UNIVERSITY OF TWENTE.

Design and validation of a face recognition framework

EE MSc Thesis

Author:

F. van Capelle, BSc.

Supervisors:

Prof.Dr.Ir. C.H. Slump

Dr.Ir. R.N.J. Veldhuis

Dr.Ir. L.J. Spreeuwers

Dr. M. Poel

June 13, 2013

Abstract

We have developed a framework that standardizes research in the field of face recognition. The framework endorses the use of interchangeable modules that can be developed and tested independently during subsequent researches. At the same time it does not impose heavy restrictions on the implementation of these modules so that future studies are not impeded in any way. Using this framework, we have implemented and tested a score-level algorithm fusion recognizer. Results show that the performance of a recognizer can be improved by using fusion even if the base classifiers are not very accurate.

Contents

Chapter 1

Introduction

> "Make everything as simple as possible, but not simpler"
>
> - Albert Einstein

Under controlled circumstances (with indoor lighting and cooperating subjects) present-day systems perform remarkably well. An example is the automatic passport control station that is presently in operation at several airports around the globe. However, the problem of face recognition is not by any means fully tackled. When uncontrolled circumstances arise, and uncooperative subjects are to be recognized, most face recognition systems fail miserably. In this area, further studies are definitely required.

The research in the field of biometric pattern recognition in general, and that of face recognition in particular, is constantly on the move these days. New recognition algorithms are proposed so regularly that it is hard to keep up reading them all. Most of these researches focus on a single stage in the recognition 'chain of events'; only rarely do we come across a paper that proposes a new registration method *and* a classifier algorithm. Of course, in principle there is nothing wrong with researching these stages separately from each other. It is, in fact, the preferred way of researching, as it eliminates the influences of the other stages. However, as these other stages are usually filled using older, less performing standards (such as Principal component analysis or the Viola-Jones face detector) the results that arise from these studies are not one-on-one comparable to the state-of-the-art in face recognition.

To make study results comparable to the state-of-the-art, a research integration program is needed. Within this program, it will be possible to easily combine the results of one stage to those of another. This way, the recognition stages can be developed independently, while contributing to the recognizer as a whole. This leads to a single face recognition system that will gradually improve over the course of multiple researches.

To this end, we start out by developing a framework that standardizes the stages of face recognition. It will serve as the basis for the described system and will, in time, serve as a fully operational camera surveillance system. Once ready, it will flexibly combine a number of cameras, multiple face recognition algorithms and image processing techniques. It is able to identify passers-by and can warn if a person on a watch list is detected[1].

Furthermore, we will implement and test a basic fusion algorithm to show the capabilities of the framework and, more importantly, make the first step towards the envisioned system.

Report overview

We will begin this report by discussing the design and implementation of the framework in depth in chapter 2. This will then be followed by a discussion on whether the specifications are met by the framework and recognizer system in chapter 3. With the validated framework as the basis, we will step into the fusion research in chapter 4. Finally, we will discuss how future research might best continue with the created work in chapter 5.

To enhance the readability of this report, we choose to write in the first-person plural active voice over the passive voice.

Enjoy.

F. van Capelle, BSc.

Chapter 2

Building the framework

Building a framework. It sounds easier than it is. A framework – one that can accommodate all types of future research in the field of face recognition – is in fact quite complex. The most challenging part is in not knowing what future research might need in a framework. Therefore, it must be as flexible as possible while, at the same time, standardization must be pursued.

We start out by defining the specifications to which the framework has to be built. When these are clear, we will dive deeper into matter and unfold how we have worked towards the requirements. At the end, we will show how we use our framework to implement and test a new implementation of a face recognition algorithm.

2.1 Specifications

In this section we will describe the project objective together with the extracted requirements and the made design considerations and choices.

2.1.1 Project objective

After the first exploratory research we have come to the following statement for the project objective, which will be our major guideline throughout this project:

> The project objective is two-fold: on the one hand it will be used to standardize research, whilst on the other, it will serve to demonstrate our group's current capabilities.

The details of these objectives are described in the following two sections.

Standardize research

First of all, the framework is designed be a standardized platform for face recognition research. At the moment, it is fairly common that a face recognition research is focused on a particular part of a system, such as the correction of illumination differences. When this approach is used, usually other necessary parts of the system (such as registration and feature extraction) are filled in using the standards like Viola-Jones' face registrator and/or Principal Component Analysis. But this approach has the major drawback that new developments are always compared to old algorithms instead of the current state of the art. Instead, we would like to be able to compare new developments in a face recognition system to some of our previous (stable) releases without much hassle.

We also want a system that is able to compete with the contemporary state of the art. As the state of the art is subject to constant development, this system must be easy to reconfigure to cater new possibilities as they arise from research. Furthermore, in order to rank our system among competitors, it is necessary to make use of one of the various available standardized testing protocols.

Demonstrate capabilities

The framework will also be used as a demonstrator of research results. Whenever our group has developed a new (and better) face recognition algorithm, we would like to be able show this to others. Of course, it is possible to simply present interested clients and fellow researchers the performance figures, but it is much more appealing to see the system live in action.

From this, it follows that the framework should be able to do enrolment and identification/verification experiments on a stand-alone computer, so that demonstrations can take place on location. But since training is a very, if not, the most, computationally expensive part of setting up a new algorithm for most recognition systems, this is usually done on a powerful multi-core mainframe computer, which is not very portable. Possibly the best solution to overcome this problem is to design the system in such a way that it allows to separately train and test algorithms.

2.1.2 Requirements

In this section we derive the framework's requirements from the system objective.

- We develop a framework for face recognition experiments. The framework standardizes experiments and a recognizer's I/O.
- We develop a recognizer that can perform automatic face recognition on single still images.
- The software can run on either Windows or Unix-based machines.
- Recognizer functionality is implemented as modules, which can be easily substituted with improved versions.
- The framework provides enough flexibility to implement the majority of future face recognition algorithms. This means that the chosen architecture may not impose great restrictions on module implementations.
- More complex recognizers, such as fusion algorithms, can be implemented.
- Recognizer modules can be trained on an external machine, separately from the enrolment and verification/identification experiments.
- The framework is well documented, since future researchers have to work with it without much hassle.
- In order to compare recognizers against the state of the art, we use the Face Recognition Grand Challenge tests[2]. The framework should thus be able to handle those datasets.
- Every recognizer under test, whatever the implementation, should yield results in the same form to accommodate an effortless comparison.
- The framework is able to capture camera stills and enrol/compare them to a gallery database and display match results.

2.1.3 Design choices

Based on the system requirements, we have made a few design choices. This was done before any actual work was started, and is based solely on on-line research and brainstorming. In this section we describe and defend our major design choices by giving the made considerations.

High-level considerations

- OpenCV is used as the primary image processing toolbox[3]. OpenCV makes coding more high-level, since a lot of basic image processing routines are already implemented and tested extensively in this library. The library is supported by an active community and is subject of ongoing development, meaning that it will become a more and more useful toolbox. OpenCV has C, C++ and Python interfaces available.

- The program will be written C++. Reasons to choose this language are 1) that it interfaces with OpenCV, and 2) that it is object-oriented, which comes in handy when designing a module-based architecture. Also, choosing a C-style language allows for any Matlab code to be converted easily[4]. This is a meaningful consideration, since a lot of research at our group is already done using Matlab.

- For the first iteration, we partition the face recognition system into the following categories: 1) Detection, 2) Registration, 3) Illumination correction, 4) Feature extraction, 5) Comparison to the gallery database. This structure serves to keep the works organized. Modules should be assigned to a certain category so that future researchers can easily find any (previously implemented) module they are looking for. On a side note, the described partitioning is by no means strict and can be expanded to suit the needs of future researches.

- Intermediate output can be stored to disk, so that computationally expensive or stable running stages need to be executed only once instead of on every run. If, for instance, research is done on feature extractors, it is undesired that the detection, registration and illumination stages must run with every trial, as their output is independent of the implementation of the feature extractor and doing this would have a rather large negative influence on the processing time.

Low-level considerations

- The first step towards face recognition is a conversion to monochromatic images. These images will be the basis for all subsequent processing steps. This step is performed in the majority of commercially available face recognition software as well as in most researches, and as such has become commonly accepted as beneficial to a system's recognition speed while barely influencing its performance.

- Internally, matrices and images (of arbitrary size NxM) are represented as OpenCV single-channel 32-bit floating point matrices (denoted as cv::Mat(N,M,CV_32FC1)). We choose to work with 2-D matrices only.

- The standard I/O format for images is the SFI format (Single Float Image)[5]. The SFI format is very compact: besides an header of approximately 17 bytes (The format specifier "CSU_SFI", the height, width and number of channels), it requires only 4 bytes per pixel for storage. Pixel grey-level intensities are represented as 32-bit floating point numbers in the range [0,1] and stored in row-by-row concatenated form. As we have chosen to work with monochromatic images, we only implement a single-channel SFI-reader/writer. Basic image formats such as PNG and JPG are also accepted as input, but are never written.

- The standard I/O format for matrices is the ASCII format, in which spaces (per column) and newline characters (per row) are used to separate matrix elements. This format uses a little more disk space than the SFI format, but has the advantage of being human readable. Furthermore, existing software at our group already use this format.

- The framework provides a form of log-file output so that system tests can be monitored easily. Such a feature is a bare necessity when it comes to finding bugs that are inherent to developing new software algorithms.

- All classes and functions associated with the framework reside in a designated namespace: `utbpr`, an abbreviation of University of Twente, Biometric Pattern Recognition.

2.2 Design

Now that we know we want to design a framework for face recognition, it seems that a good starting point is to investigate how a face recognition system roughly works and how we are going to test such a system. We will now first describe a general face recognition system, followed by an overview of the FRGC experimental setup. Lastly, we provide a summary for ease of reference.

2.2.1 Abstract of a face recognition system

Any face recognition system follows, in essence, the same principal procedure:

> To verify a person's identity, a set of input images and a claimed identity are provided to the system. Identity-representing features of the photographed individual are extracted. A comparable set of features is retrieved from a database for the claimed identity. A comparison of the two is done and the output will be a score representing the similarity between the two[1].

Each of the statements above will be elucidated in the following paragraphs.

A set of input images and a claimed identity are provided

There are very little restrictions on the composition of the set of input images, but one very important restriction is that each image in the set holds information of only one individual and this individual is the same for all images. From here on, we will refer to this set as the 'input set' and the imaged individual as the 'subject'. On each run of the system, only one input set is presented.

Identity-representing features are extracted

From the given input set the identity-representing features are extracted. This is usually done in two main stages: 1) preprocessing and 2) feature extraction. We will refer to this combination as the preprocessor and feature extractor system, or PFES for short.
The preprocessing stage tries to normalize the input set by filtering unwanted effects that are present. Examples of preprocessing steps are background separation, pose normalization and illumination correction.
From this normalized input, the feature extraction stage actually extracts the identity-representing features, also known as the 'feature vector'. An algorithm might extract multiple features vectors and thus we prefer to use the more general term 'feature set'.

[1] Of course, there are many variations on this depending on the use-case of the system (e.g. no claimed identity is provided forcing the system to search the database for the best score), but the outline stands.

Such a set can contain any number of feature vectors (one at minimum). This is represented in figure 2.1.

The key to devising a good face recognition system lies in extracting a feature set that is highly distinguishable from any other subject's feature set and is highly reproducible. This is one of the most challenging problems in the field of image processing.

