color

As mentioned in Section 2, the method we decided to adopt in order to limit the subjects for the prediction and improve the performances, is completely innovative.
The `detected_cat_eyes` function tries to detect the subject's eyes, in order to cut and save them in the *Eyes* folder. Once the function obtains the images of the eyes, it proceeds to the analysis of the color of the pixels that compose each one of them, so as

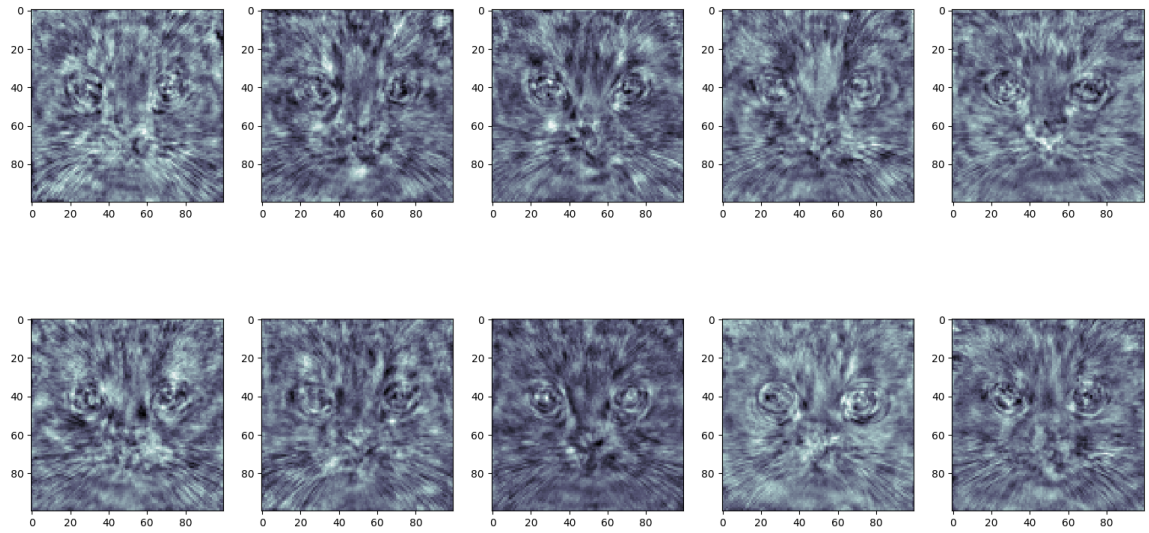

Figure 2: An example of Fisherfaces generated by our system.

to assign a possible class chosen from the following list: "*Blue*", "*Green*", "*Yellow*", "*Brown*", "*Gray*", which represent the possible colors of the cat's eyes.
The three different cases that can occur are as follows:

- Detection of both eyes
- One-eye detection
- No eye detected

One of the classes presented above will be assigned to each detected eye, so as to consider for the final filter only the classes that have been assigned to both eyes. It could rarely happen that a cat has eyes of different colors, but also this feature is detected by our program, limiting the prediction only to subjects with this peculiarity. In order to filter only the cats of interest, the *gallery_ eyes_ color.txt* file has been created, which contains, for each subject in the gallery, the respective eyes color.
In the worst case that the eyes are not detected on the probe image, the system proceeds with the normal face identification, considering all the subjects for the final prediction.

5 Performance Evaluation

We now discuss all the techniques we employed to evaluate the performances of our system.

5.1 K Fold Cross Validation

We used the *k fold cross validation* technique in order to evenly partition our *data set* into training and testing sets, respecting the property stating that there should be no overlap among them.

The technique consists in dividing the data into k subsets and then generating all the combinations where **k-1** subsets are used as training data, while the remaining subset is used as testing data. In this way, every image gets to be used for testing exactly **once**, and gets to be used for training **k-1** times. This significantly reduces bias as we are using most of the data for fitting, and also significantly reduces variance as most of the data is also being used in validation set. Interchanging the training and test sets also adds to the effectiveness of this method [1].

We decided to slightly change this technique by allowing to randomly pick an arbitrary number of subjects to be treated as **impostors** (as we dealt with an *identification open set* operation), which means that the chosen subjects' pictures would only appear in the testing sets and not in the training ones. Impostors' identities are guaranteed to be different in each one of the k subsets.

