Tutorials - OpenBR

Originalartikel

Backup

<html> <!–<div class=„toc“ data-spy=„affix“ role=„navigation“ aria-label=„main navigation“>

```
                                    <ul class="current">
                                        Table of Contents
                                        <ul>
                                                <li><a href="#quick-
start">Quick Start</a></li>
                                                <li><a href="#algorithms-in-
openbr">Algorithms in OpenBR</a></li>
                                                <li><a href="#training-
algorithms">Training Algorithms</a></li>
                                                <li><a href="#face-
recognition">Face Recognition</a></li>
                                                <li><a href="#age-
estimation">Age Estimation</a></li>
                                                <li><a href="#gender-
estimation">Gender Estimation</a></li>
                                                <li><a href="#openbr-as-a-
library">OpenBR as a Library</a></li>
                                                <li><a href="#the-evaluation-
harness">The Evaluation Harness</a></li>
                                        </ul>
                                    </ul>
                                </div>-->
                                <p>Welcome to OpenBR! Here we have a series of
tutorials designed to get you up to speed on what OpenBR is, how it works,
and its command line interface. These tutorials aren't meant to be completed
in a specific order so feel free to hop around. If you need help, feel free
to <a href="http://openbiometrics.org/docs/#help">contact us</a>.</p>
```

<hr/> This tutorial is meant to familiarize you with the ideas, objects and motivations behind OpenBR using some fun examples. Note that parts of this tutorial require a webcam. OpenBR is a C++ library built on top of Qt, OpenCV, and Eigen. It can either be used from the command line using the

```
br
```

application, or from interfacing with the C++ or C APIs. Using the

```
br
```

application is the easiest and fastest way to get started and this tutorial will use it for all of the examples. First, make sure that OpenBR has been installed on your system using the steps described in the installation section. Open up your terminal or command prompt and enter: <pre>$ br -gui -algorithm „Show(false)“ -enroll 0.webcam </pre> If everything has gone according to plan, your webcam should be on and capturing video. Congratulations, you are using OpenBR! Let's talk about what's happening in the above command.

```
-gui
```

,

```
-algorithm
```

, and

```
-enroll
```

are examples of OpenBR's flags and are used to specify instructions to

```
br
```

. OpenBR expects flags to be prepended by a

```
-
```

and arguments that follow the flags to be separated by spaces. Flags normally require a specific number of arguments. All of the possible flags and their values are documented here. Let's step through the individual arguments and values:

```
-gui
```

is the flag that tells OpenBR to open up a GUI window. Note that when

```
-gui
```

is used, it must be the first flag passed to

```
br
```

.

```
-algorithm
```

is one of the most important flags in OpenBR. It expects one argument, referred to as the algorithm string. This string determines the pipeline through which images and metadata propagate. It is composed of Transforms, which are described in detail later in this tutorial.

```
-enroll
```

reads files from a Gallery or a Format and enrolls them through the algorithm pipeline and serializes them to another Gallery or Format.

```
-enroll
```

takes one input argument (

```
0.webcam
```

in this example) and an optional output argument. OpenBR supports multiple formats including

```
.jpg
```

,

```
.png
```

,

```
.csv
```

, and

```
.xml
```

. The

```
.webcam
```

Format tells OpenBR to enroll frames from the computer's webcam. Let's try a slightly more complicated example. After all, OpenBR can do way more then just open webcams! Fire up the terminal again and enter: <pre>$ br -gui -algorithm „Cvt(Gray)+Show(false)“ -enroll 0.webcam </pre> Here, we took our normal BGR (OpenCV's alternative to RGB) image and converted it to a grayscale image simply by adding

```
Cvt(Gray)
```

to the algorithm string. Cvt, short for convert, is an example of an OpenBR Transform . Show is a Transform as

well. In fact, every algorithm string in OpenBR is just a series of Transforms joined to form a pipeline; even the

```
+
```

symbol is shorthand for a Pipe, another kind of OpenBR Transform. Typically, Transforms accept parameters. We specify

```
Gray
```

to Cvt as a runtime parameter to tell the Transform which color space to convert the image to. We also could have written

```
Cvt(HSV)
```

if we wanted to convert to the HSV color space or

```
Cvt(Luv)
```

if we wanted to convert to LUV. Parameters can be provided as key-value pairs or as keyless values (

```
Cvt(Gray)
```