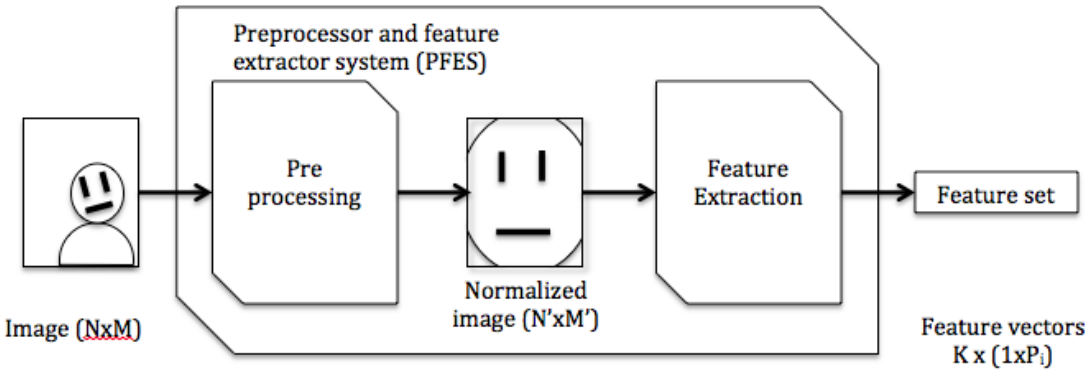


Figure 2.1: A preprocessor and feature extractor system (PFES) is comprised of preprocessing and feature extraction stages for a single image input set (generalization to multiple image input sets is straightforward). The size of the normalized image N'xM' (and therefore P_i as well) is usually fixed by the implementation of the system, i.e. independent of N and M.

A comparable set is retrieved from a database

All subjects that should be recognized have to be enrolled beforehand. Enrolment is usually done under controlled circumstances and the identity of the individuals is annotated manually. The feature set of an enrolled subject is stored in the database for future references. Given a claimed identity, the corresponding feature set can be retrieved from the database after enrolment.

Figure 2.2: Enrolment (left) and lookup phases (right) of a database. Subject lookup can only be done *after* enrolment of that subject.

Comparison of the two feature sets

The feature set extracted from the input set and the database record have to be compared. In general, there are two types of comparison outputs. On the one hand, there are similarity scores (where higher scores indicate better matches) and on the other hand there are dissimilarity or distance scores (smaller scores are better). Optionally, should we only want to check whether or not the queried subject is indeed who he claims he is, the score can be thresholded to give a match/non-match boolean value.

The comparison algorithm is usually selected to fit the algorithm used for feature extraction. In some cases it might even be specially designed. Because of this, comparison algorithms can be seen as an integral part of the face recognition system. We will discuss possible implementations later in this report.

Figure 2.3: Comparing the feature sets of the target and query. The thresholding step is optional.

2.2.2 Large scale testing

The best, and perhaps only, way to test a recognizer's performance is by enrolling and querying a large amount of face images. We could then determine in how many percent of the runs the system produces the desired outcome (i.e. correctly identifies/verifies the presented subjects), but since this result is highly dependent on the choice of the comparison threshold, we eliminate this parameter by simply not thresholding the scores. Instead, we use the Receiver Operating Characteristic (ROC) to study the performance[6].

FRGC description

To be able to compare experiments between researches, a multitude of standard databases and testing methods are made available. We have chosen to work with the FRGC. The FRGC, short for Face Recognition Grand Challenge, consists of six challenges, each for a different type of face recognition research[2]. The challenges are punctiliously documented and are all of the same form: "match all the images in the query set to the images in the target database, while using only the training set to train your system". Here, a query indicates a subject under test and a target indicates a subject that is already in the database. These three sets (query, target and training) are predefined in so-called signature files. In theory, there should be no overlap between the identities in the training data and the validation data[2], but unfortunately this is not case in the FRGC data. For the sake of comparability to other researches, we choose not to correct this.

Score matrix

While we keep in mind that, in the future, all FRGC challenges might be tackled using the proposed framework, we will focus on challenge no. 1: single controlled 2D still queries vs single controlled 2D still targets. This is an all-vs-all matching experiment: all queries are compared against all targets, and results are stored in a score matrix. The target and query lists are identical, giving rise to same-image comparisons. This is an unwanted effect and, therefore, these comparison results are discarded before analysis of the data. This is elucidated in figure 2.4 on the next page.

[2]Validation data is the combined set of target and query data.

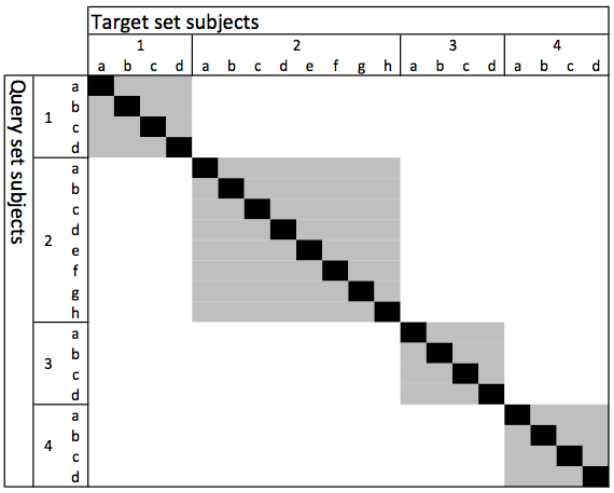


Figure 2.4: All-vs-all matching score matrix, in which each column is associated with one target and each row is associated with one query. Multiple images per subject are tested. If an image is tested against another images of the same subject, the score is marked as a genuine score (grey). When matched to a different subject, it is marked as an imposter score (white). Scores from images tested against themselves are discarded in statistical analysis (black).

2.2.3 Summary

In this section we present the abstract overview of the singe-query experiment described in section 2.2.1 and an overview of the all-vs-all matching experiment described in section 2.2.2.

Overview of a single-query experiment

The face recognition abstract that we have described can be summarized using the figure 2.7 on the following page. It describes a typical single-query experiment (e.g identity verification at an airport passport control). Before this recognizer (the combination of the PFES and comparator) can be put to use, it must be trained using training data and the database must be filled. Training is depicted in figure 2.5. Once the recognizer is fully trained, enrolment can take place (figure 2.6). All subjects that the system should identify, must be stored in the database. The PFES processes the target images and the resulting feature sets are enrolled to the database together with the target ID. The system is then ready for query processing, i.e. the actual face recognition (figure 2.7). When a query subject is presented, its image is also processed, while the enrolled data of the claimed identity is retrieved from the database. The two feature sets are compared and the resulting score can optionally be thresholded to give a match/non-match boolean value (not depicted here).

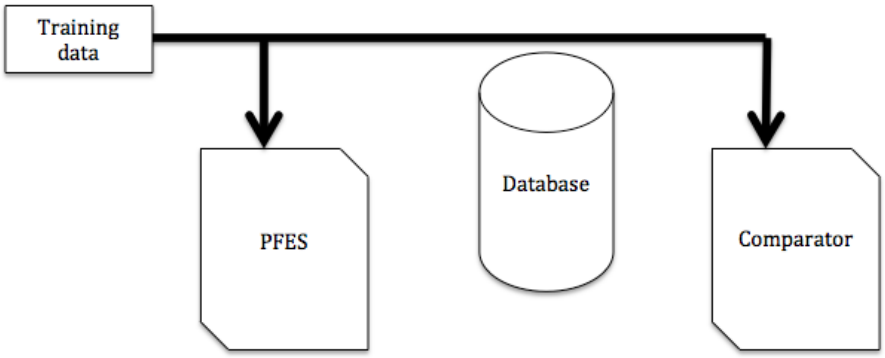


Figure 2.5: Training phase. The PFES and the comparator can be trained using the same data. Whether this is necessary depends on the implementation of the modules.

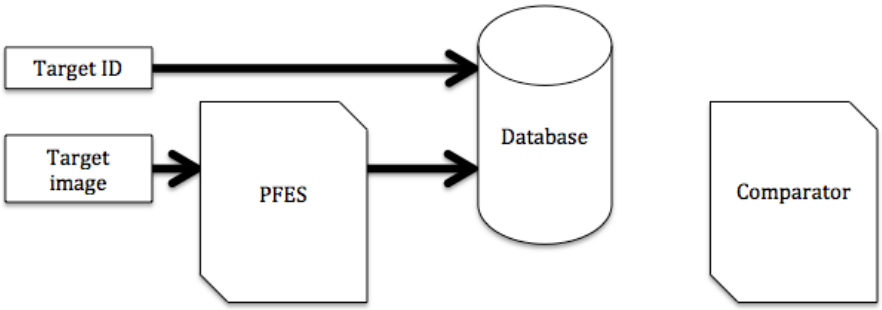


Figure 2.6: Enrolment phase. All target images are processed to feature set by the PFES and stored in the database with the ID label for future reference.

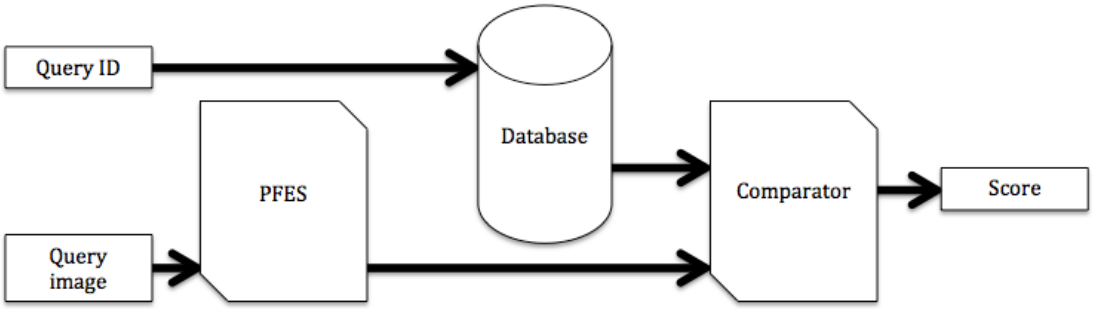


Figure 2.7: Query phase. The query image is processed to a feature set by the PFES. A feature set lookup from the database is performed using the query ID label. The two are compared to give a matching score.

Overview of an all-vs-all experiment

When doing all-vs-all matching experiments where the query and target lists are identical it is unnecessary to process all images twice. In such a case we use the simplified scheme depicted in figure 2.8. The training and enrolment is performed in the same way as before (figures 2.5 and 2.6).

As all-vs-all means that the target and queries are the same, we refrain from processing the queries using the PFES. Instead, we retrieve the queries from the target database as well. We compare all possible combinations of two database records, store the scores in a score matrix and annotate the corresponding target and query IDs.

