A Modular Framework to Detect and Analyze Faces for Audience Measurement Systems

Andreas Ernst, Tobias Ruf, Christian Kueblbeck
Fraunhofer Institute for Integrated Circuits IIS
Department Electronic Imaging
Am Wolfsmantel 33, 91058 Erlangen, Germany
{andreas.ernst, tobias.ruf, christian.kueblbeck}@iis.fraunhofer.de

Abstract: In this paper we describe an approach that enables the detection, tracking and fine analysis (classification of gender and facial expression) of faces using a single web camera. One focus of the paper lies in the description of the concept of a framework that was designed in order to create a flexible environment for varying detection tasks. We describe the functionality, the setup of the framework and we also give a coarse overview about the algorithms we are using for the classification tasks. Benchmark results are given on standard and publicly available data sets. Although the framework is designed for general object recognition tasks, our focus so far has been in the field of face detection and fine analysis. With the capabilities provided by the system we see one main application field in the area of digital signage where providers are enabled to gather information, measure audience focus or even create interactive advertisement solutions. The varying applications require different software features and the provided framework approach allows to easily create different recognition setups in order to fullfill these specific needs. We show in a demo application for still images and movies that our framework can be well used for these purposes. This can be downloaded from `http://www.iis.fraunhofer.de/EN/bf/bv/kognitiv/biom/dd.jsp`.

1 Introduction

Technology that enables the detection of faces is developed and improved especially with respect to their performance, error rates and computational effort. The same is valid for systems that analyse faces in order to extract information like time of focus, gender, age or type of reaction. We think that there are other important features that such technology should focus on:

- possibility to efficiently use the software in dedicated applications, so that developers using the software can use the technology with little integration effort,
- configuration capabilities that on the one hand make the software easy to use and on the other hand offer extensive and manifold configuration possibilities.

We present a method for fast and robust face (and object) detection and analysis. The algorithms are embedded in a library called SHORE, an acronym for Sophisticated High-

speed Object Recognition Engine. Using different setups the engine can be used either for simple face detection or for more complex tasks like gender classification, tracking of faces, the analysis of facial features, classification whether the eyes are opened and closed and so on.

In this paper we at first describe the functionality and the capabilities of the system. Then we explain the design of our framework in order to fullfill the two features mentioned above. Afterwards we describe the algorithms that we are using and finally we show the results of benchmark tests that we have carried out on standard datasets.

2 Functionality

Our software framework enables the detection, analysis and identification of objects. Up to now we have focused on faces as special objects, but it is also possible to handle other objects that consist of a typical structure. Thus we have already made some promising experiments with the detection of hands or cars.

In the training phase a certain amount of annotated sample images is needed for building a model. *Annotated* means that in advance a certain set of specific points is defined and that these points are marked in each image.

We have created a system that is able to rapidly detect faces in arbitrary images and to further analyze them. So we can also detect the positions of the eyes, the nose and the mouth. In addition we can analyze whether the eyes and the mouth are open or closed. Furthermore we have implemented the classification of gender and the estimation of certain facial expressions (so far happy, angry, sad, surprised).

3 The Framework

Software frameworks are reusable design solutions for a domain of applications or problems [GHJV94]. The presented framework addresses the problems of object detection, fine analysis and recognition (identification). We provide a so called black-box framework [Rie00] to the client, who only needs to know how to configure and use it. The internal details are hidden by the framework interfaces.

Figure 1 shows an overview with the fundamental parts of our framework including three applications (on the top), which use the framework in different use cases. The components used at runtime are located on the left and the middle of figure 1. The right side contains the parts employed for the training of models and annotation of images. Figure 1 gives an outline of our design efforts based on the following objectives:

- coverage of the functionality described in section 2,
- easy and flexible configuration and simple usage,

