Face Recognition Vendor Test (FRVT) (Ongoing)

Still Face 1:1 Verification Concept, Evaluation Plan and API

Version 2.0

Updates since version 1.0 of this document are highlighted in green.

Patrick Grother and Mei Ngan

Contact via frvt@nist.gov

Image Group
Information Access Division
Information Technology Laboratory

National Institute of Standards and Technology
U.S. Department of Commerce

May 22, 2017

Table of Contents

List of Tables

List of Figures

1. FRVT

1.1. Scope

This document establishes a concept of operations and an application programming interface (API) for evaluation of face recognition (FR) implementations submitted to NIST's ongoing Face Recognition Vendor Test. This API is for the 1:1 identity verification track. Separate API documents will be published for future additional tracks to FRVT. All images include exactly one face.

1.2. Audience

Participation in FRVT is open to any organization worldwide. There is no charge for participation. The target audience is researchers and developers of FR algorithms. While NIST intends to evaluate stable technologies that could be readily made operational, the test is also open to experimental, prototype and other technologies. All algorithms **must** be submitted as implementations of the API defined in this document.

1.3. Rules for Participation

1.3.1. Participation Agreement

A participant must properly follow, complete, and submit the FRVT Participation Agreement. This must be done once, either prior or in conjunction with the very first algorithm submission. It is not necessary to do this for each submitted implementation thereafter UNLESS there are major organizational changes to the submitting entity.

1.3.2. Number and Schedule of Submissions

Participants may send up to two initial submissions that run to completion. After that, participations may send one submission as often as every ~~120 days~~ three calendar months from the last submission for evaluation. NIST will evaluate implementations on a first-come-first-served basis, and quickly publish results.

1.4. Reporting

For all algorithms that complete the evaluations, NIST will post performance results on the NIST FRVT website. NIST will maintain an email list to inform interested parties of updates to the website. Artifacts will include a leaderboard highlighting the top performing submissions in various areas (e.g., accuracy, speed etc.) and individual implementation-specific report cards. NIST will maintain reporting on the two most recent algorithm submissions from any organization. Prior submission results will be archived but remain accessible via a public link.

Important: This is an open test in which NIST will identify the algorithm and the developing organization. Algorithm results will be attributed to the developer. Results will be machine generated (i.e. scripted) and will include timing, accuracy and other performance results. These will be posted alongside results from other implementations. Results will be expanded and modified as additional implementations are tested, and as analyses are implemented. Results may be regenerated on-the-fly, usually whenever additional implementations complete testing, or when new analysis is added.

NIST may additionally report results in workshops, conferences, conference papers and presentations, journal articles and technical reports.

1.5. Hardware specification

NIST intends to support high performance by specifying the runtime hardware beforehand. There are several types of computer blades that may be used in the testing. Each CPU has 512K cache. The bus runs at 667 Mhz. The main memory is 192 GB Memory as 24 8GB modules. We anticipate that 16 processes can be run without time slicing, though NIST will handle all multiprocessing work via `fork()`[1]. Participant-initiated multiprocessing is not permitted.

[1] http://man7.org/linux/man-pages/man2/fork.2.html

NIST is requiring use of 64 bit implementations throughout.

1.5.1. Central Processing Unit (CPU)-only platforms

The following list gives some details about the hardware of each CPU-only blade type:

- Dual Intel Xeon X5680 3.3 GHz CPUs (6 cores each)
- Dual Intel Xeon X7560 2.3 GHz CPUs (8 cores each)
- Dual Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz (10 cores each)

1.5.2. Graphics Processing Units (GPU)-enabled platforms

The following provides some details about the hardware of GPU-enabled machines:

- Dual Intel Xeon E5-2695 3.3 GHz CPUs (14 cores each; 56 logical CPUs total) with Dual NVIDIA Tesla K40 GPUs, with 12GB of memory per GPU

All GPU-enabled machines will be running CUDA version 7.5. cuDNN v5 for CUDA 7.5 will also be installed on these machines. Implementations that use GPUs will only be run on GPU-enabled machines. Please note that GPU-dependent implementations submitted to FRVT will have longer test turnaround times than CPU-only implementations due to resource constraints. Developers submitting GPU implementations are encouraged to submit "CPU-equivalent" implementations of their algorithms for timing comparisons. Algorithms using GPUs will be identified as such in public reports.

