BOSARIS Toolkit Users' Manual

v1.0

27th March 2012

The BOSARIS Toolkit is a collection of functions and classes in Matlab that can be used to calibrate, fuse and plot scores from speaker recognition (or other fields in which scores are used to test the hypothesis that two samples are from the same source) trials involving a model and a test segment. The toolkit was written at the BOSARIS2010 workshop which took place at the University of Technology in Brno, Czech Republic from 5 July to 6 August 2010. See the *User Guide* (available on the toolkit website)[1] for a discussion of the theory behind the toolkit and descriptions of some of the algorithms used.

1 Getting Started

This section gives a brief description of how to start using the BOSARIS toolkit.

Start by reading the User Guide to understand what the toolkit can do, how it does it, and when you'd want to use it.

Note: you need to have Matlab version R2008a or later to run the code.

Download and uncompress either the .zip version or the .tgz version of the toolkit (these two archives have identical contents). If you are running Windows, you probably want to download the .zip version.

To use the code in Matlab, add the following line to your scripts, or enter it at the Matlab prompt:

`addpath(genpath(` *path_to_bosaris_toolkit* `))`

Start by running the two demo scripts, *demo_main.m* and *demo_with_files.m*, in the *demo* directory to see that everything is working. If you want to see the plots produced by the demo scripts, you need to run Mablab in graphical mode. You can get an idea of some of the uses of the toolkit by looking at the demo code. Remember to add the BOSARIS toolkit to your Matlab path before running the demo.

Once you have run the demo code, read this manual to see what functions are available, and then write your own scripts to experiment with the toolkit.

2 Terminology

Because the toolkit was initially written for a NIST Speaker Recognition Evaluation, terminology from these evaluations is used in the documentation and code. This section explains the meanings of these terms to make the toolkit more accessible to users not familiar with SREs and/or speaker recognition.

[1] `https://sites.google.com/site/bosaristoolkit/`

In a speaker recognition evaluation, participants have to produce scores for trials involving speech segments in an (unlabelled) evaluation database. The participants must perform each trial independently. A trial is described by a pair of identifiers. The left hand identifier is called the *model* and the right hand identifier is called the *test segment*. A model is a group of one or more *enrolment segments* and the mapping from the model name to the list of enrolment segments is given by a *train file*. All the enrolment segments for a model are known to be from the same speaker. The question to be answered in each trial is whether the test segment is also from the same speaker. The more confident a system is that the same-speaker hypothesis is true for a trial, the larger the value it should assign to the score for that trial.

But all that the users of the toolkit need to know is that a score is for a trial which is a pair of identifiers. If either (or both) of the sides of the trial are groups (of speech segments for example), then the user just needs to create a unique identifier for the group.

The following glossary describes how speaker recognition terminology used in the toolkit code and this manual should be interpreted. Terms like *miss*, *false alarm*, *DCF* and *PRBEP* are explained in the User Guide and the explanations are not repeated here.

segment A speech recording assumed to be from a single speaker.

trial A test (run by a recognizer) to determine whether two (groups of) input segments are from the same speaker. If they are, then the trial is a target trial; if they are not, then the trial is a non-target trial.

model The identifier for the left hand side of the trial i.e. the first name to appear in the pair describing the trial.

test segment The identifier for the right hand side of the trial i.e. the second name to appear in the pair describing the trial.

index A description of a set of trials in terms of pairs of identifiers (*model* and *test segment*). It can be seen as a list containing pairs and an indication of whether each pair forms a trial or not.

key A description of a set of trials that includes answers. Like an index, it indicates whether pairs form a trial, and for those that do, it indicates whether the trial is a target trial or a non-target trial.

system Used to describe something that produces scores for trials. When statistics are calculated for a system, they are calculated using the collection of scores output by that system, given an index or trial list.

quality measure A quality measure can be seen as a score for a segment instead of a trial. The toolkit allows for quality measures to be used during fusion (of scores). Some examples for speech are duration and signal to noise ratio.

dev Short for *development database*. This is a supervised database that can be used to train calibration and fusion functions.

eval Short for *evaluation database*. This is the unsupervised database that the user wishes to study using the toolkit.