Detector	Parameter	Values	Best
Eigenfaces	# Components	[10, 80, $(width \times height)/10$, $width \times height$]	10
Fisherfaces	# Components	[10, 80, $(width \times height)/10$, $width \times height$]	80
LBPH	Radius	[1, 2]	2
	# Neighbours	[4, 8, 12, 16]	16
	Grid Size	[4×4, 8×8]	8×8

Table 1: Table containing the different parameter configurations we tested in order to select the best one for our purposes (reported in column **Best**).

5.2 $\text{All}_{\text{PROBE}}$ Against $\text{All}_{\text{GALLERY}}$ Distance Matrix

For each combination of the training and testing subsets deriving from the *k fold cross validation* operation, we built an $\text{All}_{\text{PROBE}}$-against-$\text{All}_{\text{GALLERY}}$ distance matrix, where all gallery templates are matched against each testing probe, thus associating each probe with a list of gallery templates alongside the computed distance with respect to each one of them, as our recognition operation consisted in an *identification*. The same thing obviously happened for both the probes belonging to enrolled subject and those belonging to the subjects chosen as impostors during the previous operation.

5.3 Rates calculation

Starting from the produced matrices, we computed the following rates:

- False Rejection Rate (**FRR**).
- False Acceptance (or "Alarm") Rate (**FAR**).
- Genuine Rejection Rate (**GRR**).
- Correct Detect and Identify Rate at rank k ($\textbf{DIR}_k$).

Those rates were computed for each one of the <training, testing> subsets deriving from the *k fold cross validation*, and then arithmetically averaged to draw the final results.

5.4 Tests

We conducted the tests by trying out all the available algorithms with several different configurations. We also tried to change the number of subsets to be generated by the *k fold cross validation* routine.
We performed our tests using three different versions of the same *data set*, which we named in the following ways:

- **Complete** (located in the *images/dataset/cropped* directory): this version contains all the images we extracted faces from.
- **Best** (located in the *images/dataset/best* directory): this version contains a selection of faces taken from the **Complete** *data set* with the following properties: high image quality, good cropping (i.e. face only without any background elements), (almost) exactly frontal pose.
- **Best_aligned** (located in the *images/dataset/best_aligned* directory): this version contains all the images in the **Best** *data set* which we managed to correctly align with respect to the eyes.

The reason for using those three different versions of the *data set* was to see how the performances of the system changed using more and more finely selected pictures.
We firstly ran a full suite of tests, without employing the eyes recognition routine not to bias the recognizers' results, trying a quite large variety of different configurations, in order to find out the best one(s).

After choosing the best recognizer and parameter configuration for it, we ran the tests on them once more, this time adopting the eyes recognition routine 4.1 to filter the prediction results and see whether it actually

helped improving the overall performances of the system.

6 Results

Here we describe in more detail the kind of tests we performed and their respective results.

Regarding the **k fold cross validation** technique, we chose to set the value **k=5**, which empirically is one of the most used ones when it comes to evaluating the performances of a system like ours and which actually revealed to be the best value for our system as well. **5** is also the number of subjects we decided to be picked as impostors, the reason for this choice being that this value is equal almost to $\frac{1}{4}$ of the number of subjects in our *data set*, which we thought to be a fair ratio. The **k fold** files we generated and used to get the results we illustrate in the following pages can be found in the *test/k_fold/* directory.

For what concerns the *thresholds*, we performed our tests using sets of 100 equidistant thresholds for each algorithm. The initial value from which each set is generated corresponds to the overall average of the average distances of any probe with respect to every template in the gallery, computed separately for every algorithm (as the average distance varies from one algorithm to the other).

6.1 Parameters Tuning

For each one of the available face recognition algorithms, we chose some combinations of parameters (see table 1) and compared them to establish the one that best suited our task. We performed this operation on all the three versions of our *data set* and noticed that for each algorithm every configuration followed a similar trend in each one of the versions. For this reason, and also to avoid overfilling this report with too many graphs for readability purposes, we decided to show the comparison of the **ROC** curves only for the **Complete** *data set* version (figure 3).