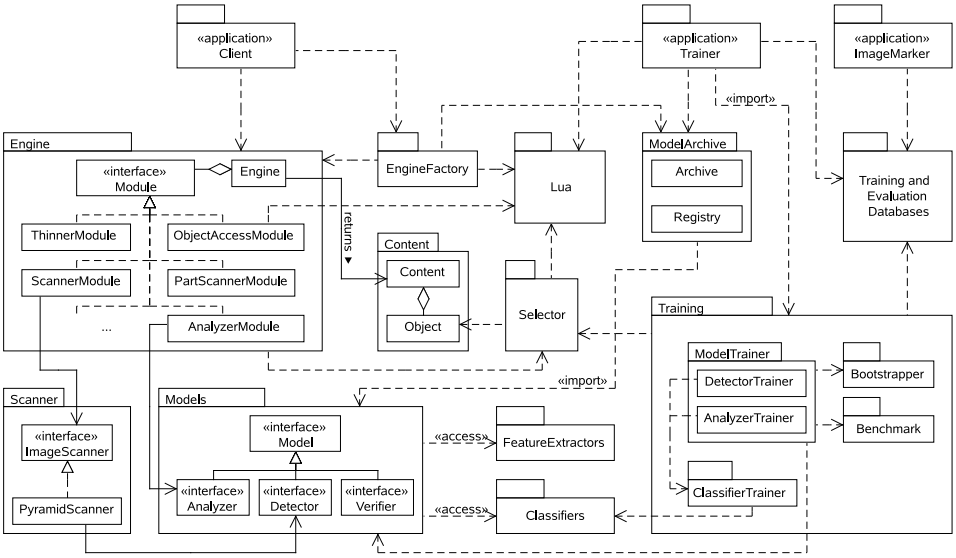

Figure 1: Overview with the fundamental components of our framework as an UML package diagram.

- generic object representation,
- reusability and extensibility,
- loosely coupled training,
- generic annotation of image databases.

To cover the functionality of object detection, fine analysis and recognition we introduced the concept of *Models* (left/middle bottom in figure 1). The three interfaces `Detector`, `Analyzer` and `Verifier` are specialized models that provide these functionalities. Models in general can be seen as an abstract representation of object characteristics, that are used for classification.

Easy configuration capability is achieved by the packages `Engine` and `EngineFactory`. A client can easily set up an `Engine` with the `EngineFactory` and the help of the Lua scripting language, see section 3.4.

Object detection requires an internal representation of the found objects. With a generic representation we can handle different objects, for example faces and cars, with one construct. An `Object`, middle of figure 1, holds its type, a region, name value pairs for ratings, attributes, marker points and other objects as parts. Different objects can be representated by these class attributes. The `Engine` returns a `Content`, that contains all detected and analyzed objects in an image.

Reusability and extensibility are common design issues. Important parts within our framework concering these issues are the `Module` and `Model` interfaces, `Classifiers` and `FeatureExtractors`. Functionality can be easy extended with new modules or models. Further feature extractors and classifiers can help to improve the overall performance.

The following three sections describe the three major use cases for the framework and show the basic design ideas. Thereafter section 3.4 gives some insights how we use the Lua scripting language within the framework.

3.1 Annotating Images

Figure 1 shows the `ImageMarker` application in the top right corner as a part of the framework. The ImageMarker was developed as a tool for flexible manual annotation of objects in images. It is possible to annotate the type and the region of objects in the image. In addition predefined and named marker points and key value pairs, so called attributes, can be annotated for each object. Which kinds of objects and how the objects are annotated is configured in the application settings. New object types can be added any time simply by defining them in a preference file.

A frontal face in the image for example is annotated by marking both eyes, the tip of the nose and both mouth corners. Each marker has a key that describes the marker, e.g. `"LeftEye"`. In addition different attributes can be assigned to the face. In the frontal face example the gender or facial expression can be annotated using the tool, e.g. `Gender:="Male"` or `Expression:="Happy"`. A screenshot of this sample annotation using the ImageMarker is given in figure 2.

The output of the annotation process is a file containing the information about all the objects in the image in xml. The annotated image databases are used for the training and benchmark of models within the framework as shown in figure 1. During the training process the objects that are used as training data can be exactly selected and cut out by using the image annotation. This is described in more detail in the following section.

3.2 Training Models

The training concept in our framework has two important properties. First the training parts are decoupled from the remaining framework, that allows us to provide only the components needed at runtime to the user of the library. Secondly the whole training is scriptable. For the training we use Lua-based training scripts, which define the training and related tasks. The `Trainer` application in figure 1 can parse these scripts with the help of Lua and uses the `Training` package to carry out the training. Details on the use of the Lua scripting language can be found in section 3.4.