3 Directories

We start our description of the toolkit by listing the top-level subdirectories contained in the toolkit distribution and describing their contents. Most of the sections in this document correspond to one of these directories.

The directories (in alphabetical order) are:

calibration Code to calibrate scores. (See section 6).

classes All the directories for the classes (see section 4) of the toolkit are stored in this common directory.

demo Contains two demo scripts (see section 11) and functions called by the demo scripts.

docs Contains this users' manual.

fusion Code to fuse calibrated scores and quality measures. (See section 7).

plotting Example scripts for using the plotting classes to make DET and normalized Bayes error-rate plots. (See section 8).

stats Functions used to calculate dcf, eer, etc. (See section 9).

utility_funcs Helper functions for the code in the other directories. (See section 10).

4 Classes

Next we describe the classes implemented by the toolkit. The toolkit has an object orientated API to (hopefully) make data manipulation easier for the user. There are ten classes: Ndx, Key, Scores, Seg_Quality, Quality, Results, Id_Map, Logger, Det_Plot and Norm_DCF_Plot. Each of these classes is described below in its own subsection.

4.1 Ndx

This class stores trial index information i.e. which combinations of model and test segment should be evaluated by a system (which should produce scores for these trials).

Indexes can also be used for encoding properties of trials. An evaluation could be run using index A, but index B describes the subset of trials in A that are between male speakers. Using index B to select scores from a score object that correspond to index A will give a set of scores for male trials without having to rerun the recognizer. Or alternatively, if the user has produced scores for two different trial lists but would now like to study the performance for the combined trial list, indexes A and B can be merged to create a larger index which can be used with merged score objects (with the two score objects being merged corresponding to indexes A and B).

Indexes have three fields:

modelset A list of model names (m of them).

segset A list of test segment names (n of them).

trialmask An m-by-n matrix of booleans. If trialmask(m, n) is true, then the mth model and the nth test segment form a trial.

The class has six dynamic methods:

filter Removes some of the models and test segments and the corresponding rows and columns in the *trialmask* matrix. This method takes three arguments: a list of models, a list of test segments, and a boolean indicating whether the lists of models and test segments must be retained in or discarded from the Ndx. It returns the filtered Ndx.

save Saves the object to a file. The type of the file is determined by the extension. It takes one argument: the name of the output file.

save_hdf5 Saves the object to an HDF5 file. It takes one argument: the name of the output file.

save_mat Saves the object to a Matlab mat file. It takes one argument: the name of the output file.

save_txt Saves the object to a text file. It takes one argument: the name of the output file.

validate Checks an Ndx object against a list of class invariants.

and five static methods:

merge Merges Ndxs to form a new Ndx. It either takes two Ndxs as input or an object array of Ndxs. The indexes being merged do not need to have disjoint modelsets and segsets.

read Creates an instance of the class from the contents of a file. The file type is determined from the extension. It takes one argument: the name of the input file.

read_hdf5 Creates an instance of the class from the contents of an HDF5 file. It takes one argument: the name of the input file.

read_mat Creates an instance of the class from the contents of a mat file. It takes one argument: the name of the input file.

read_txt Creates an instance of the class from the contents of a text file. It takes one argument: the name of the input file.

The following constructors are available:

1. `Ndx()`
2. `Ndx(ndx_struct)`
3. `Ndx(model_list,seg_list)`
4. `Ndx(model_list,seg_list,trial_mask)`

with the following behaviours:

1. Creates an empty object.
2. Constructs an object from a struct that contains the same fields as the object.
3. Constructs an index having the given modelset and segset and with the trialmask set to all true.
4. Constructs an index having the given modelset, segset and trialmask.

4.2 Key

This class stores key information i.e. information about which trials are target trials and which trials are non-target trials.