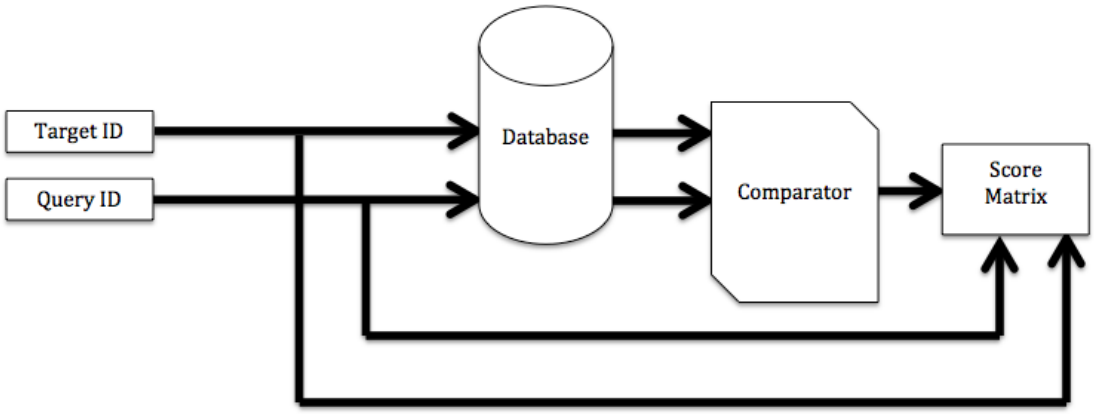


Figure 2.8: Overview of an all-vs-all matching experiment. The PFES is otiose and thus omitted. Both the target and query feature sets are retrieved from the database and compared to one another. All scores are stored in a score matrix (like the one in figure 2.4 on page 12).

2.3 Implementation

In this chapter we will give an overview of the implementation of the various aspects of the framework. Outlines and ideas are presented, as well as simple use-cases for the end user. Detailed descriptions have been omitted, but can be found in appendix C.

2.3.1 Error handling

Throughout the framework, error handling is done based on throwing exceptions. Wherever an error is foreseeable by the author, an error guard is implemented that will throw a std::runtime_error if that error occurs. The thrown exception contains a string descriptor of the error that is returned to the calling function. If the calling function does not handle the exception, it propagates to the next-level caller. This is repeated up until the main() function is reached where, if still not handled, the exception generates a terminal fault and aborts the program without informing the user of the nature of the error. As this is undesirable behaviour, it is important to handle the exception before an abortion triggered. Handling exceptions is done using a try-catch combination: whenever something inside the try-block throws an exception, the catch-block catches the exception and handles it. The most basic catch handler only displays the error on screen, but more sophisticated actions can be taken. Furthermore, nesting of try-catch combinations is allowed.

As an example, consider the following function:

```
void functionThatMightThrowAnException(){
  ...
  if(SomethingBadHappened)
    throw(std::runtime_error("Something␣bad␣happened."));
  ...
}
```

Suppose that SomethingBadHappened is set to true for some reason. Then, the runtime error will be thrown. The exception is properly handled if the calling main() program has a try-catch structure wrapped around the error producing function:

```
int main(int argc, char* argv[]){
  try{
    functionThatMightThrowAnException();
  }
  catch(std::exception &errMsg){
  printf("\nError:␣%s",errMsg.what());
  functionThatResolvesTheException();
  }
  ...
}
```

Here, any exception thrown in the try-block is caught by the catch-block. This catch-block displays the error ("Error: Something bad happened.") and then resolves the exception using functionThatResolvesTheException(). The code does not throw any further, nor does it exit or abort, but proceeds normally with whatever is after the catch-block.

2.3.2 Toolboxes

During the development of the framework, we found that several simple functions should be accessible from all classes. To prevent the reimplementation of these functions inside each class, we developed a set of toolboxes in which those functions can reside. The toolboxes are classes that contain only static functions, making instantiation unnecessary. During the first iteration we implemented two toolboxes: FileIO and Image.

Toolbox FileIO contains functions that operate on stored files, such as reading/writing of images and matrices. Furthermore, the FileIO toolbox has the functionality to create a log file that modules can log data to. To do this, a log file FILE pointer is generated in the main() program:

```
FILE* logFile = utbpr::FileIO::openOutputFile("C:/output/log.log");
```

This pointer is passed on to modules during construction. Using the pointer, a module can write data to the log file:

```
if(logFile)
  fprintf(logFile,"\nProcessing␣%i␣images.",nImages)
```

The if-statement safe-guard is used as modules may have received a NULL pointer during construction, indicating that no output should be written to file by those modules. At the end of the main() program, the file pointer is destroyed using

```
utbpr::FileIO::closeOutputFile(logFile);
```

Toolbox Image contains functions that operate on images in RAM, such as a BGR to gray conversion and an image display method. An example of a call to the Image toolbox is:

```
utbpr::Image::showImage(image,"imageTitle");
```

Both toolboxes can be expanded further in following iterations. Also, new toolboxes might be added whenever the existing toolboxes do not provide enough flexibility. When expanding the toolboxes it is important to remember that future users will only use those functions that are located in easy to find places, so please consider preserving a proper grouping.

2.3.3 Image input

As the face recognition systems that will be designed and tested using the framework will all use the same (FRGC) images as input, a standardized way of image input is desirable. The framework accommodates for this and furthermore provides a camera feed reader.

Camera

The Camera class uses OpenCV's default camera interface. Upon instantiation, one camera attached to the PC is detected. When multiple cameras are found, the one that comes first in the device listing is selected. On each call to the class, the camera feed is displayed on screen until a key is pressed by the user. Then, a snapshot is taken and stored at a reserved memory address for further processing, while returning a boolean true. If the user pressed the escape key, no image is stored and a boolean false is returned indicating that no more images should be expected.

ImageReader

The ImageReader class functions in a similar fashion as the Camera class but now a list of image paths and subject IDs is given during instantiation. On each call, the function reads the next image from disk, stores it at a reserved memory address and returns a boolean true. When the end of the list is reached, a boolean false is returned. Upon request, the corresponding subject ID and the image path can be retrieved as well.

Although the FRGC signature files are provided in XML-format, we have used derived plain text files as input for the ImageReader to reduce coding complexity. Each line in such a file is an entry and is comprised of, at minimum, the subject's identity and a string path to the location of the associated image. ImageReader can automatically prefix the given paths with a standard base path so that the signature files do not need to contain absolute paths.

2.3.4 Database

For the first iteration implementation of the framework, we created a sequential `Database` class[3]. In write mode, `Database` stores each record as one line in a database file (.db), using the format "subjectID;imagePath;featureVector1;featureVector2;...;", where imagePath is an optional parameter and the number of feature vectors depends on the given input (but at least equal to 1). Furthermore, feature vector values are stored as floating point literals, truncated to 6 decimals.

[3]Here, 'sequential' means that the records are not indexed and thus direct lookup by subject ID is not possible.

In read mode, `Database` produces the next record in the file on every call. The subject ID is returned as an unsigned integer, imagePath as a string (containing a whitespace if none was stored). The feature set is returned as a vector of cv::Mat. When the last record is reached, the database has to be rewinded to starting position.

Since the HDD I/O footprint is quite large, we have also implemented a `DatabaseCached` class. This type of database stores its records in the RAM instead of the HDD. This speeds up database lookups at the cost of using (a lot of) extra memory from the RAM. The choice for using one or the other is up to the end user.

2.3.5 ScoreMatrix

The ScoreMatrix class keeps track of matching scores. Each row in the matrix is associated with one query and each column is associated with one target in the database. The score matrix is stored as an ASCII-file, as are the lists of subject id numbers for the targets and queries. As with `Database` entries, scores can only be stored sequentially, i.e. random access storage in the matrix is not allowed for this first iteration. This is not a heavy restriction since it is common to test one query against all targets before advancing to the next query, and this is especially true for all-vs-all matching experiments (which is our primary focus). Furthermore, this implementation is suited to accommodate the behaviour of the database (which was also sequential, see previous section).

2.3.6 SISO modules

Now that we have described the supporting functionality of the framework, we can proceed to the description of the core: the Module architecture. As was stated in section 2.1.3, we divide the working of a face recognition system into five stages. Each stage uses the output of the previous stage as its input, and by doing so contributes a little to the ultimate goal of recognizing a face. Each stage can be the subject of a new research in the future and the result of such a research may be added to the framework. In order to accommodate for such expansions and improvements, while still keeping the framework manageable, we introduce standard SisoModules.

Any newly developed algorithm can be implemented as a child class of the SisoModule as long as it meets the constraint of working in a single input single output (SISO) fashion, of which both input and output are OpenCV matrices. This constraint needs to hold during the test phase only; any training phase functionality of the new module is completely free of constraints. The standard (testing phase) call is implemented as the `process()` function:

```
cv::Mat output = someSisoClass.process(cv::Mat input);
```

This is inherited by all SisoModule's child classes. SisoModules are especially useful for automated preprocessing (a single image is inserted and processed into a single new image), but as long as no supporting metadata is required the SisoModule form can be used to accommodate for any kind of image transformation, even including feature extraction.

To instantiate a class that is derived from SisoModule, at least the following parameters must be set using the constructor:

- The level of screen verbosity, given as an integer. Here, 0 indicates nothing is to be written to screen, 1 indicates text only output, and 2 indicates all intermediate output images must be displayed as well.
- A pointer to a logFile. This pointer can be generated using the designated function `openOutputFile()` in the FileIO toolbox, but a NULL pointer is also allowed. If a NULL pointer is given, no data will be logged by the module. Note that the logFile and screenVerbosity setting are completely independent.
- The name of the implemented module, given as char*. This name is used as a reference to find out which module generated a certain output or to see where an error has occurred.
- A three-letter identifier of the type of module. Examples of such identifiers are {det,reg,ill,fex,com}. Like the *name* parameter, this identifier also serves to find the source of a generated error.

Besides these parameters being set, the constructor of a class derived from SisoModule can be of arbitrary form.

The underlying concept for this approach is based on casting. Derivatives of the SisoModule can be constructed using an arbitrary set of parameters. After construction, such a derivative can be cast (by pointer) to SisoModule form. This property is useful for the cascading of SisoModule child classes in a vector. Because of this casting property, the use of the SisoModule as the base class for new algorithms is recommended whenever possible.

process() and implementationMain()

After instantiation and casting, SisoModules derivatives can only be called upon by using the process() function, whose behaviour depends on the implementation of the child class. process() is a wrapper function that calls the pure virtual private function implementationMain() under the hood. It is the implementationMain() that must be overloaded by a child class whenever a new algorithm is developed. The wrapper automatically takes care of logging the processing times and marking errors. Any error thrown by implementationMain() is caught by process() and is appended with the three-letter identifier that was set during construction. This way, bug-tracking can be done very efficiently.

For the constructor method, a similar separation using a wrapper and a core function is advisable. Examples of this can be found in the source code files and the documentation in the appendix, as well as in section 2.4.1 on page 22.

Example SISO modules

To illustrate the use of SISO modules in particular and the framework in general, we have implemented a couple of modules that can be combined to form a complete face recognizer. For each stage, at least one module is implemented. They are:

- Detection: Viola-Jones face detector
- Registration: Viola-Jones eye-coordinate based registrator
- Illumination correction: Histogram equalization, Mask applier
- Feature extraction: Local Binary Pattern Histograms, Linear Discriminant Analysis
- Comparison: Chi^2, Euclidian distance, Likelihood ratio

The recognizer that can be composed using these modules is very basic and recognition results are poor compared to the state of the art. In chapter 4, where we study fusion, we will dive deeper into the underlying theories and performance of these modules. For now, we provide them as a proof-of-concept of our framework.