A typical training script comprises the following workflow:

- select training images from annotated databases,

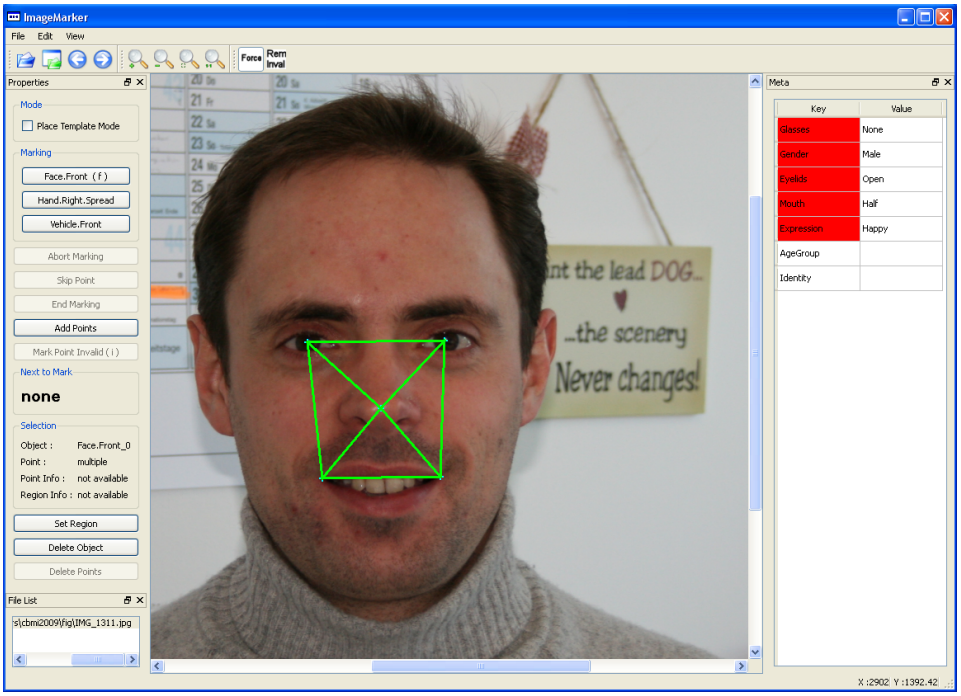

Figure 2: Screenshot of the ImageMarker with a sample annotation.

- choose feature extractors (package `FeatureExtractors`),
- choose trainers for the classifiers (package `ClassifierTrainers`),
- start training with a concrete model trainer (package `ModelTrainer`),
- optimize the model, e.g. with the `Bootstrapper` package,
- benchmark the trained model (package `Benchmark`),
- save the model with the `ModelArchive`.

The selection of the training images is done by the `Selector` package. Using selectors we can exactly define the subsets of images, that match the designated classes for the classifier training, see section 3.4.1. Depending on a concrete model we need a suitable model trainer, which knows how to train its model and handle training related tasks.

3.3 Using the Engine

The left side of figure 1 shows the usage of the framework from the viewpoint of the client application. Essentially the client only uses the `EngineFactory` and the `Engine` itself. The engine factory is able to create a custom tailored engine by parsing a Lua-based setup script that defines the configuration of the engine. An example for an engine setup script is depicted in section 3.4.3.

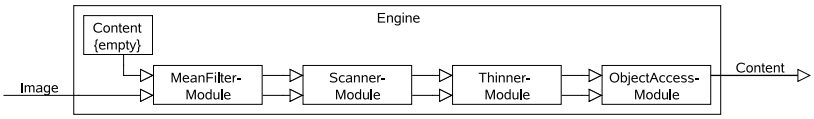

Figure 3: An engine setup configured for a simple face detection task.

Depending on the setup, a concrete engine can do just primitive image processing jobs or even quite complex object detection and analyzing tasks. A simple engine for example could just detect faces in images, whereas a more complex one makes an additional facial feature fine search and a facial expression or gender analysis.

The engine itself encapsulates a concatenation of different *Modules* as shown in figure 3 for a simple configuration example. When using the engine the image and a `Content` instance is provided to the modules one after each other. Each module is allowed to use or modify the image data and/or access the objects in the content.

A client may use the framework in the following way: First it provides the engine setup script to the engine factory. The factory parses the script to create the engine and returns the engine back to the client application. Thereafter the engine is ready to process images provided by the application. For each image the engine returns the content that contains all the objects and the information gathered by the engine. The application can read out the content for each image and may do further processing depending on the information in the content. When the engine is not needed any more it can be destroyed by the application again.

3.4 The Lua Scripting Language as an Instrument for Flexible Configuration

Lua is a very powerful, light-weight and fast scripting language. It can provide scripting capabilities to all kinds of applications. Looking at `http://www.lua.org` gives a good impression of its properties and advantages compared to other scripting languages.