1.6. Operating system, compilation, and linking environment

The operating system that the submitted implementations shall run on will be released as a downloadable file accessible from http://nigos.nist.gov:8080/evaluations/CentOS-7-x86_64-Everything-1511.iso, which is the 64-bit version of CentOS 7.2 running Linux kernel 3.10.0.

For this test, Windows machines will not be used. Windows-compiled libraries are not permitted. All software must run under CentOS 7.2.

NIST will link the provided library file(s) to our C++ language test drivers. Participants are required to provide their library in a format that is dynamically-linkable using the C++11 compiler, g++ version 4.8.5.

A typical link line might be

```
g++ -std=c++11 -I. -Wall -m64 -o frvt11 frvt11.cpp  -L.  –lfrvt11_acme_07_cpu
```

The Standard C++ library should be used for development. The prototypes from this document will be written to a file "frvt11.h" which will be included via

```
#include <frvt11.h>
```

The header files will be made available to implementers at https://github.com/usnistgov/frvt.

All compilation and testing will be performed on x86_64 platforms. Thus, participants are strongly advised to verify library-level compatibility with g++ (on an equivalent platform) prior to submitting their software to NIST to avoid linkage problems later on (e.g. symbol name and calling convention mismatches, incorrect binary file formats, etc.).

1.7. Software and Documentation

1.7.1. Library and Platform Requirements

Participants shall provide NIST with binary code only (i.e. no source code). The implementation should be submitted in the form of a dynamically-linked library file.

The core library shall be named according to Table 1. Additional supplemental libraries may be submitted that support this "core" library file (i.e. the "core" library file may have dependencies implemented in these other libraries). Supplemental libraries may have any name, but the "core" library must be dependent on supplemental libraries in order to be linked correctly. The **only** library that will be explicitly linked to the FRVT 1:1 test driver is the "core" library.

Intel Integrated Performance Primitives (IPP) ® libraries are permitted if they are delivered as a part of the developer-supplied library package. It is the provider's responsibility to establish proper licensing of all libraries. The use of IPP libraries shall not prevent running on CPUs that do not support IPP. Please take note that some IPP functions are multithreaded and threaded implementations are prohibited.

NIST will report the size of the supplied libraries.

Table 1 – Implementation library filename convention

Form	libfrvt11_provider_sequence_processor.ending				
Underscore delimited parts of the filename	libfrvt11	provider	sequence	processor	ending
Description	First part of the name, required to be this.	Single word, non-infringing name of the main provider EXAMPLE: Acme	A three digit decimal identifier to start at 000 and incremented by 1 every time a library is sent to NIST. EXAMPLE: 007	"gpu" if implementation uses GPUs; "cpu" otherwise	.so
Example	libfrvt11_acme_007_cpu.so				

Important: Public results will be attributed with the provider name and the 3-digit sequence number in the submitted library name.

1.7.2. Configuration and developer-defined data

The implementation under test may be supplied with configuration files and supporting data files. NIST will report the size of the supplied configuration files.

1.7.3. Submission folder hierarchy

Participant submissions shall contain the following folders at the top level

- lib/ - contains all participant-supplied software libraries
- config/ - contains all configuration and developer-defined data
- doc/ - contains any participant-provided documentation regarding the submission
- validation/ - contains validation output

1.7.4. Installation and Usage

The implementation shall be installable using simple file copy methods. It shall not require the use of a separate installation program and shall be executable on any number of machines without requiring additional machine-specific license control procedures or activation. The implementation shall not use nor enforce any usage controls or limits based on licenses, number of executions, presence of temporary files, etc. The implementation shall remain operable for at least six months from the submission date.

