Fusion of Face Recognition Algorithms (FOFRA) Prize Challenge 2018

Concept and API for Fusion of Face Recognition Scores and Templates

Patrick Grother
Mei Ngan
National Institute of Standards and Technology (NIST)

Christopher Boehnen
Intelligence Advanced Research Projects Activity (IARPA)

Lars Ericson
Science Applications International Corporation (SAIC)

Draft for review
May 22, 2018

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

Table of Contents

1. Overview

IARPA is sponsoring the Fusion of Face Recogition Algorithms (FOFRA) 2018 Prize Challenge to improve biometric fusion technology. The challenge addresses fusion of data from face recognition algorithms each applied to common input imagery. The goal is multi-algorithm fusion. Fusion is conducted either at the template level, or separately, at the score level. This document describes the conduct of the challenge and the application programming interface (API) behind which technology must operate.

Before reading this document, prospective participants should view the companion document, *Overview of FOFRA 2018 Fusion Prize Challenge*, that details what fusion tasks are in-scope. All supporting material is available at https://www.nist.gov/programs-projects/fusion-face-recognition-algorithms-2018.

1.1. Organization

IARPA is the sponsor the challenge. IARPA has delegated the actual evaluation to NIST who is responsible for the development and distribution of data, execution of fusion algorithms, performance analysis and reporting results to IARPA. IARPA, in consultation with a panel of US Government judges will determine who to award prizes to on the basis of the NIST results.

1.2. Motivation

Face recognition error rates, particularly on uncontrolled face imagery, are well above zero. While algorithm development has seen considerable investment, other mechanisms for improving accuracy are known. Among them, there is a large academic literature on biometric fusion, covering multimodal and multi-algorithmic fusion. It shows that substantial accuracy gains can be made over using a single mode, or a single algorithm alone, and this can be achieved, in large part, using quite simple methods. The gains reduce when the fused inputs are correlated. The vast majority of the literature addresses biometric verification, rather than identification. Moreover, the literature covers score-level fusion rather than feature (i.e. template) level fusion. The latter, on information theoretic grounds, offers greater accuracy gains at the expense of some complexity.

2. Overview

The FOFRA 2018 has two tracks, one for score-level fusion and the second for template-level fusion, as detailed below. A developer may enter either or both tracks. A developer may submit closed-source or open-source implementations, where "closed-source" means compiled libraries. NIST provides score and template training data to participants and asks them to develop fusion schemes and to provide those to NIST who will execute them on new, disjoint, test sets.

Score-level fusion: Given data from K >= 2 face recognition algorithms, a developer shall provide the following for each possible combination[1] of those algorithms:

- pre-trained "model" suitable for normalizing and fusing scores from the specified algorithms;
- a mechanism to fuse verification scores together to produce one output verification score;
- a mechanism to fuse identification candidate lists to produce one output candidate list.

Implementation of verification score fusion is not a requirement for candidate list fusion and vice versa, meaning participants are allowed to implement one and not the other.

For template-level fusion: Given data from K >= 2 face recognition algorithms, a developer shall provide the following for each possible combination of those algorithms:

[1] Given algorithms A, B, C the combinations are AB, AC, BC, ABC.

- pre-trained "models" suitable for fusing templates from the specified algorithms;
- a mechanism to fuse K templates together to produce one output template;
- a mechanism to compare two templates to produce a comparison score;
- a mechanism to enroll N templates into a gallery;
- a mechanism to search a new fused probe template against that gallery to produce a candidate list.

Implementation of template fusion and at least one of template verification or template search from a gallery is required to participate in this track. Implementation of template verification is not a requirement for template search from a gallery, and vice versa, meaning participants are allowed to implement one and not the other.

The following sections provide the APIs and supporting data structures for score and template-level fusion. The purpose of defining an API is to allow ease of automation at NIST, and to control what data the implementation has access to, so that it obeys operational constraints. Developers have the option of submitting either

1) an open or closed-source C++ implementation of their fusion method (see Section 4) or
2) an open-source R-language implementation (see Section 8).