2.3.7 MIMO modules

There will be times where SISO modules will not fit the desired purpose because multiple inputs are required. This is, for instance, the case when metadata has to be provided in order to correctly process an image. To accommodate for this, the framework has an abstract base class called `Module` (of which SisoModule is a derived class) that takes care of only the very basics, thus discarding the SISO constraint. Classes derived from Module can be described as multiple-input-multiple-output (MIMO).

The Module base class poses no restrictions on the function descriptors in its derived classes, neither in number of inputs nor type of in- and output. By making clever use of pass-by-reference, the number of outputs is also unlimited. While this allows a great flexibility in implementing derived modules, this freedom comes at the cost of having non-standardized modules.

Keeping in mind that future researchers will most likely want to re-use already implemented modules, it is highly desirable that at least some form of standardization is pursued. So although the implementationMain() and initialize() functions are not mandatory for the Module class, we strongly recommend that a similar structure is maintained (whenever possible) when implementing Module derivatives.

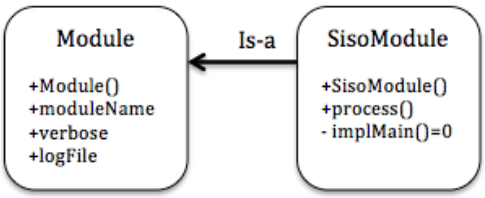


Figure 2.9: Relation between Module and SisoModule. It can readily be seen that SisoModule adds extra standardization to Module.

2.4 Using the framework to test a face recognizer

In this section, we will describe how the framework can be used. In section 2.4.1, we describe how newly developed algorithms can be implemented as modules. Then, in section 2.4.2, we give some guide rules for uniformly-styled documenting of the newly created code. Lastly, in section 2.4.3, we will show how the framework can be used to test an implemented algorithm. For both, sample code fragments will be given to illustrate the usage.

2.4.1 Implementing new modules

Before implementing a new algorithm into the framework as a module, it is important that a few design considerations are made.

1. Does the module comply with any of the following types: detector, registrator, illumination corrector, feature extractor, comparator. If one of these labels can be applied, the new modules should be implemented in the corresponding subnamespace of `utbpr` (e.g. `utbpr::featExtractor`).

2. Can the module be considered single-input single-output during testing? If so, the base class for the new module should be SisoModule. In all other cases, the MIMO-style Module base class must be used.

3. If the module can be considered as SISO *and* complies to one of the five aforementioned types, the new modules can be made a child class of that corresponding type instead of inheriting directly from SisoModule
   (e.g. `utbpr::featExtractor::FeatExtractor)`.

Once these considerations have been made, the actual implementation can be made.

To introduce the matter we will now assume that we want to implement a feature extractor for linear discriminant analysis (LDA)[7]. LDA is a typical example of a SISO system: an image can be transformed into LDA space without the need for extra information about the image. An LDA system requires a transformation matrix and mean image to be known, but as this data is equal for all images that will be processed it can be set beforehand (during construction).

So, to create a new feature extractor module for the LDA the following code fragment is added to the featExtractor.h header file:

```
namespace utbpr{
namespace featExtractor{
  class LDA: public FeatExtractor
    {
    ...
    }
}}
```

As the LDA requires the presence of an LDA transformation matrix and a mean image, we provide two global variables for this inside the class:

```
  //Global variables
  cv::Mat T; //LDA transformation matrix
  cv::Mat M; //Mean image of training set before transformation
```

These variables can be set during the construction. This is done in a newly created file bearing the name of the module (e.g. LDA.cpp):

```
LDA::LDA(int screenVerbosity, FILE* logFilePtr, cv::Mat
   ldaTransformMatrix, cv::Mat meanImage)
  : FeatExtractor("LDA module", screenVerbosity, logFilePtr)
  {
  T = ldaTransformationMatrix;
  M = meanImage;
  }
```

Since this module is derived from FeatExtractor (which in turn is derived from SisoModule) the constructor has to call *its* constructor as well. Therefore, the first two arguments of the LDA constructor are mandatory and passed to FeatExtractor. There are no restrictions on all extra arguments. It is recommended to also implement a time monitor for future reference. This is shown in the code below:

```
LDA::LDA(int screenVerbosity, FILE* logFilePtr, cv::Mat
   ldaTransformMatrix, cv::Mat meanImage)
  : FeatExtractor("LDA module", screenVerbosity, logFilePtr)
  {
  int64 timeStart = cv::getTickCount();
  T = ldaTransformationMatrix;
  M = meanImage;
  int64 timeStop = cv::getTickCount();
  int timeElapsed = (int)((timeStop-timeStart)/cv::getTickFrequency()
     *1000);
  if(logFile)
    fprintf(logFile,"\nInitializing time (module %s): %i[ms].",
    getModuleName().c_str(),timeElapsed);
  }
```

However, as we have stated in section 2.3.6, it would be better if the constructor call is separated from the implementation, as this gives a clearer view of what is being initialized exactly, especially when initializing comprises more than two lines of code:

```
LDA::LDA(int screenVerbosity, FILE* logFilePtr, cv::Mat
    ldaTransformMatrix, cv::Mat meanImage)
  : FeatExtractor("LDA module", screenVerbosity, logFilePtr)
  {
  int64 timeStart = cv::getTickCount();
  initialize(&ldaTransformMatrix, &meanImage);
  int64 timeStop = cv::getTickCount();
  int timeElapsed = (int)((timeStop-timeStart)/cv::getTickFrequency()
      *1000);
  if(logFile)
    fprintf(logFile,"\nInitializing time (module %s): %i[ms].",
       getModuleName().c_str(),timeElapsed);
  }

LDA::initialize(cv::Mat* ldaTransformMatrix, cv::Mat* meanImage)
  {
  T = *ldaTransformationMatrix;
  M = *meanImage;
  }
```

From this, it follows that for each constructor form, there will be one corresponding form of the initialize() method.

Now that the constructing is done, we can focus on the actual image processing function. To process an image in any module, the process() function from parent class SisoModule is called. This function is non-virtual and thus cannot be overloaded, but, as stated in section 2.3.6, process() calls implementationMain(), which *is* a virtual function:

```
cv::Mat SisoModule::process(cv::Mat in)
  {
  implementationMain(&in, &out);
  return out;
  }
```

As with the constructor, the process() function also takes care of timing and error reporting aspects (not shown here for simplicity. Details can be found in the documentation appendix).

For the LDA module, the implementationMain() is fairly straightforward:

```
LDA::implementationMain(cv::Mat* inPtr, cv::Mat* outPtr)
  {
  if(M.rows != inPtr->rows * inPtr->cols)
    throw(std::runtime_error("Size mismatch."));
  *outPtr = T * (inPtr->reshape(0,M.rows) - M);
  }
```

It is important to note the difference between the pass-by-value call to process() and the pass-by-reference call to implementationMain(). The reader is advised to read up on pointers if this might not be clear. furthermore, we have shown how a size-mismatch error is detected and reported by throwing a runtime error (see section 2.3.1 for details).

Figure 2.10 depicts an overview of the inheritance of the implemented LDA class.

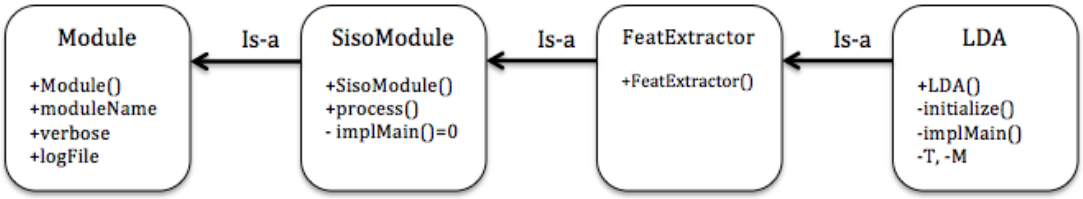


Figure 2.10: The inheritance overview of the LDA module. It can be seen that LDA inherits the process() function from Sisomodule, while overloading its implementationMain().

In essence, this is all that is needed to create a module inside the framework. It must be noted, however, that LDA requires training before it can be used to process images. This is not handled here, as training depends heavily on the type of module and is therefore not suitable to be generally exemplified. The way training is implemented is bounded by the code author's imagination only. Extra functions may be added to the module for this[4]. However, we recommend doing prior training on a high-end mainframe computer, saving the training outcome to file (or files) which can be read during construction of the module. For example:

```
LDA::initialize(char* ldaTrainFilePath)
  {
  char s[1000];
  sprintf(s,"%s_T.ascii,ldaTrainFilePath");
  T = FileIO::readAscii(s);
  sprintf(s,"%s_meanIm.ascii,ldaTrainFilePath");
  M = FileIO::readAscii(s);
  }
```

Here, files ldaTrain_T.ascii and ldaTrain_meanIm.ascii (containing training data) should be present in the current working directory if `initialize("ldaTrain")` is called.

[4]Recall that with SisoModule, the SISO-constraint applies only to the test phase.

2.4.2 Documenting new functionality

As the face recognition framework is going to be used by future researchers, it is important that the code is thoroughly documented. Every aspect of any newly created function should be described by the author for future reference. To ensure that the documentation remains uniform and extensive, we provide a guideline here.

- Documentation is done in-line by using Doxygen. Using the corresponding website[8] and by looking at existing code, the syntax for this is easily mastered.
- A doxygen configuration file (Doxyfile) is provided with the source code.
- Doxygen documentation is written in header files (.h) only. In implementation files (.cpp), plain C-style comments are used instead.
- Both the **how** and **why** of each piece of code code are described in the header, as this will give future developers an insight in why we did what we did the way we did.
- For all items, a one-line description of its purpose is given using the `@brief` command. Also, the author's name and the date of the most recent modification are documented (`@author`, `@date`).
- For every file, the contents are described using the `@file` command.
- For every class, the purpose is described and, if applicable, a reference to an affiliated paper or website is documented.
- For every function, its goal as well as all parameters and possibly a return value are documented. Furthermore, if certain preconditions have to be met, those are described in detail as well.
- In implementation files, each step in the process is accompanied by comments. A ratio of 1 line of comments for every 3 lines of code is common. Special attention is given to the documentation of for- and while-loops.
- Meaningful names for variables are used (e.g. transformationMatrix, inputImage, imIn, timeStart instead of just M,X,im,ts). Alternatively, the purpose of variables is described in a comment at declaration.

After modifications to the documentation has been made, `doxygen` is executed using the configuration file Doxyfile (see appendix A). This updates the documentation in both the LaTeX and http environment outputs.

2.4.3 Setting up a large scale test

Once new modules have been implemented (and debugged), a face recognizer can be set up using the newly created module. A face recognizer is in essence no more than a cascade of modules (recall the PFES, explained in section 2.2.1), combined with a comparator.

Initializing a PFES

To represent a PFES in C-code, we start by constructing objects of the desired modules from the `main()` program[5].

As we can only assume the modules are derived from Module (as opposed to SisoModule), we cannot make any assumptions regarding the module's interfaces. Therefore, we use a function-oriented approach instead of a object-oriented one for the main(). While this provides more flexibility for module interfacing, the downside to this approach is that the standardization is partly voided and the user must take care of the correct order of function calls himself. As we are developing the framework for research purposes, where flexibility outweighs ease-of-use, the function-oriented approach is preferred.