1.7.5. Documentation

Participants shall provide documentation of additional functionality or behavior beyond that specified here. The documentation must define all (non-zero) developer-defined error or warning return codes.

1.7.6. Modes of operation

Implementations shall not require NIST to switch "modes" of operation or algorithm parameters. For example, the use of two different feature extractors must either operate automatically or be split across two separate library submissions.

1.8. Runtime behavior

1.8.1. Interactive behavior, stdout, logging

The implementation will be tested in non-interactive "batch" mode (i.e. without terminal support). Thus, the submitted library shall:

- Not use any interactive functions such as graphical user interface (GUI) calls, or any other calls which require terminal interaction e.g. reads from "standard input".
- Run quietly, i.e. it should not write messages to "standard error" and shall not write to "standard output".
- Only if requested by NIST for debugging, include a logging facility in which debugging messages are written to a log file whose name includes the provider and library identifiers and the process PID.

1.8.2. Exception Handling

The application should include error/exception handling so that in the case of a fatal error, the return code is still provided to the calling application.

1.8.3. External communication

Processes running on NIST hosts shall not side-effect the runtime environment in any manner, except for memory allocation and release. Implementations shall not write any data to external resource (e.g. server, file, connection, or other process), nor read from such, nor otherwise manipulate it. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

1.8.4. Stateless behavior

All components in this test shall be stateless, except as noted. This applies to face detection, feature extraction and matching. Thus, all functions should give identical output, for a given input, independent of the runtime history. NIST will institute appropriate tests to detect stateful behavior. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

1.9. Single-thread Requirement/Parallelization

Implementations must run in single-threaded mode, because NIST will parallelize the test by dividing the workload across many cores and many machines. Implementations must ensure that there are no issues with their software being parallelized via the `fork()` function – this applies to both GPU and CPU implementations submitted to FRVT.

For implementations using the GPU: For any given GPU, NIST will run a single implementation process (i.e., fork() once per GPU), with 12GB of main memory available for use by the algorithm. NIST machines are equipped with dual GPUs, and the NIST test harness will load balance by telling the implementation which GPU to use via the section 3.3.2.1 setGPU() function call. All calls to setGPU() will be performed after a call to fork(). Implementations using the GPU are encouraged to perform initialization within the setGPU() function where 1. which GPU to use is provided to the implementation and 2. to support known limitations of commonly used deep learning frameworks such as Caffe, where initialization must take place in the worker process.

1.10. Time limits

The elemental functions of the implementations shall execute under the time constraints of Table 2. These time limits apply to the function call invocations defined in section 3. Assuming the times are random variables, NIST cannot regulate the maximum value, so the time limits are 90-th percentiles. This means that 90% of all operations should take less than the identified duration.

The time limits apply per image. When K images of a person are present, the time limits shall be increased by a factor K.

Table 2 – Processing time limits in milliseconds, per 640 x 480 image

Function	1:1 verification
Training	12 hours for an input set of 6000 images
Feature extraction enrollment	1000 (1 core) 640x480 pixels
Feature extraction for verification	1000 (1 core) 640x480 pixels
Matching	5 (1 core)

2. Data structures supporting the API

2.1. Requirement

FRVT 1:1 participants shall implement the relevant C++ prototyped interfaces of clause 3. C++ was chosen in order to make use of some object-oriented features.

2.2. File formats and data structures

2.2.1. Overview

In this face recognition test, an individual is represented by K ≥ 1 two-dimensional facial images. All facial images in the test will contain one and only one face per image.

Table 3 – Structure for a single image

C++ code fragment	Remarks
typedef struct Image	
{	
uint16_t image_width;	Number of pixels horizontally
uint16_t image_height;	Number of pixels vertically
uint16_t image_depth;	Number of bits per pixel. Legal values are 8 and 24.
std::shared_ptr<uint8_t> data;	Managed pointer to raster scanned data. Either RGB color or intensity. If image_depth == 24 this points to 3WH bytes RGBRGBRGB... If image_depth == 8 this points to WH bytes IIIIIII
Label description;	Single description of the image. The allowed values for this field are specified in the enumeration in Table 4.
} Image;	

An **Image** will be accompanied by one of the labels given below. Face recognition implementations should tolerate **Images** of any category.