3. Rules for Participation

3.1. Participation Agreement

NIST requires all FOFRA 2018 participants to send a completed participation agreement either prior to or in conjunction with their very first submission. The participation agreement posted on the FOFRA 2018 website.

3.2. Options for participation

All submissions shall implement exactly one of the functionalities defined in Table 1. A C++ library or R source code shall not implement the API of more than one task track. Therefore, participants shall send separate libraries or source files for each track individually.

Table 1 – FOFRA track participation requirements

Track	Score Fusion	Template Fusion
C++ API	Section 7.1	Section 7.2
R API	Section 8.1	Section 8.2

All submissions shall be named according to the naming conventions defined in the appropriate sections listed in Table 2.

Table 2 – FOFRA submission naming requirements

Track	Naming Requirements
C++ API	Section 7.5
R API	Section 8.3

3.3. Submissions for open-source implementations

3.3.1. Open-source implementation

For developers who choose to submit open-source code to NIST, there are two options.

- A full implementation – declarations and definitions - of the C++ interface given in section 4.

- R^2 code as defined in section 8.

3.3.2. Licensing

Important: In addition to completing a participation agreement, developers who submit an open-source implementation must

1. Make their software available for public download on a reputable website such as GitHub, university website, or corporate website.
2. Provide written documentation which explicitly acknowledges that their code may be released to the public and US government. This can potentially be achieved by releasing code under an Open Source Initiative (OSI)-approved license - https://opensource.org/licenses.

3.3.3. Participation Eligibility

To be eligible to win a prize under this competition, an individual or entity:

1. Must have visited the FOFRA website at Challenge.gov;
2. Must have read the FOFRA overview slides;
3. Must have completed and submitted a FOFRA participation agreement form;
4. Must have complied with all the requirements under these rules; and
5. Must be (1) an individual or team member each of whose members are 18 years of age and over, or (2) an entity incorporated.

The following are ineligible to participate

1. May not be a federal entity or federal employee acting within the scope of their employment. An individual or entity shall not be deemed ineligible because the individual or entity used federal facilities or consulted with federal employees during a competition if the facilities and employees are made available to all individuals and entities participating in the competition on an equitable basis.
2. Employees of IARPA, NIST, their affiliates, and/or any other individual or entity associated with the development, evaluation, or administration of the competition as well as members of such persons' immediate families (spouses, children, siblings, parents), and persons living in the same household as such persons, whether or not related, are not eligible to participate in the competition.
3. Foreign Nationals & International Developers: All Developers can participate with this exception: residents of, Iran, Cuba, North Korea, Crimea Region of Ukraine, Sudan or Syria or other countries prohibited on the U.S. State Department's State Sponsors of Terrorism list. In addition, Developers are not eligible to participate if they are on the Specially Designated National list promulgated and amended, from time to time, by the United States Department of the Treasury. It is the responsibility of the Developer to ensure that they are allowed to export their technology solution to the United States for the Live Test. Additionally, it is the responsibility of participants to ensure that no US law export control restrictions would prevent them from participating when foreign nationals are involved. If there are US export control concerns, please contact IARPA and we will attempt to make reasonable accommodations if possible. IARPA will not be held responsible for devices shipped or transported to the U.S. that are confiscated by local authorities or that violate local export laws.

In addition:

[2] https://www.r-project.org/

1. Federal grantees may not use federal funds to develop challenge applications unless consistent with the purpose of their grant award. Federal contractors may not use federal funds from a contract to develop challenge applications or to fund efforts in support of a challenge submission without written approval of both IARPA and their Federal government sponsor. Entities affiliated with the IARPA Janus program may participate, but are ineligible to win prize monies.
2. Entrants must agree to assume any and all risks and waive claims against the federal government and its related entities, except in the case of willful misconduct, for any injury, death, damage, or loss of property, revenue, or profits, whether direct, indirect, or consequential, arising from their participation in a competition, whether the injury, death, damage, or loss arises through negligence or otherwise.
3. Entrants must also agree to indemnify the federal government against third-party claims for damages arising from or related to competition activities. Entrants are not required to obtain liability insurance or demonstrate financial responsibility in order to participate in the competition.
4. By participating in the competition, each entrant agrees to comply with and abide by these rules and the decisions of IARPA, NIST and/or the individual judges, which shall be final and binding in all respects.