In our framework we use it as an *extension* and *extensible* language [lua03]. These views summarize how Lua and C interact, in which Lua is implemented. The use of Lua as an extension language means that C code can be extended with the scripting facilities of Lua. As an extensible language Lua offers the facility to extend it with new functionality written in C or another language [lua03]. Within our framework we use both interaction modes

for the Selector, ObjectAccessModule (extension), the EngineFactory and the Trainer application (extensible).

3.4.1 The Use of Lua as an Extension Language

A Selector exactly defines or selects a set of objects that fulfill the selector expression. Lets take a simple example. A Lua-based selector that selects all objects of type "Face" and where the attribute called "Gender" equals "Male" looks like this:

```
Object.Type                 == "Face" and
Object.Attribute["Gender"] == "Male"
```

The selector is realized by pushing all the object properties first in the so called Lua state. This makes them available as variables within Lua. After that the selector expression is parsed by Lua. The result of parsing the selector is a boolean that tells us whether the object belongs to the selected subset or not.

Selectors are widely used during training and runtime. Within the model training selectors exactly define which objects in a training image database belong to the predefined classes. This way it is for example very easy to define the positive and negative training datasets for a classifier that distinguishes between male and female faces. During runtime the selectors are used in modules of the engine that access or manipulate objects. Let's take for example an engine that detects faces as well as hands. In this case it can easily be defined by selectors which modules in the engine take care of or manipulate the hand objects and which ones take care of the face objects.

The ObjectAccessModule uses Lua to provide a very flexible way to access and manipulate objects within the engine. This module is realized in a way similiar to the Selector. All the object properties are pushed in the Lua state first. After that a given Lua expression is executed that grants access to all the object properties. Thereby the object can be manipulated within the Lua state. At the end the object is read out again from the Lua state and replaces the original one.

A simple example how to use this module is shown at the end of the engine setup in listing 1. It is described in more detail in section 3.4.3.

3.4.2 The Use of Lua as an Extensible Language

The EngineFactory and the Trainer application extend Lua to use the framework in a flexible and comfortable way. Both register functions and types (classes) in the Lua environment [lua03] to make them available for scripting. For the registration of external parts their identifiers and the name for the Lua access must be pushed to the Lua state. For technical details see [lua03, Part IV] or [IdFC06].

Since the Trainer is a stand-alone application, we extended the default Lua interpreter with our requirements. All components accessible from a training script are registered in the Lua state of the interpreter. The registration by hand would be impractical and inefficient, because we have lots of training related components. One easy and flexible

way to make C and C++ accessible in Lua is to employ SWIG (Simplified Wrapper and Interface Generator), see `http://www.swig.org`.

Unlike the `Trainer` the `EngineFactory` is not an application. But it is also able to parse scripts with configuration information to setup an engine. The factory holds its own Lua state, which is extended with the `Engine` class and setup functions. An extract of our registered functions for the setup can be found in listing 1.

3.4.3 Configuration Example

A sample engine configured for a simple face detection task is shown in figure 3. It starts with a `MeanFilterModule` that just applies a 3×3 mean filter on the input image to reduce noise. The second module scans the image for faces using a face model. The `ThinnerModule` removes multiple detections by applying geometric constraints. And finally the `ObjectAccessModule` is able to modify the found objects in a very flexible way by using Lua.

The corresponding setup script for this configuration is given in listing 1. A primitive example for the flexibility of manipulating objects with the `ObjectAccessModule` is shown at the end of the script. In this case it calculates the eye distance and adds it to the object as an attribute with key `'EyeDistance'`. The provided selector makes sure that only faces are accessed by the module. The `EngineFactory` is able to parse this script to create the `Engine`.

The `Engine` package in figure 1 shows five different sample modules. In our current framework we implemented over 25 different modules for various kinds of image analysis tasks but also for object and content manipulation that can be used within the engine.

Further more complex engine sample configurations are realized and shown in our demo application (for a screenshot see figure 4).

4 The Algorithms

In this section we explain in short all four parts that make up our (and normally every) pattern classification system: preprocessing, feature extraction, classification and training. A more detailed description of the workflow with respect to certain speedups of the algorithms is presented in [KE06].

As a preprocessing step we simply use a mean filter which has turned out to be an easy and fast way to smooth noise artifacts. Further preprocessing is not carried out.