At the end of this phase, we concluded that the best configurations were those in the **Best** column of table 1, which led to

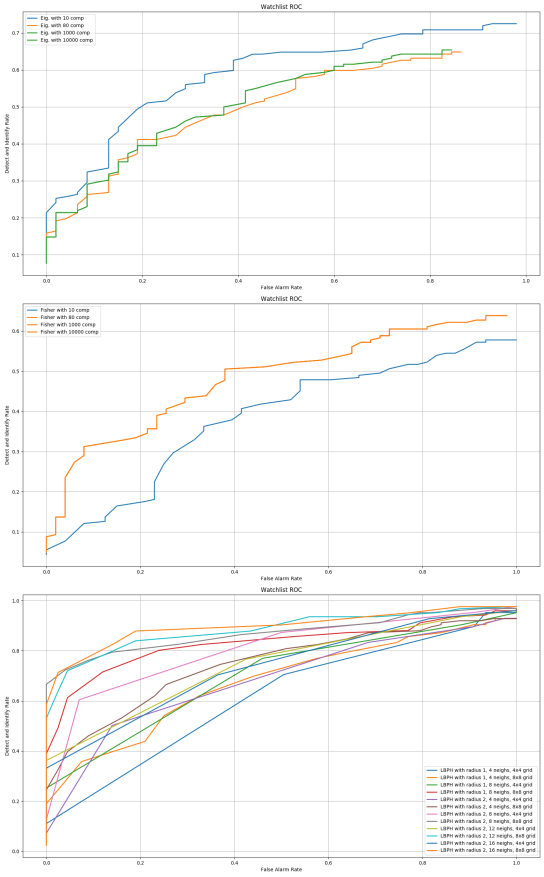

Figure 3: Comparison of ROC curves for all the selected configurations of the three models (*Eigenfaces*, *Fisherfaces*, *LBPH* in order) on the *Complete data set*.

achieving the lowest **ERR** values for each algorithm.

Let us go a little deeper into the details:

Eigenfaces We noticed that, surprisingly enough, the **Eigenfaces** algorithm works better in our system when the number of components is low: **10**.

Fisherfaces Regarding the **Fisherfaces** algorithm's performances, we noticed that they remain stable when the number of components is greater or equal than **80**, which is the value we chose also because it is the least demanding one in terms of required computational time.

LPBH In the case of the **LBPH** algorithm, we noticed that the best values for the radius, number of neighbors to consider

and grid size are those requiring more computational time and resources.

6.2 Recognizers' Perfomances and Comparisons

Once we identified the best parameters configurations for each face recognition algorithm, we ran our performance evaluation routine for each one of them using the selected parameters on all the three versions of our *data set*, in order to eventually compare the three algorithms and elect the best one for our task.

6.2.1 Complete

In figure 4 we show the FAR-FRR curves for each one of the three algorithms when the perfomances are evaluated over the **Complete** version of the *data set*. The precise **ERR** values can be found in table 2.

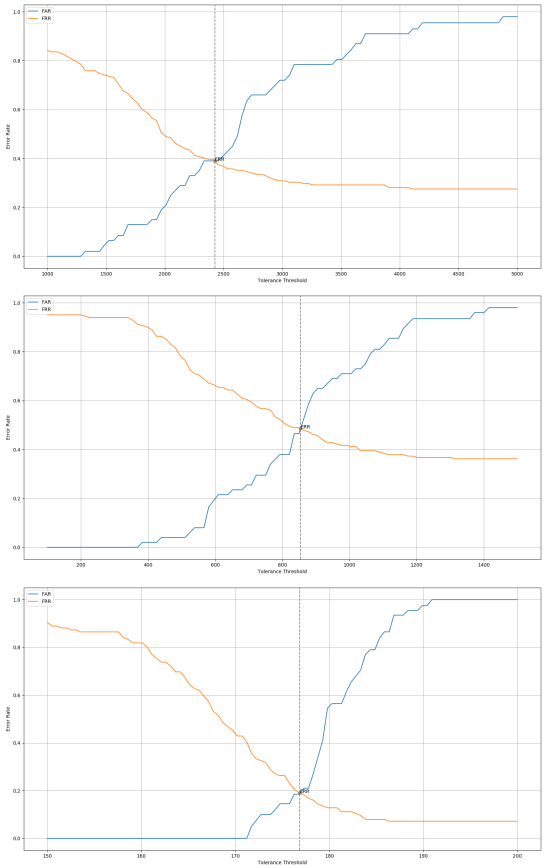

Figure 4: Plotting of the FRR-FAR curves for the (in order) *Eigenfaces*, *Fisherfaces* and *LBPH* algorithms for the *Complete* version of the *data set*.