By participating in the competition, each entrant agrees to follow all applicable local, state, federal and country of residence laws and regulations.

4. Training Data

Provided as a part of this challenge is training data, which consists of 1) match scores and 2) feature vectors from K different algorithms for the purposes of fusion development. Participants are to take the provided training data to develop fusion scheme(s) for the K algorithm scores and feature vectors. As a part of the validation process, the developer's submission will be run on the training data, and the output(s) will be submitted to NIST, along with the implementation.

5. Validation

A validation package will be provided for the FOFRA 2018, and all participants must run their software through the provided validation package prior to submission. The purpose of validation is to ensure consistent algorithm output between the participant's execution and NIST's execution.

6. Reference Implementation

Included with the FOFRA 2018 validation package is a reference implementation of this API. The reference implementation provides very basic fusion schemes for example purposes only, but more importantly, it demonstrates mechanically how one could go about implementing, compiling, and building this API.

7. C++ APIs for score and template-level fusion (Open or closed-source implementation)

This section provides the C++ APIs and supporting data structures for score and template-level fusion.

7.1. Score fusion of verification scores and identification candidate lists

The software must implement the interface of

Table 3. It requires support for fusion of scores from named algorithms, for which prior training has occurred. Participants may implement either or both score/candidate list fusion. For any function that isn't implemented, the software shall return ReturnCode::NotImplemented.

Table 3 - Score level fusion of verification scores and identification candidate lists

	C++ code fragment	Remarks
1.	`class ScoreFuserInterface {`	

2.	`public:`
	`  static enum class Type {`
	`  Verification = 0,`
	`  Identification`
	`  };`	

3.	`  virtual ReturnStatus initialize(`
	`  const std::string &directory,`
	`  const ScoreFuserInterface::Type &type) = 0;`
		The function reads a pre-computed fusion scheme from the provided directory (e.g. pre-trained models), including any normalization information. The contents of the directory are developer-defined and are provided to NIST by the developer. Type indicates whether to load the fusion scheme for fusion of verification scores *(Type::Verification)* or identification candidate lists *(Type::Identification)*.

4.	`  virtual ReturnStatus fuseVerificationScores(`
	`  const ScoreSet &inputScores,`
	`  double &fusedScore) = 0;`
		Function to execute fusion of verification scores. Given K ≥ 2 scores, it produces one fused score. This function will be preceded by a call to *initialize(type=Type::Verification)*.

5.	`  virtual ReturnStatus fuseCandidateLists(`
	`  const std::vector<CandidateList> &inputLists,`
	`  CandidateList &fusedLists) = 0;`
		Function to execute fusion of candidate lists. Given K ≥ 2 candidate lists, it produces one output candidate list. The output *fusedLists* will initially be an empty vector. It is up to the implementation to add entries into the list. All input lists have the same length, L. The output lists may have variable length L ≤ x ≤ 2L. This funtion will be preceded by a call to *initialize(type=Type::Identification)*.

6.	`  static std::shared_ptr<ScoreFuserInterface>` `  getImplementation();`	Factory method to return a managed pointer to the `ScoreFuserInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `ScoreFuserInterface` object.

7.2. Template fusion for verification and identification

A developer may implement a scheme to fuse templates. Because matching of fused templates is potentially a non-trivial task – e.g. determining a distance metric – the implementation must also provide recognition support. The software must implement the interface of Table 4, which declares the following:

- a function to fuse (two or more) templates
- template comparison function to support biometric verification
- gallery construction and search functions to support identification

Participants may implement either or both template verification/identification. For any function that isn't implemented, the software shall return ReturnCode::NotImplemented.