is equivalent to

```
Cvt(colorSpace=Gray)
```

) . Note that if you are supplying values only, the parameters are expected to be supplied in the order they are defined. Try changing the algorithm string above to include

```
Show(true)
```

to see how modifying the parameters affects the output of the command (Hint: hit a key to cycle through the images). Let's make this example a little bit more exciting and relevant to OpenBR's biometric roots. Face detection is normally the first step in a face recognition algorithm. Let's perform face detection in OpenBR. Back in the terminal enter: <pre>$ br -gui -algorithm „Cvt(Gray)+Cascade(FrontalFace)+Draw(lineThickness=3)+Show(false)“ -enroll 0.webcam </pre> You're webcam should be open again but this time a bounding-box should have appeared around your face! We added two new Transforms to our string: <a

href=„http://openbiometrics.org/docs/plugin_docs/metadata/#cascadetransform“>Cascade and Draw. Let's walk through this Transform by Transform and see how it works: Cvt(Gray): Convert the image from BGR to grayscale. Grayscale is required for Cascade to work properly. Cascade(FrontalFace): This is a wrapper on the OpenCV Cascade Classification framework. It detects frontal faces using the

```
FrontalFace
```

model. Draw(lineThickness=3): Take the rectangles detected by Cascade and draw them onto the frame from the webcam.

```
lineThickness
```

determines the thickness of the drawn rectangle. Show(false): Show the image in a GUI window.

```
false
```

indicates the images should be shown in succession without waiting for a key press. Each Transform completes one task and the passes the output on to the next Transform. You can pipe together as many Transforms as you like, but note that certain Transforms have specific expectations for their input. You may be wondering what objects are actually being propagated through the algorithm pipeline. There are two objects that handle data in OpenBR: Files are typically used to store the path to a file on disk with associated metadata (in the form of key-value pairs). In the example above, we store the rectangles detected by Cascade as metadata which are then used by Draw for visualization. Templates are containers for images and <a

href=„http://openbiometrics.org/docs/api_docs/cpp_api/file/file/“>Files. Images in OpenBR are OpenCV Mats and are member variables of Templates. Templates can contain one or more images. If you want to learn more about the command line or all of the plugins and the key data structures, please refer to the linked documentation. The next few tutorials will explore algorithms and their use in more depth. <hr/> One advantage of OpenBR is the ease with which one can express biometrics algorithms in a consistent and simple way. In OpenBR, an algorithm string defines a technique for enrolling images and (optionally) a method for comparing them. Instead of storing the entire raw image for comparison, it is common practice to store an optimized representation, or template, of the image for the task at hand. We note for the sake of clarity that while the OpenBR object Template gets it name from this concept, template is a more general biometrics concept. The process of generating this optimized representation is called template enrollment or template generation. Given two templates, template comparison computes the similarity between them, where the higher values indicate more probable matches. Operationally, one seeks to generate templates that are small, accurate, and fast to compare. As previously noted, an algorithm is defined in OpenBR through an algorithm string. There are several advantages in mandating that algorithms are defined from strings, the most important of which are the following: It ensures good software development practices by forcibly decoupling the development of each step in an algorithm, facilitating the modification of algorithms and the re-use of individual steps. It spares the creation and maintenance of a lot of very similar header files that would otherwise be needed for each step in an algorithm (observe the absence of headers in

```
openbr/plugins
```

files). It allows for algorithm parameter tuning without recompiling. It is completely unambiguous, both the OpenBR interpreter and anyone familiar with the project can understand exactly what your algorithm does just from this description. OpenBR provides a syntax for setting plugin property values and creating concise algorithm strings. The relevant symbols are: <table><thead><tr><th>Symbol</th> <th>Meaning</th> </tr></thead><tbody readability=„9“><tr readability=„5“><td>PluginName(property1=value1,...propertyN=valueN)</td> <td>A plugin is described by its name (without the abstraction) and a list of properties and values. Properties of a plugin that are not specified are set to their default values.</td> </tr><tr readability=„5“><td>:</td> <td>Seperates template enrollment from template comparison. Enrollment is on the left of the colon in the algorithm string, while comparison is on the right. Defining an algorithm with a template comparison step is optional.</td> </tr><tr readability=„3“><td>+</td> <td>Abbreviation for a Pipe. Joins Transforms together and projects input through them in series. The output of a Transform to the left of + become the input of the Transform to the right.</td> </tr><tr readability=„4“><td>/</td> <td>Abbreviation for a Fork. Joins Transforms together and projects input through them in parallel. All Transforms