Table 4 – Labels describing categories of Images

Label as C++ enumeration	Meaning
enum class Label {	
UNKNOWN=0,	Either the label is unknown or unassigned.
ISO=1,	Frontal, intended to be in conformity to ISO/IEC 19794-5:2005.
MUGSHOT=2,	From law enforcement booking processes. Nominally frontal.
PHOTOJOURNALISM=3,	The image might appear in a news source or magazine. The images are typically taken by professional photographer and are well exposed and focused but exhibit pose and illumination variations.
EXPLOITATION=4	The image is taken from a child exploitation database. This imagery has highly unconstrained pose and illumination, expression and resolution.
WILD=5	Unconstrained image, taken by an amateur photographer, exhibiting wide variations in pose, illumination, and resolution.
};	

Table 5 – Structure for a set of images from a single person

C++ code fragment	Remarks
`using Multiface = std::vector<Image>;`	Vector of Image objects

2.2.2. Data structure for eye coordinates

Implementations should return eye coordinates of each facial image. This function, while not necessary for a recognition test, will assist NIST in assuring the correctness of the test database. The primary mode of use will be for NIST to inspect images for which eye coordinates are not returned, or differ between implementations.

The eye coordinates shall follow the placement semantics of the ISO/IEC 19794-5:2005 standard - the geometric midpoints of the endocanthion and exocanthion (see clause 5.6.4 of the ISO standard).

Sense: The label "left" refers to subject's left eye (and similarly for the right eye), such that xright < xleft.

Table 6 – Structure for a pair of eye coordinates

C++ code fragment	Remarks
`typedef struct EyePair`	
`{`	
`bool isLeftAssigned;`	If the subject's left eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
`bool isRightAssigned;`	If the subject's right eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
`uint16_t xleft;` `uint16_t yleft;`	X and Y coordinate of the center of the subject's left eye. If the eye coordinate is out of range (e.g. x < 0 or x >= width), `isLeftAssigned` should be set to false.
`uint16_t xright;` `uint16_t yright;`	X and Y coordinate of the center of the subject's right eye. If the eye coordinate is out of range (e.g. x < 0 or x >= width), `isRightAssigned` should be set to false.
`} EyePair;`	

2.2.3. Template Role

Labels describing the type/role of the template to be generated will be provided as input to template generation.

Table 7 – Labels describing template role

Label as C++ enumeration	Meaning
`enum class TemplateRole {`	
`Enrollment_11,`	Enrollment template for 1:1 matching
`Verification_11`	Verification template for 1:1 matching
`};`	

2.2.4. Data type for similarity scores

Identification and verification functions shall return a measure of the similarity between the face data contained in the two templates. The datatype shall be an eight byte double precision real. The legal range is [0, DBL_MAX], where the DBL_MAX constant is larger than practically needed and defined in the <climits> include file. Larger values indicate more likelihood that the two samples are from the same person.

Providers are cautioned that algorithms that natively produce few unique values (e.g. integers on [0,127]) will be disadvantaged by the inability to set a threshold precisely, as might be required to attain a false match rate of exactly 0.0001, for example.

2.2.5. Data structure for return value of API function calls

Table 8 – Enumeration of return codes

Return code as C++ enumeration	Meaning
`enum class ReturnCode {`	
`Success=0,`	Success
`ConfigError=1,`	Error reading configuration files
`RefuseInput=2,`	Elective refusal to process the input, e.g. because cannot handle greyscale
`ExtractError=3,`	Involuntary failure to process the image, e.g. after catching exception
`ParseError=4,`	Cannot parse the input data
`TemplateCreationError=5,`	Elective refusal to produce a template (e.g. insufficient pixels between the eyes)
`VerifTemplateError=6,`	For matching, either or both of the input templates were result of failed feature extraction
`NumDataError=7,`	The implementation cannot support the number of images
`TemplateFormatError=8,`	Template file is in an incorrect format or defective
`GPUError=9,`	There was a problem setting or accessing the GPU
`VendorError=10`	Vendor-defined failure. Failure codes must be documented and communicated to NIST with the submission of the implementation under test.
`};`	