NOTE: The identification part of this could be built by implementing a 1:N search as N 1:1 comparisons followed by a sort operation. Note however, that faster and more accurate solutions exist.

Table 4 – Template-level fuser class

	C++ code fragment	Remarks
1.	`class TemplateFuserInterface {`	

2.	`public:`
	`  static enum class Action {`
	`  Fuse = 0,`
	`  Verify,`
	`  Identify`
	`  };`	

3.	`  virtual ReturnStatus initialize(`
	`  const std::string &directory,`
	`  const TemplateFuserInterface::Action &action) = 0;`
		This function initializes the capability as specified via the *action* parameter:
		Action::Fuse - reads a pre-computed fusion scheme from the provided directory (e.g. pre-trained models), including any normalization information.
		Action::Verify - Initialize a verifier. The directory must contain sufficient information to identify which algorithms were fused and to load an appropriate verifier.
		Action::Identify - Initialize an identifier. The directory must contain sufficient information to identity which algorithms were fused and to load an appropriate identifier.
		The contents of the directory are developer-defined and are provided to NIST by the developer

4.	`  virtual ReturnStatus fuseTemplates(`
	`  const std::vector<Template> &inputTemplates,`
	`  Template &fusedTemplate) = 0;`
		Function to execute fusion. This function will be preceded by a call to *initialize(action = Action::Fuse)*. Given a vector of templates (each template is generated from a different algorithm), the implementation produces one template, which is the fusion between all input templates.

5.	`  virtual ReturnStatus verify(`
	`  const Template &enroll,`
	`  const Template &authentication,`
	`  double &score) = 0;`
		Given fused templates, the implementation must support one-to-one comparison of two such templates via this function. Compare an authentication template with an enrollment template and return a similarity score. This function will be preceded by a call to *initialize(action = Action::Verify).*

6.	`  virtual ReturnStatus createGallery(`
	`  const std::vector<Template> &templates,`
	`  const std::vector<uint32_t> &ids) = 0;`
		Create a gallery by adding a set of N identified templates to the implementation's internal gallery structure. This function should copy or otherwise process the input so that searches can follow. This function will be preceded by a call to *initialize(action = Action::Identify)*. The provided templates will contain N templates of N identities or people. ids[i] corresponds to templates[i].

7.	`virtual ReturnStatus search(`
	`  const Template &probe,`
	`  CandidateList &candidates);`
		Search a probe template against the gallery and fill the pre-allocated candidate list with hypothesized candidates. This function will be preceded by a call to *initialize(action=Action::Identify) and createGallery()*. The number of candidates to populate is specified by candidates.size().

8.	`static std::shared_ptr<TemplateFuserInterface> getImplementation();`	Factory method to return a managed pointer to the `TemplateFuserInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `TemplateFuserInterface` object.

7.3. C++ API data elements

The C++ data structures defined in this section are used in the C++ score- and template-level fusion APIs.

7.3.1. Data structure for return value of API function calls

Table 5 – Enumeration of return codes

Return code as C++ enumeration	Meaning
enum class ReturnCode {	
Success=0,	Success
ConfigError,	Error reading configuration files
ParseError,	Cannot parse the input data
TemplateCreationError,	Elective refusal to produce a fused template (e.g. too little information)
VerifTemplateError,	Either or both of the input templates were result of failed feature extraction
NumDataError,	The implementation cannot support the number of images
TemplateFormatError,	Template file is in an incorrect format or defective
InputLocationError,	Cannot locate the input data – the input files or names seem incorrect
MemoryError,	Memory allocation failed (e.g. out of memory)
NotImplemented,	Function is not implemented
NonCongruentVectors,	Vectors of different lengths passed to function expecting same lengths
VendorError	Vendor-defined failure. Vendor errors shall return this error code and document the specific failure in the ReturnStatus.info string from Table 6.
};	