receive the same input, the output of which is concatenated together.</td> </tr><tr readability=„1“><td>{}</td> <td>Abbreviation for Cache. Cache the output of a plugin in memory for quick lookups later on.</td> </tr><tr readability=„3“><td><></td> <td>Abbreviation for LoadStore. Parameters for Transforms inside the brackets are stored on disk after training and loaded from disk before projection.</td> </tr><tr readability=„1“><td>()</td> <td>Order of operations. Change the order of operations using parantheses.</td> </tr></tbody></table> Let's look at some of the important parts of the codebase that make this possible: In

```
AlgorithmCore::init()
```

in

```
openbr/core/core.cpp
```

you can see the code for splitting the algorithm description at the colon. Shortly thereafter in this function we

```
make
```

the template generation and comparison objects. These

```
make
```

calls are defined in the public C++ plugin API and can also be called from end user code. Below we discuss some of the source code for

```
Transform::make
```

in

```
openbr/openbr_plugin.cpp
```

. Note, the

```
make
```

functions for other plugin types are similar in spirit and will not be covered. One of the first steps when converting the template generation description into Transforms is to replace the operators, like '+', with their full form: <pre>{ *Check for use of* '+' *as shorthand for Pipe(...) QStringList words = parse(str, '+'); if (words.size() > 1) return make(„Pipe([“ + words.join(„,“) + „])“, parent); } </pre> After operator expansion, the template enrollment description forms a tree, and the Transform is*

*constructed from this description recursively, starting at the root of the tree: <pre>Transform *transform = Factory<Transform>::make(„.“ + str); </pre> Let's use the algorithm in <code>scripts/helloWorld.sh</code> as an example. The algorithm is: <pre>Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.33,0.45)+CvtFloat+PCA(0.95):Dist(L2) </pre> Let's expand this using our new knowledge of OpenBR's algorithm syntax. First, the algorithm will be split into enrollment and comparison portions at the <code>:</code>. So enrollment becomes: <pre>Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.33,0.45)+CvtFloat+PCA(0.95) </pre> and comparison is: <pre>Dist(L2) </pre> On the enrollment side, Transforms joined by the <code>+</code> operators are converted into children of a Pipe. Thus, the enrollment algorithm is constructed as: <pre>Pipe(transforms=[Open,Cvt(Gray),Cascade(FrontalFace),ASEFEyes,Affine(128,128,0.33,0.45,CvtFloat,PCA(0.95)]) </pre> Low-level detail of the operations involved in this algorithm can be found in the project function implemented by each of these Transforms. To briefly summarize: <pre>1. Reads the image from disk 2. Converts the image to grayscale 3. Detects faces 4. Detects eyes in detected faces 5. Normalize the face with respect to rotation and scale using the eye locations 6. Converts the image to floating point format 7. Embeds the image in a PCA subspace trained on face images </pre> If you are familiar with face recognition, you will likely recognize this as the Eigenfaces<sup id=„fnref:1“/> algorithm. As a final note, the Eigenfaces algorithms uses the Euclidean distance (or L2-norm) to compare templates. Since OpenBR expects similarity values when comparing templates and not dissimilarity values, the DistDistance will return -log(distance+1) by default so that smaller distances (in the Euclidean sense) indicate higher similarity. Note that NegativeLogPlusOne distance also exists such that you can convert the output of any distance using the above function. <hr/> OpenBR makes it easy to create and train your own algorithms on custom datasets. Let's start with the algorithm string for the Eigenfaces<sup id=„fnref:1“/> algorithm described in the Algorithms tutorial. Recall that the algorithm is: <pre>$ Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.33,0.45)+CvtFloat+PCA(0.95) </pre> Suppose we want to train this algorithm on some data we gathered. First, let's examine some of the underlying principles of training in OpenBR. Recall that every algorithm is composed of Transforms but not all Transforms need to be trained. In our example, <code>Cvt(Gray)</code>, which converts the image to grayscale, does not need to be trained at all, and neither does <code>Open</code>, <code>ASEFEyes</code>, <code>Affine(128,128,0.33,0.45)</code> or <code>CvtFloat</code>. These are UntrainableTransforms (a subclass of Transform). <code>Cascade(FrontalFace)</code> is a special case; it is a Transform and therefore can be trained. However, we have passed it an argument indicating it should use pre-*