As an example, we assume a cascade of four modules is needed for the preprocessing:

```
utbpr::detector::ViolaJones det(...);
utbpr::registrator::ImprovedEyeFinder reg(...);
utbpr::illuminator::HistEq ill(...);
utbpr::featExtractor::LDA fex(...);
```

The arguments required for initialization of the modules are not shown her for simplicity: details can be found in the documentation. To cascade these modules in a function-oriented style, we call them sequentially:

```
std::vector<cv::Mat> getFeature(cv::Mat inputImage)
  {
  cv::Mat imDet = det.process(inputImage);
  cv::Mat imReg = reg.process(imDet);
  cv::Mat imIll = ill.process(imReg);
  cv::Mat featVector = fex.process(imIll);

  std::vector featureSet;
  featureSet.push_back(featVector);
  return featureSet;
  }
```

As can de deduced from the function prototypes, these modules are all SISO and thus getFeature could be as well, but we stress again that this may not always be the case and,

[5]The complete example of the main file for large scale testing can be found in appendix B.

therefore, we use the generalized vector form as the return format. This above block of code can be regarded as a PFES that processes one image into one feature vector set.

Enrolment

For large scale testing, a bulk of test images is required that all have to be independently processed by the PFES and then stored in a database. These two objects are instantiated like this:

```
//Open a database file
char* dbFilePath = "C:/resource/database.db";
utbpr::Database db(dbFilePath,'w');

//Create an image feed
std::vector<std::string> targetLocations = ...;
std::vector<unsigned int> targetsIds = ...;
utbpr::ImageReader imFeed(targetLocations,targetIds);
```

Opening a database file for writing is straight-forward. A destination file is specified, as well as a mode, which can be either (r)ead or (w)rite. In write mode, any new record that is presented for storage is appended to the existing database file.

The image feed requires a little more work to instantiate. In its most pure form, the ImageReader class requires two lists: one containing the target image locations on disk, and one containing the corresponding subject ID specifiers. How these lists are filled depends on the used database. However, as we have chosen to use the FRGC as our primary image source, we have implemented a direct method for reading in those signature sets:

```
//Create an image feed
char* imageListPath = "C:/resource/frgc_sigset.txt";
utbpr::ImageReader imFeed(imageListPath, true);
```

With the ImageReader, PFES and Database ready, it is possible to start the enrolment phase. First we create the appropriate placeholders and let ImageReader fill them by reference:

```
cv::Mat inputImage;
unsigned int inputId;
std::string inputPath;
bool neof = imFeed.getNextImage(&inputImage,&inputId,&inputPath);
```

Then, we enter a while loop, that continues as long as the end of the image list is not reached (no-end-of-file or neof). For each image, the feature is extracted. This feature set is stored in the database and a new image is loaded from the ImageReader:

```
while(neof)
  {
  std::vector<cv::Mat> featureSet = getFeature(inputImage);
  db.storeFeature(inputId,featureSet,inputPath);
  neof = imFeed.getNextImage(&inputImage,&inputId,&inputPath);
  }
```

Now, as we want the enrolment to continue even if a certain image could not be processed (e.g. one or both eyes were undetectable during registration) we wrap it in a try-catch block (see section 2.3.1). This ensures that when an arbitrary exception is thrown during enrolment, the exception is displayed to the user and the while-loop automatically continues with the next image:

```
while(neof){
  try{
    std::vector<cv::Mat> featureSet = getFeature(inputImage);
    db.storeFeature(inputId,featureSet,inputPath);
    neof = imFeed.getNextImage(&inputImage,&inputId,&inputPath);
  }catch(std::exception &errMsg){
    printf("Error: %s.",errMsg.what());
    neof = imFeed.getNextImage(&inputImage,&inputId,&inputPath);
    continue;
  }
}
```

Testing

To start the testing phase we first need to initialize two more objects: a Comparator and a ScoreMatrix.

The comparator is considered part of the face recognition system, and thus the choice for a comparator depends on the implementation of the PFES[6]. For details of the syntaxes, we refer the reader to the appendix.

```
utbpr::comparator::LdaLikelihood com(...);
```

To create a Scorematrix we an output path is required where the scores will be stored. Furthermore, ScoreMatrix needs to be told whether the scores will represent similarity scores (true) or dissimilarity scores (false):

```
char* scoreOutputPath = "C:\output\frgc1_lda_test";
utbpr::ScoreMatrix scm(outputPath,true);
```

Depending on the exact goal of the test it is possible to either extract the query feature sets from a query image set or, if we are dealing with all-vs-all matching, we can re-use the target database as query data. We will assume this last option and load the database to RAM to increase the test speed:

```
//Ready databases for RAM storage
db.changeMode('r');
utbpr::DatabaseRAM dbRAM;

//Create placeholders in memory and get first the record
unsigned int subjectId;
std::vector<cv::Mat> featVector;
bool neodb = db.getNextFeature(&subjectId,&featVector);

//While not-end-of-database, load each record to RAM memory
while(neodb){
  dbRAM.storeFeature(subjectId,featVector);
  neodb = db.getNextFeature(&subjectId,&featVector);
}
```

[6]Here, we use an likelihood ratio classifier that is trained using the same data as the LDA module we constructed in section 2.4.1.

Using the DatabaseRAM variant has the added benefit that it has two sequential accessors instead of one. `getNextTargetRecord()` for targets and `getNextQueryRecord()` for queries. This has the benefit that the records have to be stored only once, thus saving valuable RAM space. To use this architecture to do the all-vs-all experiment, we use the following code:

```
unsigned int queryId, targetId;
std::vector<cv::Mat> queryFeatVector, targetFeatVector;
bool targetsIdsSet = false;
bool neoql = dbRAM.getNextQueryFeature(&queryId,&queryFeatVector);

//While not-end-of-query-list, match query to all targets
while(neoql){
  scm.newQueryId(queryId);
  bool neotl = dbRAM.getNextTargetFeature(&targetId,&targetFeatVector);
  //While not-end-of-target-list, match query to this target and store
     score
  while(neotl){
    if(!targetsIdsSet)
      scm.newTargetId(targetId);
    double score = com.process(targetFeatVector[0],queryFeatVector[0]);
    scm.setScore(score);
    neotl = dbRAM.getNextTargetFeature(&targetId,&targetFeatVector);
  }
  targetIdsSet = true;
  neoql = dbRAM.getNextQueryFeature(&queryId,&queryFeatVector);
}
```

Here, it is important to notice that DatabaseRAM will automatically rewind its stack pointer when a getNextFeature()-call returns false. Furthermore, it can be seen that the query and targets IDs are stored in two separate lists within the ScoreMatrix. Although this might seem redundant for the all-vs-all experiment, we want to show its use for other possible use-cases.

Data interpretation

The procedure described in the previous section shows how to get a score matrix for an all-vs-all experiment. From this, the performance measures for the face recognizer can be determined using statistical analysis. However, such functionality has not yet been implemented in the framework due to time constraints. A note on this is made in the documentation as well as in the recommendations section of this report.

Chapter 3

Validation of the framework

In this chapter we will present a short recap on the system's specifications from section 2.1. We will repeat them and discuss them briefly to check whether they were met by the implemented framework and recognizer.

3.1 Requirements check

We develop a framework for face recognition experiments. The framework standardizes experiments and the recognizer's I/O. ✓ The framework, as described in detail in the previous sections, provides full capabilities for future face recognition experiments, while standardizing the input and output of the system and modules.

We develop a recognizer that can perform automatic face recognition on single still images. ✓ In section 2.3.6, we mention the implementation of two feature extraction modules, one based on Local Binary Pattern Histograms and one on Linear Discriminant Analysis. Together with the preprocessing modules from the same section, these are capable of performing face recognition on single still images. However, performance figures indicate that these recognizers are not worthy competitors for the current state-of-the-art.

The software can run on either Windows or Unix-based machines. ?
During construction, we have paid special attention to this aspect by bypassing any piece of code that is (or might be) platform specific. For low-level disk operations we have also taken into account little/big-endian differences. It must be noted, however, that platform independence has not actually been tested due to limited time being available.

Recognizer functionality is implemented as modules, which can be easily substituted with improved versions. ✓ As described in section 2.3.6, the modules architecture is implemented according to this specification.

The framework provides enough flexibility to implement the majority of future face recognition algorithms. This means that the chosen architecture may not impose great restrictions on module implementations. ✓ Although we have focussed our framework around single-input-single-output modules, the framework is suited to handle multiple-input-multiple-output as well, ensuring the required flexibility. More on this can be found in section 2.3.7.

More complex recognizers, such as fusion algorithms, can be implemented. ✓ By making use of multiple-inputs-multiple-outputs modules together with the function-oriented main program, the complexity of the algorithms is virtually boundless. This is illustrated in-depth in the following chapter, where we research the gain of using a multiple algorithm fusion recognizer.

Recognizer modules can be trained on an external machine, separately from the enrolment and verification/identification experiments. ✓ By making use of training files it is possible to train recognizer modules on an external machine. Such files can be loaded quickly during experiments. This concept is exemplified on page 25.

The framework is well documented, since future researchers have to work with it without much hassle. ✓ Besides in this report, the proposed framework is documented thoroughly using the code documenting tool Doxygen. As this documentation is written in source files instead of a separate document, it is easy to document new modules' functionality and changes to existing ones. By making documenting code easy, we hope to encourage future contributors to keep the framework well documented as well. How to keep the documentation organized is described in section 2.4.2.

In order to compare recognizers against the state of the art, we use the Face Recognition Grand Challenge tests. The framework should thus be able to handle those datasets. ✓ In section 2.3.3, we have described how images can be loaded from disk using the imageReader. This class is specially equipped with a function to handle the FRGC signature sets, albeit in *.txt format instead of the standard *.xml format.

Every recognizer under test, whatever the implementation, should yield results in the same form to accommodate an effortless comparison. ✓ By introducing the ScoreMatrix class, we have effectively standardized the output of any face recognizer that is implemented using the framework. However, data interpretation functions for this standardized output are not present in this development iteration (see the note on page 31 for details).

The framework is able to capture camera stills and enrol/compare them to a gallery database and display match results. ✓ The framework carries all functionality to implement this feature. We refer to the Camera class from section 2.3.3, and the Database class from section 2.3.4.

3.2 Objectives check

Standardize research

Looking at the requirements check, it seems safe to conclude that the framework is indeed capable of standardization of future research. The framework provides easy access to earlier implemented algorithms through its module structure, and has a special interface to use the FRGC experiments for benchmarking. Together, these make that comparison of research results to those of other recognizers can be done without much hassle.

Although, we did not implement a recognizer that is capable of competing with the current state-of-the-art in face recognition, we *did* create an environment in which upgrading the functionality is remarkably simple. Therefore, we dare to say that the proposed framework is a good first step towards achieving the state-of-the-art status.

Demonstrate capabilities

All that is needed for live demonstration of current recognizer capabilities – besides a properly trained recognizer – is a camera and a target database. As the framework contains interfaces for both, this objective could be said to be accomplished. Unfortunately, the one thing that is missing is the actual target database data. Although the software is virtually ready to be used for enrolment of known subjects, at the time of writing this has not yet been done.

3.3 Validation conclusion

Given that all the discussed requirements and objectives have either been met, or can be met easily within a short period of time, we conclude that the framework is ready to be used as the basis for future face recognition research within our research group.