Table 9 – ReturnStatus structure

C++ code fragment	Meaning
`struct ReturnStatus {`	
`ReturnCode code;`	Return Code
`std::string info;`	Optional information string
`// constructors`	
`};`	

2.2.6. Data structure for encapsulating training data

The following structure represents subject attributes that may be available to the implementation during training.

Table 10 – Structure containing subject metadata information

	Meaning
`typedef struct Attributes {`	
`enum class Gender {Unknown, Male, Female};`	
`enum class Race {Unknown, White, Black, EastAsian, SouthAsian, Hispanic};`	
`enum class EyeGlasses {Unknown, NotWearing, Wearing};`	
`enum class FacialHair {Unknown, Moustache, Goatee, Beard};`	
`enum class SkinTone {Unknown, LightPink, LightYellow, MediumPinkBrown, MediumYellowBrown, MediumDarkBrown, DarkBrown};`	
`std::string id;`	A subject ID that identifies a person. Images of the same person will have the same subject ID.
`double age;`	Subject age (in years). A negative value indicates age is unknown.
`Gender gender;`	Subject gender
`Race race;`	Subject race. This value may be a proxy.
`EyeGlasses eyeglasses;`	Whether the subject is wearing eyeglasses
`FacialHair facialhair;`	Facial hair type if applicable
`double height;`	Subject height (in meters). A negative value indicates height is unknown.
`double weight;`	Subject weight (in kilograms). A negative value indicates weight is unknown.
`SkinTone skintone;`	Subject skin tone

```
} Attributes;
```

Table 11 – Structure representing face image and associated attributes

C++ code fragment	Remarks
`using faceAttributePair = std::pair<Image, Attributes>;`	A pair of face image and associated attributes

3. API Specification

3.1. Namespace

All data structures and API interfaces/function calls will be declared in the `FRVT` namespace.

3.2. Overview

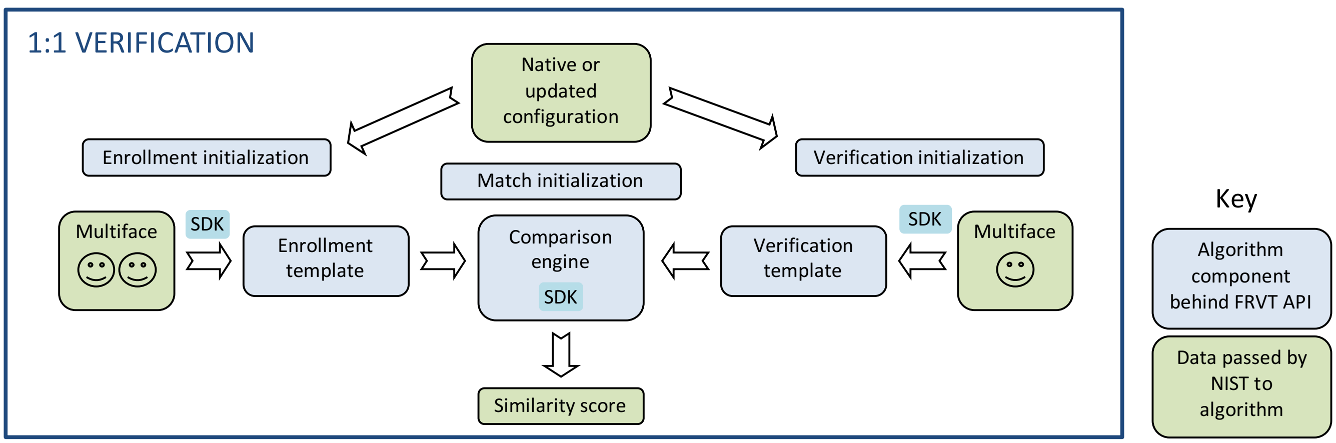

Figure 1 – Schematic of 1:1 verification

The 1:1 testing will proceed in the following phases: optional offline training; preparation of enrollment templates; preparation of verification templates; and matching. Note that training, template creation, and matching may all be performed as separate processes. These are detailed in Table 12.