Recognizer	Complete	Best	Aligned
Eigenfaces	0.39	0.38	0.33
Fisherfaces	0.49	0.36	0.35
LBPH	**0.25**	**0.16**	**0.19**

Table 2: Table containing the values for the *ERR* statistic computed for each *data set* version using each face recognizer with best configuration selected in [6.1].

6.2.2 Best

In figure 5 we show the FAR-FRR curves for each one of the three algorithms when the perfomances are evaluated over the **Best** version of the *data set*. The precise **ERR** values can be found in table 2.

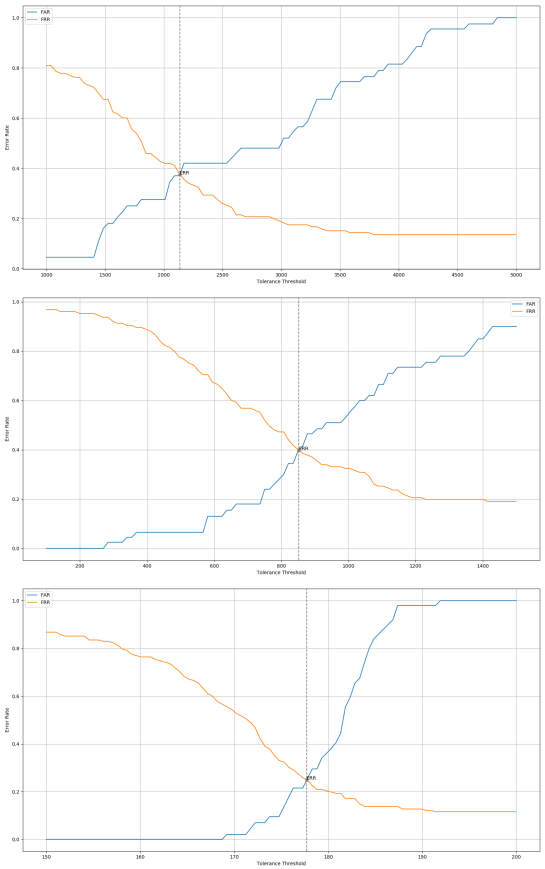

Figure 5: Plotting of the FRR-FAR curves for the (in order) *Eigenfaces*, *Fisherfaces* and *LBPH* algorithms for the *Best* version of the *data set*.

6.2.3 Best_aligned

In figure 6 we show the FAR-FRR curves for each one of the three algorithms when

the perfomances are evaluated over the **Best_aligned** version of the *data set*. The precise **ERR** values can be found in table 2.