Chapter 4

Fusion

For any classification problem there are multiple algorithms available. Each of them has its advantages and drawbacks, and these differ from algorithm to algorithm. Given the parameters of the posed classification problem, one of the algorithms may be tested to perform better than any of the others. By fine-tuning the parameters of the classifier algorithms the performance can be tweaked even further. The downside of such an approach is that the choice of the algorithms – and the optimal parameter values with them – depends heavily on the used training and validation sets. Furthermore, it might be the case that even the best performing algorithm is still not accurate enough. However, based on the 'two know more than one'-principle, we may be able to combine the advantages of multiple algorithms, while minimizing the drawbacks of both. This is called fusion.

In order to implement a fusion classifier, basically two question need to be answered[9]:

1. How do we generate base classifiers that complement each other?
2. How do we combine the outputs of those base classifiers for maximum accuracy?

We will investigate both questions in the next sections. We start out by taking a look at different ways to generate base classifiers. Then, we will discuss the classifiers that were chosen in more detail and take a look at their individual performances. This answers the first question. For the second question, we will dive into the styles of classifier combinations and their influence on the performance. Finally, we will show the overall fusion classifier performance figures and discuss the applicability of fusion as a performance enhancer.

4.1 Base classifiers

Classifier fusion requires multiple base classifier algorithms to be trained. To do this, the following theories might be used:

Parameter differentiation One classifier is trained using different training parameters. For instance, a Principal Component Analysis classifier can be trained using 50, 100 and 200 components, resulting in three different classifiers.

Training set differentiation One classifier is trained using different training sets. The training sets can be filled randomly, but it makes more sense to fill each subsequent set in such a way that there will be more emphasis on difficultly classifiable subjects.

Algorithm differentiation Multiple, independent classifier algorithms are used. Usually, each is trained on the same training set, but this need not be the case.

As we are interested in classifiers that complement each other as much as possible, the most promising combinations of classifiers are those whose base classifiers are (as much as possible) independent of each other i.e. operate using unrelated principles. In this light, we expect the best results from Algorithm differentiation. Moreover, while using multiple algorithm *might* increase the overall accuracy, it *will* increase the processing time. Therefore, we want simple, fast algorithms over complex but accurate ones.
Choosing from the vast assortment of available algorithms that fit these two criteria, we select Local Binary Patterns[10] and Linear Discriminant Analysis[7] to do our research with. For both we present a summary of their theories on the next pages.

4.1.1 Local Binary Patterns

Source: [10].

The Local Binary Pattern algorithm describes the texture and shape information of a face, rather than specifying specific landmark properties. The face image is first divided into small regions from which local binary pattern (LBP) features are extracted and concatenated into a single feature histogram, efficiently representing the face image. The textures of the facial regions are locally encoded by the LBP patterns while the whole shape of the face is recovered by the construction of the face feature histogram. The idea is that the face can be seen a composition of micro-patterns which are invariant with respect to monotonic grey scale transformations (i.e. illumination differences). Combining these micro-patterns, a global description of the face is obtained.

In practise, we divide the image into 49 regions (7x7 grid). In each region, we determine the LBP for each pixel. This is done by comparing them to each of the surrounding 8 pixels. If the surrounding pixel is brighter, we notate a '1', otherwise we notate '0'. This results in a binary string of 8 bits, which can be interpreted as a decimal integer[1]. We calculate a histogram over the selected region and optimize it by compressing non-unitary patterns into one bin (a pattern is called uniform if it contains at most two bitwise transitions from 0 to 1 or vice versa when the string is considered singular). Finally, the compressed histograms of all regions are concatenated to form the representing feature vector.

The feature vectors resulting from the LBP algorithm can now be compared efficiently using a Chi^2 statistics test.

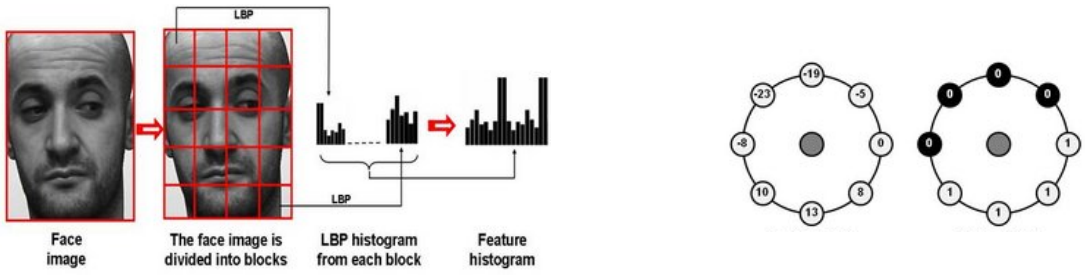


Figure 4.1: Graphical representation of the LBP algortihm. Left: an image is divided in regions. Each yields a histogram, which are concatenated. Right: each pixel is thresholded against the center pixel to give a string of bits. Source:[11].

[1]the number of surrounding pixels, as well as the radius at which these points lie can be fine-tuned to optimize the performance. The grid size might also be adjusted.

4.1.2 Linear Discriminant Analysis

Source:[9].

Linear Discriminant Analysis (LDA) uses training instances to see how they can best be separated, given a class grouping. Given a training set with multiple images of each class (subject), LDA searches for base vectors in an underlying space that best discriminate among the classes. In formulas: LDA maximizes the function J by fine-tuning w:

$$J(w) = \frac{w^T \cdot S_B \cdot w}{w^T \cdot S_W \cdot w}$$

where

$$S_B = \sum_{j=1}^{m} (\bar{x}_j - \bar{x})(\bar{x}_j - \bar{x})^T, \quad S_W = \sum_{j=1}^{m} \sum_{i=1}^{N_i} (x_i^j - \bar{x})(x_i^j - \bar{x})^T$$

represent the between-class and within-class scatter matrix, S_B and S_W, respectively. Here, m is the number of classes in the training set, $\bar{x}_j$ is the mean of class i, $\bar{x}$ is the mean of all classes, N_i is the number of samples in class i and x_i^j is the ith sample of class j.
Because S_W is usually singular (by the curse of dimensionality) the dimensionality is first reduced using a PCA signal representation. In this subspace, the LDA performs a further dimensionality reduction that takes the classification problem into account. Using only PCA would result in a feature space that separates the input signals instead of one that best distinguishes among the classes.
After the LDA transformation matrix w has been determined, the feature vector representation V is calculated for any input I using

$$V = w \times (I - \mu)^T$$

where I is a row-vectorized image and μ is the mean image of the training set. The resulting feature vector V is then compared to the database using a likelihood ratio comparator.

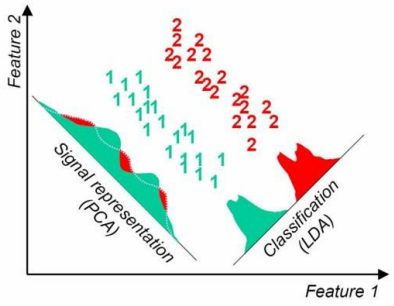
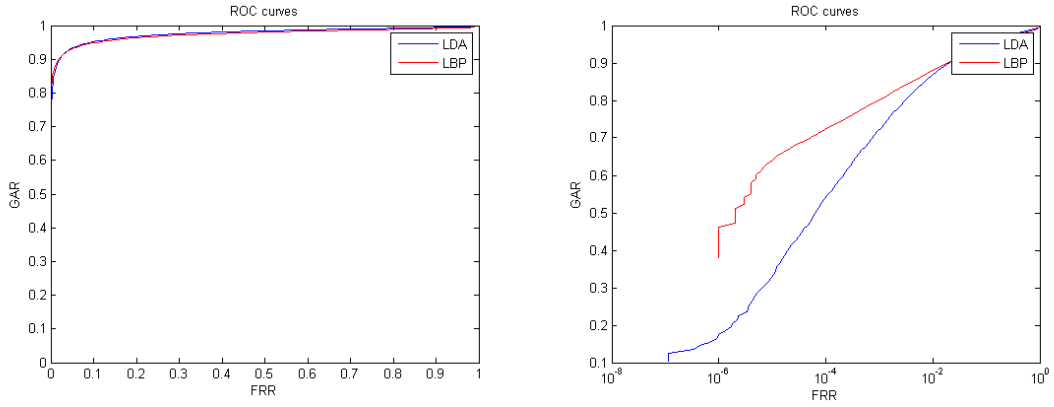


Figure 4.2: Graphical representation of the LDA algorithm for two classes. PCA projects along the axis of largest variance, while LDA also takes the classes into account[12].

4.1.3 Base classifier performance

To get a baseline measure, we test the performance of the base classifiers. For this, we use a subset of the FRGC database to reduce simulation times of the all-vs-all experiments. For training, we use a subset of the FRGC experiment 1 training set, where we used 10 images of each person in the database (where possible), resulting in 2220 images. For validation, we have randomly selected 4229 subjects from the FRGC target set. This validation set was registered automatically, whereas the training set uses manually aligned images.



(a) Linear scale **(b)** Semi-log scale

Figure 4.3: ROC curves of the LBP and LDA classifiers.

In figure 4.3 we show the ROC curves of both the LDA and LBP algorithms. As both graphs follow a similar curve we printed a log-scaled figure to show the differences. Furthermore, in the table below we show the points of interest for both base classifiers.

	EER (%)	VR@0.1%FAR (%)	VR@0.01%FAR (%)
LBP	6.3	88.2	80.2
LDA	6.1	87.7	72.6

Table 4.1: Base classifiers' performance figures.

From the table it can easily be seen that the performance does not significantly differ between the two classifiers, although LBP performs slightly better. Also, neither classifier performs extremely well.

However, as stated in the beginning of this chapter, our aim is not to optimize the base classifiers. Instead, we strive to optimize the performance by combining the results of the base classifiers using an appropriate fusion algorithm. Therefore, the shown results indicate that the chosen classifiers are particularly suited for our research goal.

4.1.4 Score normalization

The two classifiers described above both output a floating point score, but their ranges differ which complicates the score fusion. We therefore opt to normalize both scores before the fusion takes place using Z-score normalization on the imposter scores[13]. This is a linear normalization that is especially suited for Gaussian distributions. Since imposter scores can be considered as random variables (as they are the result of matching unrelated images) and the experiment is done on a very large scale, the Central Limit Theorem predicts that the distribution will be normal (by approximation). We determine the Z-score normalization parameters from a subset of the imposter validation data and do not use this data during interpretation to avoid mixing training and validation data. After normalization, the LDA and LBP scores are shown as a scatter plot in figure 4.4.

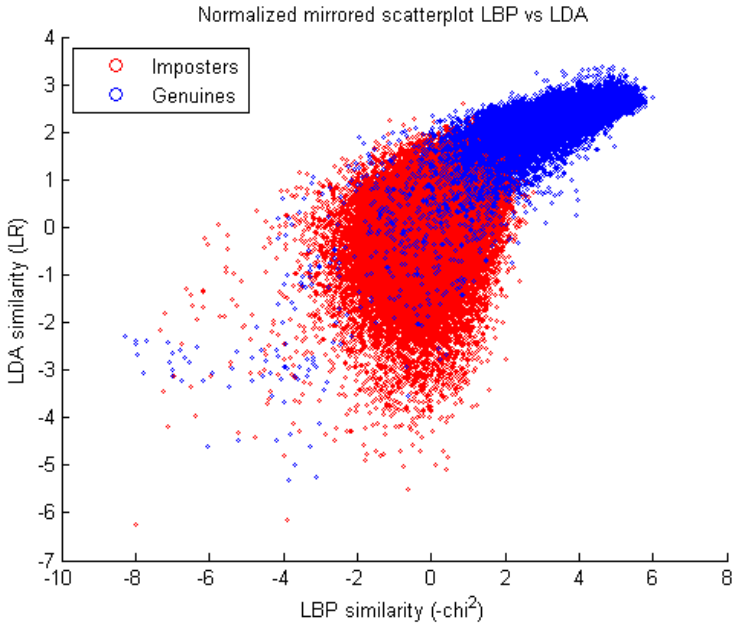


Figure 4.4: Scatter plot of LBP vs LDA. The LBP scores have been mirrored in $x = 0$ to represent similarity scores.