Table 12 – Functional summary of the 1:1 application

Phase	Description	Performance Metrics to be reported by NIST
Training (Optional)	Given 1) K ≥ 1 images with associated subject ID and attribute data and 2) the implementation's configuration directory, the implementation may use the provided training data to populate a new "trained" configuration directory. This directory will be used to initialize the algorithm during subsequent template creation and matching processes. Images of the same person will have the same subject ID. Images with different subject IDs indicate they are different people. Attribute data may include the subject's age, gender, race, and other information. Please note that this function may or may not be called prior to creation of templates or matching. The implementation's ability to create or match templates should not be dependent on this function.	
Initialization	Function to read configuration data, if any.	None
Enrollment	Given K ≥ 1 input images of an individual, the implementation will create a proprietary enrollment template. NIST will	Statistics of the time needed to produce a template. Statistics of template size. Rate of failure to produce a

	manage storage of these templates.	template
Verification	Given K ≥ 1 input images of an individual, the implementation will create a proprietary verification template. NIST will manage storage of these templates.	Statistics of the time needed to produce a template. Statistics of template size. Rate of failure to produce a template.
Matching (i.e. comparison)	Given a proprietary enrollment and a proprietary verification template, compare them to produce a similarity score.	Statistics of the time taken to compare two templates. Accuracy measures, primarily reported as DETs, including for partitions of the input datasets.

NIST requires that these operations may be executed in a loop in a single process invocation, or as a sequence of independent process invocations, or a mixture of both.

3.3. API

3.3.1.1. Interface

The software under test must implement the interface `Interface` by subclassing this class and implementing each method specified therein.

	C++ code fragment	Remarks
1.	`class Interface`	
2.	`{` `public:`	
3.	`virtual ReturnStatus initialize(` `const std::string &configDir ) = 0;`	

4.	`virtual ReturnStatus createTemplate(`
	`const Multiface &faces,`
	`TemplateRole role,`
	`std::vector<uint8_t> &templ,`
	`std::vector<EyePair> &eyeCoordinates) = 0;`	

5.	`virtual ReturnStatus matchTemplates(`
	`const std::vector<uint8_t> &verifTemplate,`
	`const std::vector<uint8_t> &enrollTemplate,`
	`double &similarity) = 0;`	

6.	`virtual` ~~`void`~~ `ReturnStatus setGPU(uint8_t gpuNum) = 0;`	
7.	`static std::shared_ptr<Interface> getImplementation();`	Factory method to return a managed pointer to the `Interface` object. This function is implemented by the submitted library and must return a managed pointer to the `Interface` object.

8.	`virtual ReturnStatus train(`
	`const std::string &configDir,`
	`const std::string &trainedConfigDir,`
	`const std::vector<faceAttributePair> &faces) = 0;`	

9.	`};`	

There is one class (static) method declared in `Interface.getImplementation()` which must also be implemented by the implementation. This method returns a shared pointer to the object of the interface type, an instantiation of the implementation class. A typical implementation of this method is also shown below as an example.

	C++ code fragment	Remarks

```
#include "frvt11.h"

using namespace FRVT;

NullImpl:: NullImpl () { }

NullImpl::~ NullImpl () { }

std::shared_ptr<Interface>
Interface::getImplementation()
{
    return std::make_shared<NullImpl>();
}

// Other implemented functions
```

3.3.2. Initialization

The NIST test harness will call the initialization function in Table 13 before calling template generation or matching. This function will be called BEFORE any calls to fork() are made.