trained model (<code>FrontalFace</code>). Thus, <code>PCA(0.95)</code> is the only trainable Transform in our algorithm string. For the sake of completeness, we note that this transform is performing principal component analysis and retaining dimensions that account for 95% of the variance. Of course, we need to supply data to train our algorithm. Let's step back and consider the full training command. An example of this might be: <pre>$ br -algorithm „Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.33,0.45)+CvtFloat+PCA(0.95)“ -train training_data.csv EigenFaces </pre> Notice the <code>-train</code> flag used in the algorithm. -train requires at least one argument, a training Gallery. Note that certain Transforms expect labelled training data. While -train needs only a single gallery Gallery, more than one can be provided: <pre>$ br -algorithm „Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.33,0.45)+CvtFloat+PCA(0.95)“ -train training_data1.csv training_data2.csv EigenFaces </pre> -train has an optional second argument: the name for a trained model (<code>EigenFaces</code> in the example above). The optional model file is a serialized and compressed binary file that stores the parameters learned during algorithm training. The model file should only be considered optional when your algorithm string uses a LoadStoreTransform (discussed in depth later in this tutorial). Otherwise, none of the parameters learned during algorithm training will be stored! As was briefly discussed above, each Transform in is either trainable or not (in our case only <code>PCA(0.95)</code> is trainable). At train time, the training data is projected through each UntrainableTransform in sequence, just as it would be at test time. When the data reaches a trainable transform, the train method is called with the data projected through the preceding Transforms as its input. After training, the project method is called on the newly trained transform and the data continues to propagate through the algorithm. After training is complete the algorithm is serialized to a model file (if you have specified one). The algorithm string is serialized first such that the algorithm can be recreated, followed by the parameters for each transform using the store method. Note that only trainable Transforms need to implement store because UntrainableTransforms can be recreated solely from their algorithm string descriptions. We can then -enroll

*images using the trained algorithm by replacing the algorithm string with the model file: <pre>$ br -algorithm EigenFaces -enroll enroll_data.csv enroll_data.gal </pre> In the case that we want our training and testing algorithms to be different, we can use LoadStoreTransform to serialize specific parts of the algorithm string. Reusing our EigenFaces example, we could specify that only <code>CvtFloat</code> and <code>PCA(0.95)</code> should be serialized to the model, allowing the other algorithmic steps to be specified at test time. The command to accomplish this is: <pre>$ br -algorithm "Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.33,0.45)+<CvtFloat+PCA(0.95),EigenFaces>" -train training_data.csv </pre> Recall from the Algorithms tutorial that <code><></code> is shorthand for a LoadStoreTransform. Also note that the LoadStoreTransform takes two arguments: the algorithm string and an optional model name. If a name is not provided, a random name is created. Using this model would like this: <pre>$ br -algorithm "Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.2,0.55)+<EigenFaces>" </pre> Since we haven't serialized that portion of the algorithm, the parameters of <code>Affine</code>, for example, can now be changed at test time! Note that, in this contrived example, changing the <code>Affine</code> parameters will severely degrade performance. As a final note, when a LoadStoreTransform is present in the algorithm string used for training, OpenBR will not overwrite the specified model file if it already exists. Instead, it will load the old model file and treat the associated Transforms as untrainable (as they have already been trained!). This can helpful when you want to isolate a particular algorithmic step (e.g. to explore parameters) but don't want to retrain every part of the algorithm during each iteration. Now that we've covered training a generic algorithm, the next tutorials will cover popular use cases supported by OpenBR including FaceRecognition, Age Estimation, and Gender Estimation. <hr/> This tutorial gives an example on how to perform face recognition in OpenBR. OpenBR implements the 4SF<sup id="fnref:2"/> algorithm to perform face recognition. Please read the paper for more specific algorithm details. To start, lets run face recognition from the command line. Open the terminal and enter <pre>$ br -algorithm FaceRecognition \ -compare ../data/MEDS/img/S354-01-t10_01.jpg ../data/MEDS/img/S354-02-t10_01.jpg \ -compare ../data/MEDS/img/S354-01-t10_01.jpg ../data/MEDS/img/S386-04-t10_01.jpg </pre> Easy enough? You should see results printed to terminal that look like <pre>$ Set algorithm to FaceRecognition $ Loading /usr/local/share/openbr/models/algorithms/FaceRecognition $ Loading /usr/local/share/openbr/models/transforms*FaceRecognitionExtraction $ Loading /usr/local/share/openbr/models/transforms*FaceRecognitionEmbedding $ Loading /usr/local/share/openbr/models/transforms*FaceRecognitionQuantization $ Comparing ../data/MEDS/img/S354-01-t10_01.jpg and ../data/MEDS/img/S354-02-t10_01.jpg $ Enrolling ../data/MEDS/img/S354-01-t10_01.jpg to S354-01-t10_01r7Rv4W.mem $ 100.00% ELAPSED=00:00:00 REMAINING=00:00:00 COUNT=1 $ 100.00% ELAPSED=00:00:00 REMAINING=00:00:00 COUNT=1 $ 1.8812 $ Comparing ../data/MEDS/img/S354-01-t10_01.jpg and ../data/MEDS/img/S386-04-t10_01.jpg $ Enrolling ../data/MEDS/img/S354-01-t10_01.jpg to S354-01-t10_01r7Rv4W.mem $ 100.00% ELAPSED=00:00:00 REMAINING=00:00:00 COUNT=1 $ 100.00% ELAPSED=00:00:00 REMAINING=00:00:00 COUNT=1 $ 0.571219 </pre> So, what is

```
FaceRecognition
```