It can be seen from the figure that a line with an approximate angle of -45 degrees can better separate the two coloured clouds than a straight horizontal line (LDA only classifier) or straight vertical line (LBP only classifier). This is an indication that score fusion using the sum rule is appropriate. We will investigate this in the next section. Furthermore, although most of the genuine scores are located in the upper right-hand corner, there are a lot of outliers that score lower than certain imposter scores. This is, of course, unwanted but cannot be circumvented by using score fusion. We will come back to this in the Discussion section.

4.2 Fusion forms

There are many forms of fusion. We give a brief overview of the possibilities and then continue with the in-depth description of selected method.

Algorithm selection Multiple classifiers are trained but (based on the input's parameters) only one is selected to do the feature extraction and comparison. This requires an extra module that takes care of the selection.

Algorithm fusion I Multiple base classifier algorithms are implemented and each uses the *same* sensor information to find multiple representations in different feature spaces.

Algorithm fusion II Multiple base classifier algorithms are implemented and each uses information from a *different* sensor to find multiple representations in different feature spaces.

Our research focusses on FRGC experiment 1, which deals with single controlled 2D still images. This means that we must discard Algorithm fusion type II as option since this assumes the availability of multiple inputs. This leaves the choice for either Algorithm selection or Algorithm fusion type I. We choose the latter, purely for its relative simplicity.

As Algorithm fusion produces more than one feature vector, this type of fusion is always accompanied by one of the following combination approaches:

Feature fusion The features are concatenated and the resulting larger vector is compared to the database by one comparator.

Score fusion Each feature representation is compared independently from the other representations, and the scores of each comparator are combined.

Decision fusion After comparison, each score is thresholded to a match/non-match boolean. These booleans are then combined to a single output.

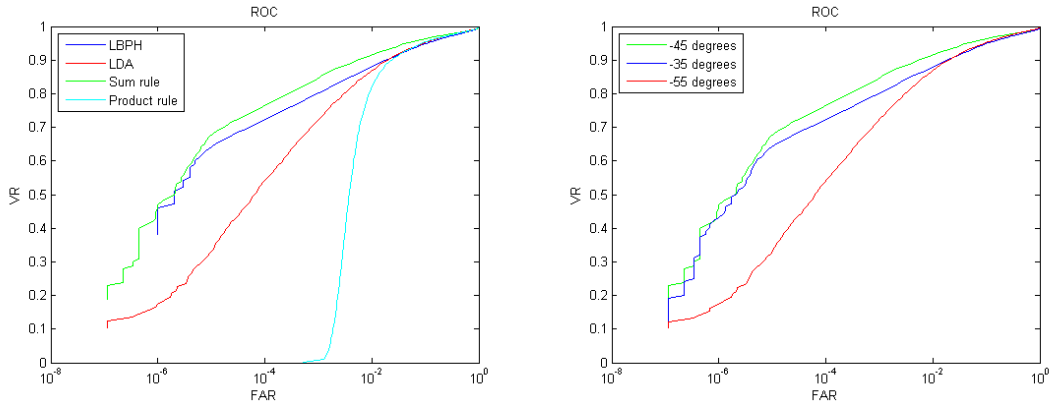
In section 2.2.1, we have argued that the comparator can be considered as an integral part of a face recognizer. As the LBP and LDA algorithms have very different types of feature representations (and thus comparators) we cannot use only one comparator for both. This implies that Feature fusion must be discarded. In addition, Decision fusion can be considered as a non-linear type of Score fusion and so we select Score fusion as our research focus.

4.2.1 Score fusion

For the fusion of the normalized scores, we use the relatively simple Product, Sum and Weighed sum rules:

$$\begin{aligned} S_{product} &= S_{LDA} \cdot S_{LBP} \\ S_{sum} &= S_{LDA} + S_{LBP} \\ S_{weighedsum} &= S_{LDA} + a \cdot S_{LBP} \end{aligned}$$

For each rule we generate a new score matrix and, from these, we determine the performance figures and the ROC curves. As the weighed sum rule requires the optimization of parameter a, we determine the ROC for three values that seem appropriate from the scatter plot: $a = 2/3$, $a = 1$ and $a = 3/2$ (which corresponds to threshold boundaries of approximately -35 degrees, -45 degrees and -55 degrees in the scatter plot of figure 4.4). We show the results in figure 4.5 and table 4.2 on the following page.



(a) Effect of using sum rule and product rule fusion.

(b) Effect of using different weights for the weighed sum fusion

Figure 4.5: ROC curves of the fusion classifiers.

The first thing that is apparent is that the performance is negatively affected by using the Product rule. Although EER scores do not differ very much, the VR@FAR scores are significantly lower than those of each separate base classifier. The Sum rule performs the best, with both EER and VR@FAR scores improved over the base classifiers. Also in comparison to the Weighed sum rule performance, the standard, balanced sum rule performs the best.

Fusion	EER (%)	VR@0.1%FAR (%)	VR@0.01%FAR (%)
LBP	6.3	88.2	80.2
LDA	6.1	87.7	72.6
Product	6.1	82.4	1.0
Sum	4.8	92.2	85.1
Weighed ($a = 2/3$)	6.4	88.4	80.4
Weighed ($a = 3/2$)	6.1	87.6	72.6

Table 4.2: Performance figures for the fusion classifiers.

4.3 Discussion

From the results we can see that fusion can have a positive effect on the face recognition performance. We have successfully combined two base classifiers by summing the normalized output. This resulted in a slight, but significant increase in performance figures. Other methods – Product rule and Weighed sum rule fusion – were less successful. In this section, we will discuss how the performance can be optimized even further.

4.3.1 Product rule

Looking at the results from figure 4.5, we see that Product rule fusion shows a rather large drop in verification rate for small FAR rates. Diving into the matter, we find that the cause for this effect is in the combination with the Z-score normalization. This normalization has shifted all points of the scatter clouds to around the origin (see figure 4.4 on page 40). Because double negatives give positive values in multiplication, it is easy to see that both the points from the first quadrant *and* those of from the third quadrant will produce positive scores. This results in an even greater overlap of imposter and genuine scores, which affects the performance severely. A solution to this is to shift all points to the first quadrant before multiplication. Doing this leads to the following performance figures:

Fusion	EER (%)	VR@0.1%FAR (%)	VR@0.01%FAR (%)
Product	4.9	92.1	85.0

Table 4.3: Performance figures for the corrected product fusion.

With this correction, the product rule fusion classifier performs about equally well as the sum fusion. For simplicity, we advise to use the Sum rule, as the Product rule requires an extra processing step for it to operate correctly.

4.3.2 Base classifier optimization

In section 4.1.4 on page 40, we mentioned that the normalized scatter plot (figure 4.4) shows a lot of outliers in the genuine scores. These outliers score low on both the LDA and LBP axis, indicating that the cause may not lie in the classifier algorithms but in the preprocessing step. To investigate this we examine the automatic detector and registrator modules.

The goal of the preprocessing step is to align all faces and correct the intensity of the lighting. The alignment is based on the position of the eyes, which are automatically detected using the Viola-Jones Haar cascade algorithm. If this not done accurately, the result of both the LDA en LBP classifiers is negatively influenced. As the Viola-Jones algorithm is known to have a good detection rate but a low accuracy, this might be a cause for the found outliers.

To test this hypothesis, we set up a manually registered query set *and* an automatic registered set that originate from the same underlying input images. Both are tested against a manually aligned target set. Using this data, we perform an LDA-only verification experiment and to minimize the influence of the LDA training for this experiment, we use the FRGC *training* set as the validation set. The results are shown in figure 4.6.

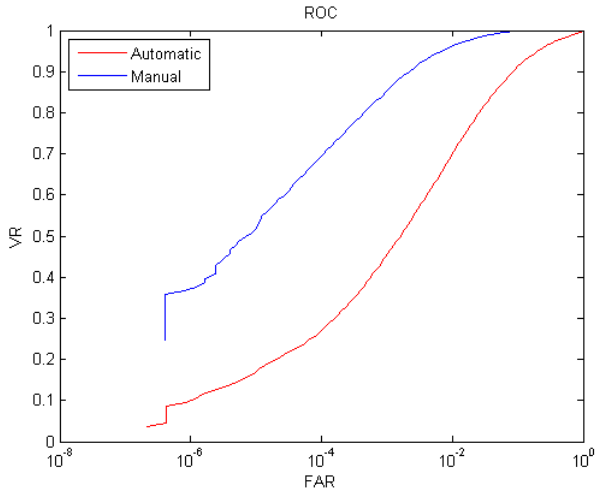


Figure 4.6: ROC curve of the manual and automatic registration experiments. It can be seen that the manual aligned images are recognized much better.

It can be seen from the figure that the automatic registrator has a rather large influence on the performance. A significant performance gain is feasible for the recognition system by improving the registration module. Therefore, we recommend that, in a future research, special attention is paid to the improvement of the automatic registration process.

Chapter 5

Conclusion

In this report, we have proposed a framework for the recognition of faces. The design is well documented in chapter 2 and appendix C. Also, the framework is tested to meet up with its specifications in chapter 3.

The resulting framework can be the basis for future studies in the field of face recognition, as it standardizes the way in which recognizer modules are implemented. We have shown how future researchers can use the framework to simplify their work and, more importantly, how *their* implemented modules can be stored and documented in a way that subsequent studies will benefit from their work.

We have further shown how the framework can be used to set up a complex, fusion, recognizer. From chapter 4 we have learned that, by using fusion, multiple base classifiers can be combined to form a better one, albeit that future research should probably focus more on creating better base modules than fancier fusion algorithms.

Although the currently implemented face recognizer modules do not meet up with the state-of-the-art in face recognition, the framework provides all the means to gradually improve this as each module can be separately developed and tested. We expect that, in a few years, newly created modules will be the basis for a contemporary competitor in the field of face recognition. To help achieve this goal we provide some recommendations for future work on the following page.

5.1 Recommendations for future work

These recommendations are sorted from most significant to least significant.

Improve the registration algorithm As we have seen in section 4.3, the automatic registrator module does perform very well. As the registration can be considered as the foundation for a face recognizer, we strongly suggest that this module be among the first to be improved.
As a workaround, a manually labelled and registered images can be a good solution. Therefore, we advise to keep a read-only copy of such a set next to the original FRGC database.

Implement XML parser for FRGC data The framework uses the FRGC as its main validation database. The FRGC originally provided the experiments setups in XML-formatted signature sets, but we have used derived TXT-formatted lists instead. To be able to really use the FRGC database, an XML input parser is necessary that can handle the signature sets directly.

Expand ImageReader to accept multi image input sets The FRGC provides experiments that consider multi-image input sets. The ImageReader class should thus be redefined to handle this aspect.