Table 6 – ReturnStatus structure

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

7.3.2. Datatypes for sets of verification scores

A face verification algorithm, or a fusion algorithm, produce comparison scores, represented in double precision. When applied to N image pairs, N scores result and are encoded as:

Table 7 – Datatype for a set of similarity scores

C++ code fragment	Meaning
using ScoreSet = std::vector<double>;	A set of scores, some genuine, some impostor

7.3.3. Datatype for candidates

Table 8 – Candidate structure

C++ code fragment	Meaning
struct Candidate {	
double score;	Similarity score from recognition or fusion
uint32_t identity;	Identity hypothesis, a valid gallery identity label
};	
using Candidate = struct Candidate;	

7.3.4. Datatype for candidate lists

When searching a gallery containing N enrollments, biometric identification algorithms generally return L << N candidates, where L may be user specified, or a random variable. In FOFRA, the number of candidates will be user-specified, e.g. L = 50. An identification search shall produce a candidate list encoded as:

Table 9 – Datatype for a candidate list

C++ code fragment	Meaning
using CandidateList = std::vector<Candidate>;	A set of scores and hypothesized identities

7.3.5. Datatype for a single template

Templates, i.e. feature vectors, are represented as:

Table 10 – Datatype for a single template

C++ code fragment	Meaning
using Template = std::vector<double>;	Features for recognition

7.4. 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 fofra2018 fofra2018.cpp -L. -lfofra2018_scoreFusion_acme_0
```

The Standard C++ library should be used for development. The prototypes from this document will be written to a file "FOFRA2018.h" which will be included via