Table 13 – Initialization

Prototype	ReturnStatus initialize(const string &configDir);	 Input
Description	This function initializes the implementation under test. It will be called by the NIST application before any call to `createTemplate()` or `matchTemplates()`. The implementation under test should set all parameters. This function will be called N=1 times by the NIST application, prior to parallelizing M >= 1 calls to `createTemplate()` via `fork()`.	
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files. The name of this directory is assigned by NIST, not hardwired by the provider. The names of the files in this directory are hardwired in the implementation and are unrestricted.
Output Parameters	none	
Return Value	See Table 8 for all valid return code values.	

3.3.2.1. GPU Index Specification

For implementations using GPUs, the function of Table 14 specifies a sequential index for which GPU device to execute on. This enables the test software to orchestrate load balancing across multiple GPUs. This function will be called AFTER a call to fork() is made.

Table 14 – GPU index specification

Prototypes	~~void~~ ReturnStatus setGPU (uint8_t gpuNum);	 Input
Description	This function sets the GPU device number to be used by all subsequent implementation function calls. `gpuNum` is a zero-based sequence value of which GPU device to use. 0 would mean the first detected GPU, 1 would be the second GPU, etc. If the implementation does not use GPUs, then this function call should simply do nothing.	
Input Parameters	gpuNum	Index number representing which GPU to use.
Return Value	See Table 8 for all valid return code values.	

3.3.3. Template generation

The function of Table 15 supports role-specific generation of a template data. Template format is entirely proprietary.

Table 15 – Template generation

Prototypes	ReturnStatus createTemplate(const Multiface &faces,	 Input

	TemplateRole role,	Input
	std::vector<uint8_t> &templ,	Output
	std::vector<EyePair> &eyeCoordinates);	Output
Description	Takes a Multiface and outputs a proprietary template and associated eye coordinates. The vectors to store the template and eye coordinates will be initially empty, and it is up to the implementation to populate them with the appropriate data. In all cases, even when unable to extract features, the output shall be a template that may be passed to the matchTemplates() function without error. That is, this routine must internally encode "template creation failed" and the matcher must transparently handle this.	
Input Parameters	faces	Implementations must alter their behavior according to the number of images contained in the structure and the TemplateRole type.
	role	Label describing the type/role of the template to be generated
Output Parameters	templ	The output template. The format is entirely unregulated. This will be an empty vector when passed into the function, and the implementation can resize and populate it with the appropriate data.
	eyeCoordinates	For each input image in the Multiface, the function shall return the estimated eye centers. This will be an empty vector when passed into the function, and the implementation shall populate it with the appropriate number of entries. Values in eyeCoordinates[i] shall correspond to faces[i].
Return Value	See Table 8 for all valid return code values.	

3.3.4. Matching

Matching of one enrollment against one verification template shall be implemented by the function of Table 16.

Table 16 – Template matching

Prototype	ReturnStatus matchTemplates(	
	const std::vector<uint8_t> &verifTemplate,	Input
	const std::vector<uint8_t> &enrollTemplate,	Input
	double &similarity);	Output
Description	Compare two proprietary templates and output a similarity score, which need not satisfy the metric properties. When either or both of the input templates are the result of a failed template generation (see Table 15), the similarity score shall be -1 and the function return value shall be `VerifTemplateError`.	
Input Parameters	verifTemplate	A verification template from createTemplate(role=Verification_11). The underlying data can be accessed via verifTemplate.data(). The size, in bytes, of the template could be retrieved as verifTemplate.size().
	enrollTemplate	An enrollment template from createTemplate(role=Enrollment_11). The underlying data can be accessed via enrollTemplate.data(). The size, in bytes, of the template could be retrieved as enrollTemplate.size().
Output Parameters	similarity	A similarity score resulting from comparison of the templates, on the range [0,DBL_MAX]. See section 2.2.4.
Return Value	See Table 8 for all valid return code values.	