For feature classification we have decided to use features that are at least somewhat independent with respect to illumination conditions. From these features we are creating a *feature pool*. In the training phase it is selected which features are actually going to be used.

In our feature pool we have implemented census features (also known as local binary

```
function CreateFaceEngine()
   -- Create an empty engine instance
   engine = NewEngine()

   -- Add a 3x3 mean filter to the engine
   engine:AddMeanFilter()

   -- Define the model used for detection
   model = "FaceFront_24x24_2008_08_29_161712_7"

   -- Add the face scanner
   engine:AddImageScanner(
      0, 0, 1, 1,  -- Search region in image
      0, 1,        -- Min/max face size
      engine:GetPyramidScanner(
         engine:GetGridScanner(
            model,         -- Defined above
            2,             -- Search step x
            2              -- Search step y
         ),
         1.24,    -- Scale step
         2        -- Thread count
      )
   )

   -- Define a selector that selects
   -- only objects of type "Face"
   selector = 'Object.Type == "Face.Front"'

   -- Remove multiple detections
   engine:AddObjectThinner(
      selector,  -- Only frontal faces
      0.2,       -- Min overlap
      "Score"    -- Rating key
   )

   -- Calculate the eye distance and add it as
   -- an attribute with key 'EyeDistance'
   engine:AddObjectAccess(
      selector,  -- Access only faces
      [[
         le = Object.Marker.LeftEye
         re = Object.Marker.RightEye

         dx = le.X - re.X
         dy = le.Y - re.Y

         ed = math.sqrt( dx*dx + dy*dy )

         Object.Attribute["EyeDistance"] = ed
      ]]
   )

   return engine
end
```

Listing 1: The engine setup script for the sample configuration of figure 3.

patterns) [ZW96]. These features are defined as structure kernels of size 3×3 which summarize the local spatial image structure. Within the kernel structure information is coded as binary information $\{0, 1\}$ and the resulting binary patterns can represent oriented edges, line segments, junctions, ridges, saddle points, etc. In addition we are using resized

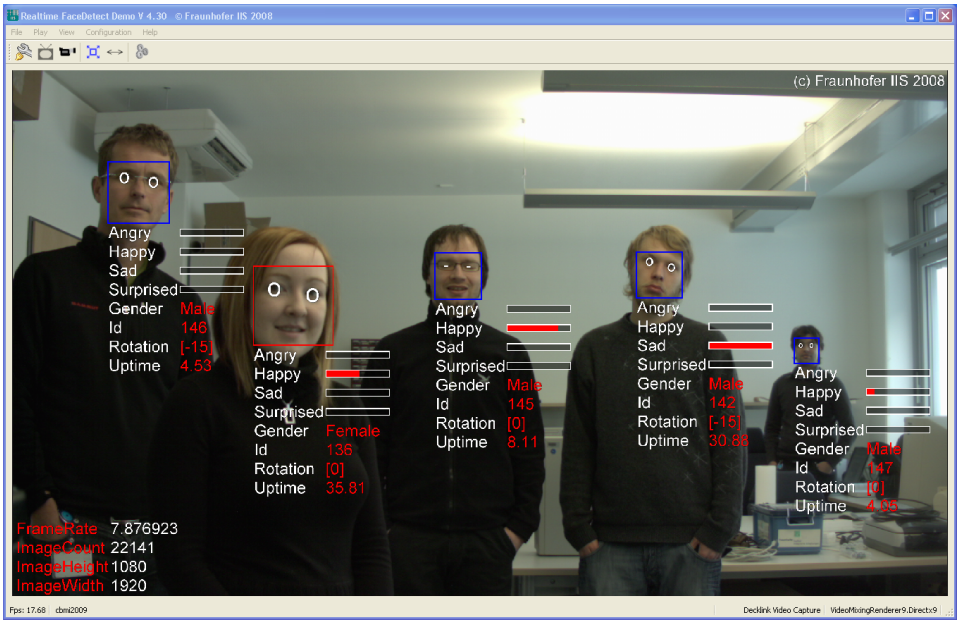

Figure 4: Screenshot of our demo application.

versions of these features which means that the features are not only calculated on simple 3×3 patterns but also on $3n \times 3m$ multiples of these windows.

Other features in the pool are edge orientation features. They can be calculated for each feature as $atan2(sx, sy)$ whereas sx and sy mean the result of the sobel filter applied to the respective pixel in x- and y- direction similar to the description in [FK02].