```
#include <fofra2018.h>
```

The header files will be made available to implementers at https://github.com/usnistgov/fofra2018.

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.).

7.5. Library naming requirements

Participants shall provide NIST with pre-compiled binary code and optionally source code if they choose to participate with an open-source implementation. The implementation, at a minimum, should be submitted in the form of a dynamically-linked library file.

The core library shall be named according to Table 11. 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 FOFRA test driver is the "core" library.

Table 11 – Implementation library filename convention

Form	libfofra2018_track_provider_sequence.ending				
Underscore delimited parts of the filename	libfofra2018	track	provider	sequence	ending
Description	First part of the name, required to be this.	"scoreFusion" for score-level fusion implementation "templateFusion" for template-level fusion implementation	Single word, non-infringing name of the main provider EXAMPLE: Acme	A one digit decimal identifier to start at 0 and incremented by 1 for each library sent to NIST.	.so
Example	libfofra2018_scoreFusion_acme_0.so				

7.6. Hardware Specifications

NIST will run submitted software on server-class machines equipped with multiple CPUs. The following list gives some details about the hardware of each CPU-only blade type:

- Dual Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz (10 cores each)[3]
- Dual Intel® Xeon® CPU E5-2680 v4 @ 2.4GHz (14 cores each)[3]

Implementations will not have access to any graphics processing units (GPUs) for this test.

[3] cat /proc/cpuinfo returns fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida arat epb pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdseed adx smap xsaveopt cqm_llc cqm_occup_llc

7.7. Single-thread requirement and 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.

7.8. 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, pre-computed fusion schemes/models, etc.
- doc/ - contains any participant-provided documentation regarding the submission
- validation/ - contains validation output, and in this case, is the fusion output of the provided training data
- src/ - contains implementation source code if the participant chooses to participate with an open-source implementation. This directory is optional.

8. R APIs for score and template-level fusion (Open-source implementation)

For developers who choose to submit open-source R code to NIST, the API is defined in this section.

8.1. Score-level fusion

8.1.1. Fusion of verification scores and identification candidate lists

Table 12 – R API for fusion of verification scores and identification candidate lists

	C++ code fragment	Remarks
1.	`read_verification_fuser <- function(directory)`	The function reads a pre-computed fusion scheme from the provided directory, including any normalization information. The contents of the directory are developer-defined and are assumed to be computed in the function above (if implemented) or by the developer using training data provided by NIST.
		The function returns a function that does fusion. myFuser <- function(scores, algorithms = NULL) The function accepts as input two vectors: – scores – a vector of scores, one from each of K = length(S) algorithms – algorithms – a vector of character strings identifying the K algorithms. This argument will be provided only on the first call to the function, to indicate the order of scores. Thereafter, the order will be same. Return value: The fused score, a scalar numeric.
2.	`read_identification_fuser <- function(directory)`	The function reads a pre-computed fusion scheme from the provided directory, including any normalization information. The contents of the directory are developer-defined and are assumed to be computed in the function above, if implemented, or by the developer using training data provided by NIST.

		The function returns a function that does fusion: myFuser <- function(clists, algorithms = NULL) It has semantics: - Input: clists - a list of K unnamed data.frames. Each data.frame is a sorted candidate list containing 2 columns and L ≥ 1 rows. The first column is named scores, the second column has name hypothesized_ids. The list of data.frames may be indexed as clists[[1]] and the scores as clists[[1]]$scores - algorithms – a vector of character strings identifying the K algorithms. This argument will be provided only on the first call to the function, to indicate the order of scores. Thereafter, the order will be same. Return value: A candidate list with two columns and number of rows on [L,2L], where L is the number of rows in the input data frames. The two columns must be named scores and hypothesized_ids.

8.2. Template-level fusion

The following sub-clauses describe three functions that must be implemented. Example R code is available in examples/fusion_example_template_level.R

8.2.1. Template fuser

	C++ code fragment	Remarks
1.	`read_template_fuser <-` `function(directory)`	Initialize the verifier, from permanent storage. Input: The directory must contain sufficient information to identity which algorithms were fused and to load an appropriate verifier. Return value: A function myTemplateFuser <- function(templates) that takes an unnamed list whose elements are templates (numeric vectors) and returns a single fused template (numeric vector). The number of templates to be fused is length(templates). A template may be indexed as templates[i]

8.2.2. Verifier

Given fused templates, the implementation must support one-to-one comparison of two such templates via:

Table 13 – R API for fusion of comparison of fused templates

	C++ code fragment	Remarks
1.	`read_template_verifier <-` `function(directory)`	Initialize the verifier, from permanent storage. Input: The directory must contain sufficient information to identity which algorithms were fused and to load an appropriate verifier. Return value: A function that takes exactly two templates (numeric vectors) and returns a similarity score (numeric scalar).

8.2.3. Identifier

The implementation most provide functions to support identification using fused templates as follows:

Table 14 – R API for fusion of comparison of fused templates

	C++ code fragment	Remarks