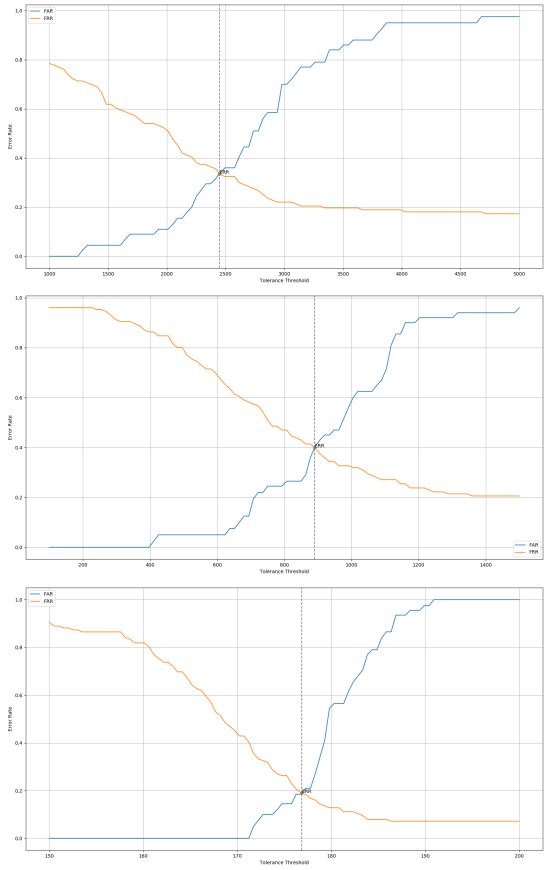

Figure 6: Plotting of the FRR-FAR curves for the (in order) *Eigenfaces*, *Fisherfaces* and *LBPH* algorithms for the *Best_aligned* version of the *data set*.

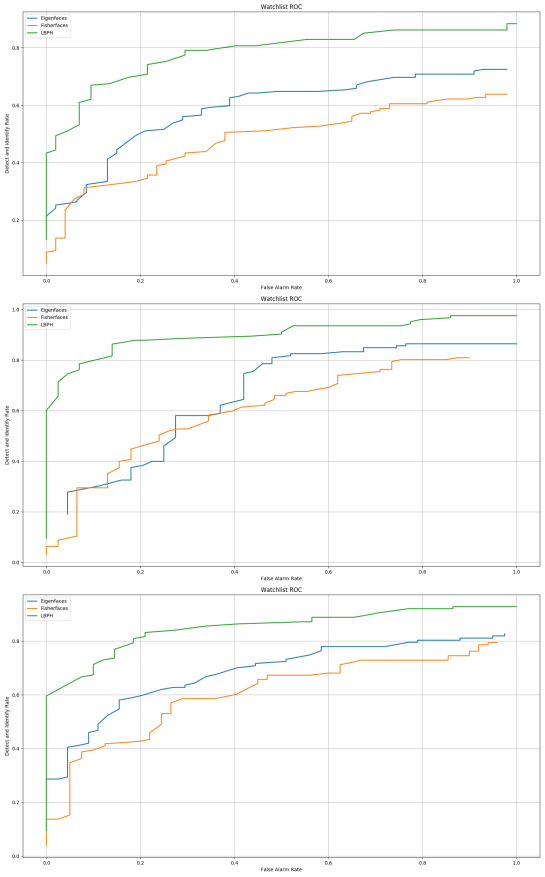

Figure 7: Comparison of *ROC* curves for each one of the three algorithms after selecting their best configuration. One graph per *data set* version (*Complete*, *Best*, *Best_aligned* in order).

6.3 Overall Comparison

As it can be seen from the single graphs plotted for each *data set* version and the values present in table 2, it straightforward to observe that he **LBPH** algorithm resulted to be the best one among *OpenCV*'s face recognizers for our *data set* of cats faces. We decided to plot the **ROC** curves for each algorithm together in figure 7 for a quicker and more immediate comparison, while we chose not to do the same thing for the FAR-FRR curves as their plots would get too messy when put together.

For what concerns the *data set* versions, it is quite evident to see, especially from table 2, that performances generally improve when the data is more refined and retains an overall better quality.

6.4 Integration of the Eyes Color Detection

Once having selected the best recognition algorithm and its best parameter configuration, we used them to evaluate the performances of our system while also employing the eyes recognition routine [4.1] we developed. Unfortunately, we noticed that this approach did not bring any benefits to the performances of our system, actually making them worse. This is mainly due to the fact that *OpenCV*'s eye detector often detects as an eye what is actually not one (e.g. some fur parts in the body of the cat in most of the cases), which led the `detect_cat_eyes` function to make mistakes in the sense of

cutting out the correct subjects from the recognition results, thus negatively affecting the overall prediction process. Graphs are reported (see figures 8 and 9) for reference and comparison with respect to the recognition process without employing this routine.