3.3.1. Training

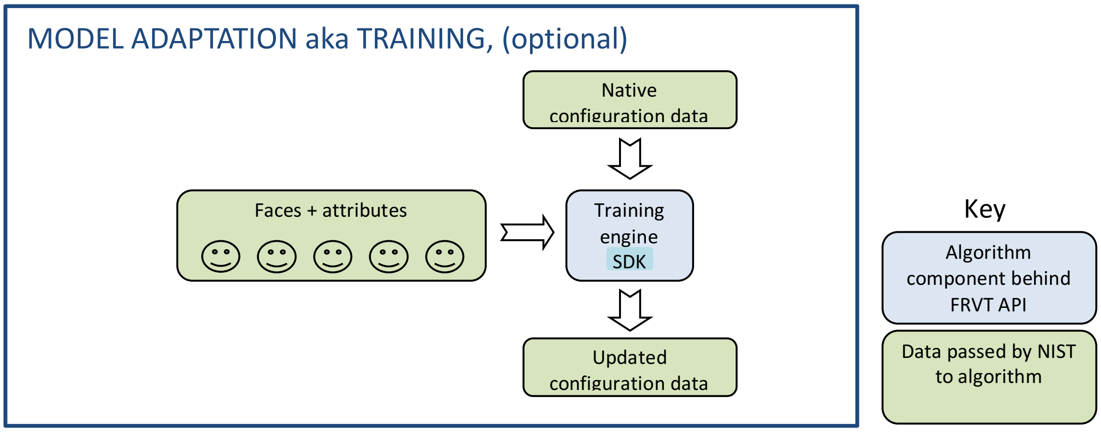

Figure 2 – Schematic of training

The NIST test harness may optionally call the training function in Table 17 as a separate process outside of the template generation and matching process. The implementation will be provided with the read-only configuration directory as supplied to NIST in the original submission, a read-write directory to store output(s) from training, and a set of face images and subject attributes where available.

Table 17 – Training

Prototype	ReturnStatus train(	
	const std::string &configDir,	Input
	const std::string &trainedConfigDir,	Input
	const std::vector<faceAttributePair> &faces);	Input
Description	This function provides the implementation with face images and associated attributes where available. Attributes include a subject ID (this value is always assigned), and where available, subject data such as age, gender, race, and other information. Images of the same person will have the same subject ID. Genuine associations can be created using images with the same subject ID, and imposter associations can be derived using images with different subject IDs. This function may or may not be called prior to creation of templates or matching. The implementation's ability to create or match templates should not be dependent on this function.	
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files. The name of this directory is assigned by NIST, not hardwired by the provider. The names of the files in this directory are hardwired in the implementation and are unrestricted.
	trainedConfigDir	A directory with read-write permissions where the implementation can store any training output. The name of this directory is assigned by NIST, not hardwired by the provider. The names of the files in this directory are hardwired in the implementation and are unrestricted. Important: This directory is what will subsequently be provided to the implementation's `initialize()` function as the input configuration directory if this training function is invoked. Therefore, at a minimum, even if you choose not to implement this function, the necessary data from the original configuration `configDir` must be copied over into this directory.
	faces	A vector of face image-attribute pairs provided to the implementation for training purposes
Output Parameters	none	
Return Value	See Table 8 for all valid return code values.	

Purpose of training: Broadly NIST is seeking a repeatable and robust mechanism to provide end users with an easy to use, automated, mechanism to get the benefits of training on their own data. The training function is intended to improve some aspect of recognition.

NIST's first attempt at exploiting the functionality of the train() API function call will be to address this problem: Some recognition algorithms give different impostor distributions for different age groups. So NIST will call train with thousands of images associated with an identity and age labels. An effective training mechanism would yield some configuration data that allowed the recognition components (createTemplate() and matchTemplates()) to improve stability of the impostor distribution across age groups. As a second test, we will then repeat this with race labels.

That said, developers can use this function for any purpose. You can assume that the tests that use the result of this step will be with images of the same type as that passed to the function. The training and test sets will have disjoint sets of people, reflecting the operational case where a training function should have utility over new users of a system.