1.	`read_template_identifier <- function(directory)`	Initialize the identifier from permanent storage. Input: The directory must contain sufficient information to identity which algorithms were fused and to load an appropriate identifier. Return value: A list containing two named items. 1. Item "builder" is a function that accepts as input a. gvectors - a rectangular matrix whose columns are feature vectors b. gids - a set of identity labels c. N, the number of enrolled feature vectors = number of columns in the matrix = length(gids) d. Nfeatures, the feature dimensionality = number of rows in the matrix The function returns nothing, retaining data internally. 2. Item "searcher" is a function that accepts as input a. probe – a template i.e. numeric vector b. L – a candidate list length The function returns a candidate list of length L. A candidate list is a data.frame containing two named columns and L rows a. scores – numeric similarity values in descending order b. hypothesized_ids – integer values from the "gids" set provided to the builder function.

8.3. Source code naming requirements

The implementation should be submitted in the form of an R source file.

The core source file shall be named according to Table 11. Additional supplemental source files may be submitted that support this "core" R file (i.e. the "core" R file may have dependencies implemented in these other source files). The **only** R source file that will be explicitly called is the "core" source file.

Table 15 – Implementation source filename convention

Form	fofra2018_track_provider_sequence.ending				
Underscore delimited parts of the filename	fofra2018	track	Provider	sequence	ending
Description	First part of the name, required to be this.	"scoreFusion" for score-level fusion implementation "templateFusion" for template-level fusion implementation	Single word, non-infringing name of the main provider EXAMPLE: Acme	A one digit decimal identifier to start at 0 and incremented by 1 for each library sent to NIST.	.R
Example	fofra2018_scoreFusion_acme_0.R				

8.4. Submission folder hierarchy

Participant submissions shall contain the following folders at the top level

- src/ - contains implementation R source code
- config/ - contains all configuration and developer-defined data, pre-computed fusion schemes/models, etc.
- doc/ - contains any participant-provided documentation regarding the submission

9. General Liability Release

By participating in the competition, each entrant hereby agrees that:

1. IARPA shall not be responsible or liable for any losses, damages, or injuries of any kind (including death) resulting from participation in the competition or any competition-related activity, or from entrants' acceptance, receipt, possession, use, or misuse of any prize; and
2. Entrants will indemnify, defend, and hold harmless IARPA, NIST, and ODNI from and against all third party claims, actions, or proceedings of any kind and from any and all damages, liabilities, costs, and expenses relating to or arising from entrant's participation in the competition.

Without limiting the generality of the foregoing, IARPA is not responsible for incomplete, illegible, misdirected, misprinted, late, lost, postage-due, damaged, or stolen entries or prize notifications; or for lost, interrupted, inaccessible, or unavailable networks, servers, satellites, Internet Service Providers, websites, or other connections; or for miscommunications, failed, jumbled, scrambled, delayed, or misdirected computer, telephone, cable transmissions or other communications; or for any technical malfunctions, failures, difficulties, or other errors of any kind or nature; or for the incorrect or inaccurate capture of information, or the failure to capture any information. These rules cannot be modified except by IARPA. All decisions by IARPA regarding adherence to these rules are final. The invalidity or unenforceability of any provision of these rules shall not affect the validity or enforceability of any other provision. In the event that any provision is determined to be invalid or otherwise unenforceable or illegal, these rules shall otherwise remain in effect and shall be construed in accordance with their terms as if the invalid or illegal provision were not contained herein.

10. Warranties / Indemnification

By participating in the competition, each entrant represents, warrants, and covenants as follows:

1. The entrant – whether an individual, team or entity – is the sole author, creator, and owner of the submission or has secured usage rights to include as part of the software submission;
2. The submission is not the subject of any actual or threatened litigation or claim;
3. The submission does not and will not violate or infringe upon the intellectual property rights, privacy rights, publicity rights, or other legal rights of any third party;
4. The submission does not and will not contain any known harmful equipment that can cause injury or long term risks of exposure in humans; and
5. The Submission, and entrants' use of the Submission, does not and will not violate any applicable laws or regulations, including, without limitation, applicable export control laws and regulations of the U.S. and other jurisdictions.

If the Submission includes any third party works (such as third party content, equipment, or open source code), entrant must be able to provide, upon the request of IARPA, documentation of all appropriate licenses and releases for such third party works. If entrant cannot provide documentation of all required licenses and releases, IARPA reserves the right to disqualify the applicable Submission, or seek to secure the licenses and releases for the benefit of IARPA, and allow the applicable Submission to remain in the Competition. IARPA also

reserves all rights with respect to claims based on any damages caused by participant's failure to obtain such licenses and releases.

Entrants – whether an individual, a team or an entity – will indemnify, defend, and hold IARPA, NIST, and ODNI from and against all third party claims, actions, or proceedings of any kind and from any and all damages, liabilities, costs, and expenses relating to or arising from entrant's Submission or any breach or alleged breach of any of the representations, warranties, and covenants of entrant hereunder.

IARPA reserves the right to disqualify any Submission that IARPA, in its discretion, deems to violate the Rules. IARPA also reserves the right to amend these rules throughout the duration of the contest should extenuating circumstances arise.