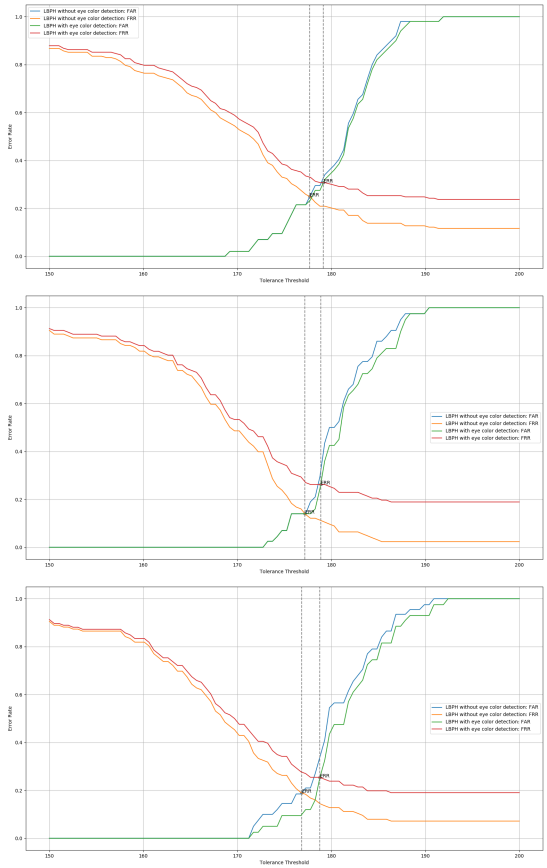

Figure 8: Comparison of the system's performances (FAR-FRR curves) while employing the eye color detection routine with respect to the previous tests (without employing the routine). One graph per *data set* version (Complete, Best, Best_aligned in order).

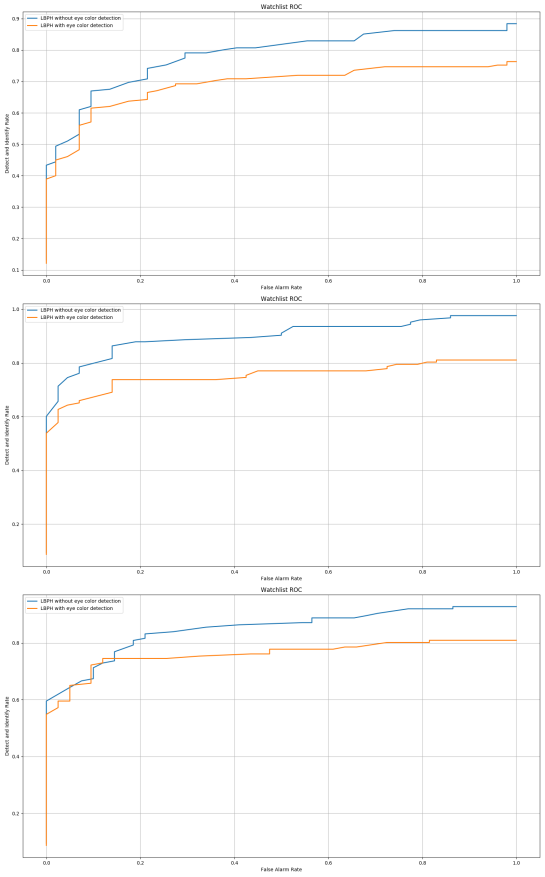

Figure 9: Comparison of the system's performances (*ROC* curves) while employing the eye color detection routine with respect to the previous tests (without employing the routine). One graph per *data set* version (Complete, Best, Best_aligned in order).

7 Conclusions

Analysing the results we obtained, we cannot say that our system can boast particularly great performances, this can be due to several reasons: from the small-sized *data set* which was made by us in a limited time using pictures not always retaining good quality standards, to the use of algorithms which might not actually be suitable to be employed to recognize faces belonging to cats.

Several kinds of attempts could be made to try to improve our system, for example the construction of a better *data set* containing a higher number of subjects and more high quality pictures for each one of them and the employment of a more suitable technique (possibly machine learning-based) for the recognition of the faces. The idea of using the subjects' eyes was quite interesting but at the same time hard to implement due to the limitations posed by *OpenCV*'s detector, which we could also try to train ourselves using only images of eyes belonging to cats.

Nevertheless we do not regret our decision of taking on this unusual and challenging task no one probably ever dealt with before and we must admit it was something quite interesting for us to work on.

References

[1] Maria De Marsico. Lesson 2bis - more on performance evaluation. URL https://drive.google.com/file/d/1eeqBvXhONkqRUxQLoHOuUhD_4PNG4WpT/view.