? It's an abbrieviation to simplify execution of the algorithm. All of the algorithm abbreviations are located in

```
openbr/plugins/core/algorithms.cpp
```

. It is also possible to: <ul readability=„11“><li readability=„6“> Evaluate face recognition performance (Note that this requires R to be installed): <pre>$ br -algorithm FaceRecognition -path ../data/MEDS/img/ \ -enroll ../data/MEDS/sigset/MEDS_frontal_target.xml target.gal \ -enroll ../data/MEDS/sigset/MEDS_frontal_query.xml query.gal \ -compare target.gal query.gal scores.mtx \ -makeMask ../data/MEDS/sigset/MEDS_frontal_target.xml ../data/MEDS/sigset/MEDS_frontal_query.xml MEDS.mask \ -eval scores.mtx MEDS.mask Algorithm_Dataset/FaceRecognition_MEDS.csv \ -plot Algorithm_Dataset/FaceRecognition_MEDS.csv MEDS </pre> <li readability=„3“> Perform a 1:N face recognition search: <pre>$ br -algorithm FaceRecognition -enrollAll -enroll ../data/MEDS/img 'meds.gal' $ br -algorithm FaceRecognition -compare meds.gal ../data/MEDS/img/S001-01-t10_01.jpg match_scores.csv </pre> <li readability=„16“> Train a new face recognition algorithm (on a different dataset): <pre>$ br -algorithm 'Open+Cvt(Gray)+Cascade(FrontalFace)+ASEFEyes+Affine(128,128,0.33,0.45)+(Grid(10,10)+SIFTDescriptor(12)+ByRow)/(Blur(1.1)+Gamma(0.2)+DoG(1,2)+ContrastEq(0.1,10)+LBP(1,2)+RectRegions(8,8,6,6)+Hist(59))+PCA(0.95)+Normalize(L2)+Dup(12)+RndSubspace(0.05,1)+LDA(0.98)+Cat+PCA(0.95)+Normalize(L1)+Quantize:NegativeLogPlusOne(ByteL1)' -train ../data/ATT/img FaceRecognitionATT </pre>   The entire command line API is documented here. <hr/> Age estimation is very similar in spirit to Face Recognition and will be covered in far less detail. To perform age estimation from the command line you can run: <pre>$ br -algorithm AgeEstimation \

1. enroll ../data/MEDS/img/S354-01-t10_01.jpg ../data/MEDS/img/S001-01-t10_01.jpg metadata.csv