Doing so, starting from a 24×24 window we yield a total of approximately 30000 features, building the set $\mathcal{F}$ of available features.

During the training phase these features are selected and weighted using the AdaBoost method. By using boosting a number of weak classifiers are combined to form a final strong classifier. The goodness of a weak classifier is measured by its error ϵ on the training set. Our classifier then consists of a set of look-up tables $\{h_{\mathbf{x}}; \mathbf{x} \in \mathcal{F}\}$ for the features $\mathbf{x}$ chosen by the algorithm. Each look-up table holds a weight for each feature value. This method is described in more detail in [KE06, FS99].

One important advantage of using the AdaBoost method is that we gain a very fast classifier that uses look-up tables for the classification process. The detection can be further improved by using multiple stages as described in [VJ01] and [KE06]. Further speed improvements are made by carrying out a coarse-to-fine grid search as explained in [FK02].

5 Benchmark Results

The performance of our system was tested on several image data sets. None of the datasets that were used for benchmarking are part of the training sets, all sets of data are disjoint. We show rates for face detection, gender classification and the classification of happy faces. The face detection rates were calculated on the CMU+MIT (consisting of 130 images with 507 faces) and the BioID data base (consisting of 1521 images with 1522 faces[1]). The outcomes are shown in figure 5. When examining the detection rate for certain false positives we compare to competing technologies as shown in table 1. The numbers from Viola and Delakis are picked from the paper [RV08], where also other methods are compared.

false positives	10	31	65
This work	91.5	93.3	93.9
Viola and Jones[VJ01]	83.2	88.4	92.0
Garcia and Delakis[GD04]	90.5	91.5	92.3

Table 1: Comparison of selected algorithms with respect to detection rate on the CMU+MIT dataset. All numbers are given in percent.

The following measurements were made on images where we used pre-annotated eye-positions. The given recognition rates are based on the optimal classification thresholds choosen separately for each test data set.

The performance of our gender classification module was tested on the BioID data set and on the Feret fafb data set [PWHR98]. We receive a recognition rate of 94.3% on the BioID data base and 92.4% on the FERET fafb data base.

Our happiness analyzer was evaluated on the JAFFE data base [LAKG98]. It consists of 213 images of Japanese women with 31 annotated as happy. Here the recognition rate is 95.3%.

In table 2 we show the results of speed measurements on our system. The measurements were carried out on an Intel Core 2 Duo 6420 CPU. Although the library supports multi-threading we only used one core in this test. For evaluation we used the full BioID data set containing images of size 384×286 and we determined the average calculation time per image.

face detection	×	×	×	×
eye fine search		×	×	×
gender classification			×	×
analysis of 4 expressions				×
time [ms]	9.4	19.3	19.9	22.0

Table 2: Assessment of computation time for different engine setups on a single core of a Intel Core 2 Duo 6420 CPU for an image of size $\mathbf{384 \times 286}$.

[1] In the BioID data set there is one image (number 1140) showing two faces.