Implement score matrix data interpretation The current ScoreMatrix class only stores the outcomes of each recognition experiment. Data interpretation is then done outside the framework using, for instance, Matlab or the roc-tool. It would be more efficient if the interpretation of the data could be done directly from the framework, so that (among others) equal error rates, verification rates, rank-n scores and ROC-curves can be calculated for any experiment without further hassle.

Expand ScoreMatrix to hold unique (per-image) identifiers Currently, ScoreMatrix does not record any image identifiers which prevents that a distinction can be made between the different images of the same person during data interpretation. While in all-vs-all experiments this distinction can be deducted from the score's position in the matrix (on-diagonal or off-diagonal), this is not true for experiments in general. Therefore, it would be wise to keep track of the image numbers as well.

Implement database indexing function At the moment, the Database class can only be used to match a given query to *all* targets in the database. For verification purposes we would like to be able to retrieve only a single target record. Therefore, an indexing or 'search by target ID' function needs to be implemented.

Appendices

Appendix A

File and folder structure

The legacy of this project is added to this report as a disk. To help future researchers find their way around this disk, we will provide an overview of its contents here.

In the root, there are two files: `UTSurface.sln` and `UTSurface.ncb`. These are required by Microsoft Visual C Studio to set up a solution (=project).

Debug

After compilation of the source code, this folder will contain the compiled program.

Resources folder

This folder contains everything that either the framework or any of the modules need to work properly. The subfolders have names that correspond to the class name of the modules that uses them, as well as a three-letter prefix for the function of that class. Currently, the following resources are present:

- **db** Database files with target feature vectors of existing modules.
- **sigsets** Signature Files for the FRGC experiments 1 and 4 in TXT-format.
- **det.violajones** The Haar cascade wavelets for the ViolaJones detector.
- **reg.haarcascade** The Haar cascade wavelets for the HaarCascade *and* ImprovedEyeFinder registrators.
- **ill.mask** Mask files for the Mask applier.
- **fex.lda** Training files for the LDA feature extractor and LdaLikelihood comparator.
- **fex.pca** Training files for the PCA feature extractor.

Sfireader folder

This folder contains a tool to show SFI-formatted images. As the framework uses this as its standard output for images, and there is no tool available for reading such files from the internet, we have developed it ourselves. Please read the `readme.txt` file for more information.

Docs folder

This folder contains the full documentation of the project in both HTML and LaTeX forms. Open the HTML variant from the `/html/index.html`, or the LaTeX variant using `/latex/refman.pdf`.

UTSurface folder

This folder contains all source files for the program. Inside you will find the following files:

- **UTSurface.cpp** The main file of the program, where the program is started.
- **stdafx.cpp /.h** The precompiled header files for MS Visual Studio.
- **...** All other files are less relevant, but are required by MS Visual Studio.

UTSurface/debug folder

Contains all debug files that are created during compilation, except for the final executable, which is found in the Debug folder in the root directory.

UTSurface/docsrc folder

This folder contains the source files for the generation of the Doxygen documentation, including the Doxyfile configuration file for this.

UTSurface/modules folder

All the source files for modules that are designed for the framework are located in this folder. All the modules are declared in the *.h files of the appropriate stage in this folder. The *.cpp files are organized in subfolders.

UTSurface/supFunctions folder

All classes that are part of the framework but cannot be regarded as modules reside here.

Appendix B

Main file for large scale tests

This program compares a list of queries to an existing database of targets (not per se all-vs-all). It assumes that the query input images have been registered beforehand.

```
/** @file UTSurface.cpp
  * @brief Main. This is where the entry point for the console
  * application: main() is defined.
  */

//Include the precompiled header file to speed up compile times
#include "stdafx.h"

//This header can be included to trace memory leaks
#include "c:/dev/vld/include/vld.h"

//These include the header for the modules
#include "Modules/utbpr_module.h"
#include "Modules/utbpr_detector.h"
#include "Modules/utbpr_registrator.h"
#include "Modules/utbpr_illuminator.h"
#include "Modules/utbpr_featExtractor.h"
#include "Modules/utbpr_comparator.h"

//These are the headers for the supporting functions
#include "SupFunctions/utbpr_database.h"
#include "SupFunctions/utbpr_imfeed.h"


/** @brief The main entry point for the console application.
  * This is where it all begins.
  */
int main(int argc, char* argv[])
  {
```

```
std::cout << "\nWelcome to UTSurFace.\n";

//Start by encapsulating everything in a try-catch block
try
  {

  //Define the paths that are necessary for the modules
  char* sourceBasePath = "C:/dev/inputFolder";
  char* outputBasePath = "C:/dev/outputFolder";
  char* queryListPath = "C:/dev/resources/sigsets/frgc1_query.txt";
  char* ldaTrainFile = "C:/dev/resources/fex.lda/lda_90_128";
  char* logFilePath = "C:/dev/outputFolder/log.log";
  char* outputPath = "C:/dev/outputFolder/lda_exp";
  char* dbFilePath = "C:/dev/resources/db/frgc_exp1.db";

  //Set the verbosity level and create an logFile pointer
  int verbose = 1;
  FILE* logFile = utbpr::FileIO::openOutputFile(logFilePath,1);

  //Construct the modules of the face recognizer
  utbpr::illuminator::HistEq ill(verbose,logFile);
  utbpr::featExtractor::LDA fex(verbose,logFile,ldaTrainFile);
  utbpr::comparator::LdaLikelihood com(verbose,logFile,ldaTrainFile);

  //Create the score matrix and database containers
  utbpr::ScoreMatrix scm(outputPath,true,false);
  utbpr::Database dbonce(dbFilePath,'r');
  utbpr::DatabaseRAM db;

  //Create a list of subject IDs (based on the filename)
  std::vector<std::string> imageLst;
  imageLst = utbpr::FileIO::readListFromFile(targetListPath);
  std::vector<std::string> idList;
  unsigned int sz = imageLst.size();
  idList.resize(sz);
  printf("\n  Size: %i",sz);
  for(unsigned int i=0;i<sz;i++)
    idList[i] = imageLst[i].substr(imageLst[i].find_last_of("/")+1,5);

  //Load the filename list and subject IDs in the ImageReader
  utbpr::ImageReader imr(imageList,idList);
  imr.setBasePath(sourceBasePath);
  imr.setLogFile(logFile);

  //Create memory placeholders
  unsigned int subjectId;
  std::vector<cv::Mat> featVectors;
```

```
    //Load all records from the disk    database to the RAM database
    printf("\n   Read database to RAM");
    bool neof = dbonce.getNextFeature(&subjectId,&featVectors);
    //While not end of database file, do:
    while(neof)
      {
      db.storeFeature(subjectId,featVectors);
      neof = dbonce.getNextFeature(&subjectId,&featVectors);
      }

    printf("\n   #Database Records:%i",db.size());
    printf("\n   Done init.");

//End of initialization. Continuing with the actual experiment

    //Create memory placeholders
    unsigned int targetId, queryId, errors=0;
    unsigned int t=0, q=0;
    cv::Mat image, imIll, featVec;
    std::string imPath;
    std::vector<cv::Mat> dbFeatVec;
    bool dbRecordsAvailable;

    //Read in the first image
    neof = imr.getNextImage(&image,&queryId,&imPath);
    //While not all queries are read, do:
    while(neof)
      {
      //Catch errors caused by bad images using a try-catch block
      //inside the while-loop
      try
        {

        //Display progress info
        printf("\n%i: %s",q++,imPath.c_str());

        //Process the image to a feature vector
        imIll = ill.process(image);
        featVec = fex.process(imIll);

        //Record the query ID for the scoreMatrix
        scm.newQueryId(queryId);

        //Get the first target feature set from the database
        db.rewindFile();
        dbRecordsAvailable = db.getNextFeature(&targetId,&dbFeatVec);
```

```
        //While there are records in the database, do:
        while(dbRecordsAvailable)
          {
          //Compare the query to all targets and record the score
          double score = com.process(dbFeatVec[0].t(),featVec);
          scm.setScore(score);

          //On the first run, record the target IDs as well
          if(q==0)
            scm.setTargetId(targetId);

          //Retrieve the next database record
          dbRecordsAvailable = db.getNextFeature(&targetId,&dbFeatVec);
          }

        //Retrieve the next query image from the ImageReader
        neof = imr.getNextImage(&image,&queryId,&imPath);

        }
      catch(std::exception &errMsg)
        {
        //If something goes wrong during image processing, report the
            error
        //and continue with the next image
        printf("\n! Error at %s: %s.",imPath.c_str(),errMsg.what());
        errors++;
        neof = imr.getNextImage(&image,&queryId,&imPath);
        continue;
        }
      }//End of ImageReader while-loop

    //Report the number of errors
    printf("\n  nErrors:%i.",errors);

    }
  catch(std::exception &errMsg)
    {
    //If something goes wrong in general, report the error and terminate
    printf("\n! Error: %s",errMsg.what());
    }

  return -1;
  } //End of main file
```

The output consists of three files generated by the ScoreMatrix class: lda_exp_queryId.ascii lda_exp_targetId.ascii lda_exp_scoreMat.ascii in folder C:/dev/outputFolder/.

Appendix C

Full documentation

To save the environment it was decided that the full documentation will only be made available digitally. This is done because it counts well over 70 pages and, above that, will be constantly changing as the system evolves over following researches. Together with the documentation, the full source code will also be made available on the same disc. Please inquire at the group's administrator to access these records.

Bibliography

[1] Signals and Systems research group, "Face recognition on the move," University of Twente, 2012.

[2] P. J. Phillips, P. J. Flynn, T. Scruggs, K. W. Bowyer, J. Chang, K. Hoffman, J. Marques, J. Min, and W. Worek, "Overview of the face recognition grand challenge," in *Computer Vision and Pattern Recognition*, vol. 1. IEEE, 2005, pp. 947–954.

[3] OpenCV, "Open source computer vision," URL: http://opencv.org/, 2013.

[4] Mathworks, "Matlab coder project 2013," URL: http://www.mathworks.nl/products/matlab-coder/, 2013.

[5] D. S. Bolme, J. R. Beveridge, M. Teixeira, and B. A. Draper, "The csu face identification evaluation system: its purpose, features, and structure," in *Computer Vision Systems*. Springer, 2003, pp. 304–313.

[6] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[7] S. Mika, G. Ratsch, J. Weston, B. Scholkopf, and K. Mullers, "Fisher discriminant analysis with kernels," in *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop.* IEEE, 1999, pp. 41–48.

[8] "Doxygen code documenting tool," URL: http://www.doxygen.org, May 2013.

[9] E. Alpaydin, *Introduction to machine learning*. MIT Press, 2010.

[10] T. Ahonen, A. Hadid, and M. Pietikainen, "Face description with local binary patterns: Application to face recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 28, no. 12, pp. 2037–2041, 2006.

[11] Scholarpedia, "Local binary patterns," URL: http://www.scholarpedia.org/article/Local_Binary_Patterns, 2013.

[12] CSDN, "Linear discriminant analysis," URL: http://blog.csdn.net/ytlcainiao/article/details/8969772, 2013.

[13] A. Jain, K. Nandakumar, and A. Ross, "Score normalization in multimodal biometric systems," *Pattern recognition*, vol. 38, no. 12, pp. 2270–2285, 2005.

– END OF REPORT –

Thank you for reading :)