</pre> The results will be stored in metadata.csv under the key 'Age'. Remember from the Face Recognition tutorial that

```
AgeEstimation
```

is just an abbreviation for the full algorithm description. The source code to run age estimation as an application is in

```
app/examples/age_estimation.cpp
```

 <hr/> As with age estimation, gender estimation is very similar in spirit to Face Recognition and will be covered in far less detail. To perform gender estimation from the command line you can run: <pre>$ br -algorithm GenderEstimation \

1. enroll ../data/MEDS/img/S354-01-t10_01.jpg ../data/MEDS/img/S001-01-t10_01.jpg metadata.csv

</pre> The results will be stored in metadata.csv under the key 'Gender'. Remember from the Face Recognition tutorial that

```
GenderEstimation
```

is just an abbreviation for the full algorithm description. The source code to run gender estimation as an application is in

```
app/examples/gender_estimation.cpp
```

 <hr/> OpenBR exposes a C++ API that can be embedded into your own applications. Let's step through the example code at

```
app/example/face_recognition.cpp
```

and learn about using OpenBR as a library. Our main function starts with: <pre>br::Context::initialize(argc, argv) </pre> This is the first step in any OpenBR-based application, it initializes the global context. <pre>QSharedPointer<br::Transform> transform = br::Transform::fromAlgorithm(„FaceRecognition“); QSharedPointer<br::Distance> distance = br::Distance::fromAlgorithm(„FaceRecognition“); </pre> Here, we split our algorithm into enrollment (Transform::fromAlgorithm) and comparison (Distance::fromAlgorithm) <pre>br::Template queryA(„../data/MEDS/img/S354-01-t10_01.jpg“); br::Template queryB(„../data/MEDS/img/S382-08-t10_01.jpg“); br::Template target(„../data/MEDS/img/S354-02-t10_01.jpg“); </pre> These lines create our Templates for enrollment. At this point, the Templates simply store the file path to the specified image on disk. In this example,

```
queryA
```

depicts the same person as

```
target
```

(often referred to as a genuine match) and

```
queryB
```

depicts a different person from

```
target
```

(often referred to as an impostor match). <pre>queryA >> *transform; queryB >> *transform; target >> *transform; </pre>

```
&gt;&gt;
```

is a convienience operator for enrolling Templates in Transforms. Thus, at this stage, our Templates now store the images enrolled via the

```
FaceRecognition
```

algorithm. <pre>float comparisonA = distance->compare(target, queryA); float comparisonB = distance->compare(target, queryB); </pre> We then compare our query Templates against the target Template. The result is a floating point value indicating similarity. <pre>printf(„Genuine match score: %.3f\n“, comparisonA); printf(„Impostor match score: %.3f\n“, comparisonB); </pre> After printing the results, you can see that

```
comparisonA
```

(between

```
queryA
```

and

```
target
```

) has a higher similarity score then

```
comparisonB
```

, which is exactly what we expect! <pre>br::Context::finalize(); </pre> The last line in any OpenBR application has to be call to

```
finalize
```

. This functions performs the clean up of OpenBR. That's it! You can now embed face recognition into all of your applications. <hr/> OpenBR implements a complete, NIST compliant, evaluation harness for evaluating face recognition, face detection, and facial landmarking. The goal is to provide a consistent environment for the repeatable evaluation of algorithms to the academic and open source communities. To accompish this OpenBR defines the following portions of the biometrics evaluation environment (BEE) standard- <ul readability=„5“><li readability=„3“> Signature set - A signature set (or sigset) is an XML file-list specified on page 9 of the MBGC File Overview and is implemented in xmlGallery. Sigsets are identified with an

```
.xml
```

extension. <li readability=„3“> Similarity matrix - A similarity matrix (or simmat) is a binary score matrix specified on page 12 of the MBGC File Overview and is implemented in mtxOutput. Simmats are identified with a

```
.mtx
```

extension. See br_eval for more information. <li readability=„4“> Mask matrix - A mask matrix (or mask) is a binary matrix specified on page 14 of the MBGC File Overview identifying the genuine and impostor matches within a corresponding simmat. Masks are identified with a

```
.mask
```

extension. See br_make_mask and br_combine_masks for more information.   The evaluation harness is also accessible from the command line. See -eval, -evalDetection, -evalLandmarking, -evalClassification, -evalClustering, or -evalRegression for relevant information. </html>