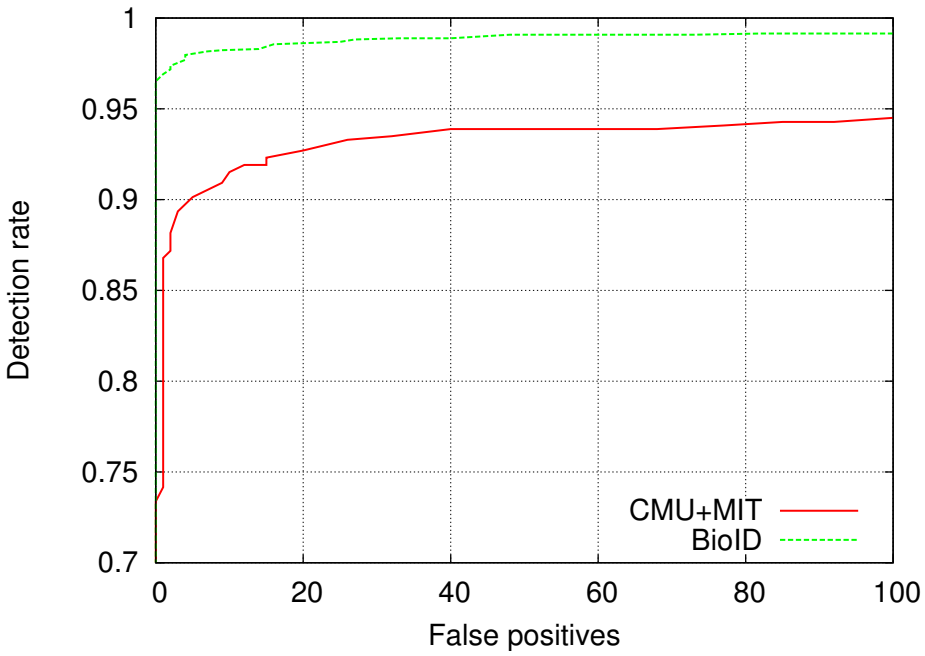

Figure 5: ROC-curve for the two data bases BioID [Fri] and CMU+MIT [Wan].

6 Conclusions

In this paper we have shown an approach to a modular framework to detect and analyze faces. The system features flexibility and modularity so that the integration to specific tasks can be carried out efficiently. We have shown a variety of different recognition tasks that can be combined by setting up the recognition engine in different ways. In addition it is easy to use the script Language Lua to add additional functionality like script logging or a XML-based TCP/IP-Server. We think that due to these properties the system is well suited for running in a real-time environment for audience measurement.

Everyone who is interested trying out the system can download and evaluate our demo software from `http://www.iis.fraunhofer.de/EN/bf/bv/kognitiv/biom/dd.jsp`. At the moment we are involved in adding functionality like age estimation and pose estimation. In addition we intend to train further classifiers for example for robust hand detection. Furthermore the different components like feature extractors and classifiers will probably be extended in the future.

This research work was partly supported by the European Commission under contract: FP6-2005-IST-5 (IMAGINATION), `http://www.imagination-project.org`.

References

[FK02] Bernhard Fröba und Christian Küblbeck. Robust Face Detection at Video Frame Rate based on Edge Orientation Features. In *International Conference on Automatic Face and Gesture Recognition (FG '02)*, Seiten 342–347, Washington D.C., May 2002.

[Fri] Robert Frischholz. The BioID Face Database. Website. Available online at `http://www.bioid.com/downloads/facedb/index.php`; visited on January 7th 2009.

[FS99] Yoav Freund und Robert E. Shapire. A Short Introduction to Boosting. In *Journal of Japanese Society for Artificial Intelligence*, number 14, Seiten 771–780, September 1999.

[GD04] Christophe Garcia und Manolis Delakis. Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(11):1408–1423, 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[IdFC06] Roberto Ierusalimschy, Luiz Henrique de Figueiredo und Waldemar Celes. *Lua 5.1 Reference Manual*, August 2006. Available online at `http://www.lua.org/manual/5.1`; visited on January 7th 2009.

[KE06] Christian Kueblbeck und Andreas Ernst. Face Detection and tracking in video sequences using the modified census transformation. *Image Vision Computing*, 24(6):564–572, June 2006.

[LAKG98] M. Lyons, S. Akamatsu, M. Kamachi und J. Gyoba. Coding facial expressions with Gabor wavelets. *Automatic Face and Gesture Recognition, 1998. Proceedings. Third IEEE International Conference on*, Seiten 200–205, Apr 1998.

[lua03] *Programming in Lua*. Roberto Ierusalimschy, first. Auflage, 2003. Available online at `http://www.lua.org/pil`; visited on January 7th 2009.

[PWHR98] P. Jonathon Phillips, Harry Wechsler, Jeffery Huang und Patrick J. Rauss. The FERET database and evaluation procedure for face-recognition algorithms. *Image and Vision Computing*, 16(5):295 – 306, 1998.

[Rie00] Dirk Riehle. *Framework Design - A Role Modeling Approach*. Dissertation, Swiss Federal Institute of Technology Zurich, May 2000. Available online at `http://www.riehle.org/computer-science/research/dissertation/diss-a4.pdf`, visited on January 7th 2009.

[RV08] Mauricio Correa Rodrigo Verschae, Javier Ruiz-del-Solar. A unified learning framework for object detection and classification using nested cascades of boosted classifiers. *Machine Vision and Applications (2008) 19:85103*, 2008.

[VJ01] Paul Viola und Michael Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition*, 2001.

[Wan] Chieh Chih Wang. CMU Image Data Base: face. Website. Available online at `http://vasc.ri.cmu.edu/idb/html/face/frontal_images/`; visited on January 7th 2009.

[ZW96] Ramin Zabih und John Woodfill. A Non-Parametric Approach to Visual Correspondence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1996.