It has four fields:

modelset A list of model names (m of them).

segset A list of test segment names (n of them).

tar An m-by-n matrix of booleans. If $\mathrm{tar}(m, n)$ is true, then the mth model and the nth test segment form a target trial.

non An m-by-n matrix of booleans. If $\mathrm{non}(m, n)$ is true, then the mth model and the nth test segment form a non-target trial.

The class has seven dynamic methods:

filter Removes some of the models and test segments and the corresponding rows and columns in the *tar* and *non* matrices. This method takes three arguments: a list of models, a list of test segments, and a boolean indicating whether the lists of models and test segments must be retained in or discarded from the key.

save Saves the object to a file. The type of the file is determined by the extension. It takes one argument: the name of the output file.

save_hdf5 Saves the object to an HDF5 file. This method takes one argument: the name of the output file.

save_mat Saves the object to a Matlab mat file. It takes one argument: the name of the output file.

save_txt Saves the object to a text file. It takes one argument: the name of the output file.

to_ndx Returns an Ndx constructed from the Key i.e. the trialmask is produced by *or*ing the *tar* and *non* matrices. This method takes no arguments.

validate Checks a Key object against a list of class invariants.

and five static methods:

merge Merges keys to form a new key. It either takes two keys as input or an object array of keys. The keys being merged do not need to have disjoint modelsets and segsets.

read Creates an instance of the class from the contents of a file. The file type is determined from the extension. It takes one argument: the name of the input file.

read_hdf5 Creates an instance of the class from the contents of an HDF5 file. This method takes one argument: the name of the input file.

read_mat Creates an instance of the class from the contents of a mat file. It takes one argument: the name of the input file.

read_txt Creates an instance of the class from the contents of a text file. It takes one argument: the name of the input file.

The following constructors are available:

1. `Key()`
2. `Key(key_struct)`
3. `Key(model_list,seg_list,target_mask,nontarget_mask)`

with the following behaviours:

1. Creates an empty object.
2. Constructs an object from a struct that contains the same fields as the object.
3. Constructs a key having the given modelset, segset, target_mask and non-target_mask.

4.3 Scores

This class stores scores of trials (specified by an index). The class duplicates the type of information contained in an index so that it is a self contained i.e. it does not have to be bundled with a separate index file to give the scores meaning.

It has four fields:

modelset A list of model names (m of them).

segset A list of test segment names (n of them).

scoremask An m-by-n matrix of booleans. If scoremask(m, n) is true, then there is a score for the trial involving the mth model and the nth test segment.

scoremat An m-by-n matrix of real numbers. If scoremask(m, n) is true, scoremat(m, n) is the score for the trial involving the mth model and nth test segment. Otherwise; that entry should be ignored.

The class has ten dynamic methods:

align_with_ndx Re-orders the modelset and segset (and the scoremat and scoremask) to match the order in a Key or Ndx. Any models or test segments missing from the Key or Ndx are dropped from the Scores object (along with corresponding rows and columns in the scoremat and scoremask). This method takes one argument: a Key or Ndx object.

filter Removes some of the models and test segments and the corresponding rows and columns in the *scoremat* and *scoremask* matrices. This method takes three arguments: a list of models, a list of test segments, and a boolean indicating whether the lists of models and test segments must be retained in or discarded from the Scores object.

get_tar_non Returns a vector of target scores and a vector of non-target scores corresponding to target and non-target trials specified by a key. It takes one argument—a Key object.

save Saves the object to a file. The type of the file is determined by the extension. It takes one argument: the name of the output file.

save_hdf5 Saves the object to an HDF5 file. This method takes one argument: the name of the output file.

save_mat Saves the object to a Matlab mat file. It takes one argument: the name of the output file.

save_txt Saves the object to a text file. It takes one argument: the name of the output file.

set_missing_to_value Sets all scores for which the *trialmask* is true but the *scoremask* is false to the same value, supplied by the user. It takes two inputs: a Key or Ndx and the value for missing scores. If a Key is given, the method converts it to an Ndx internally to produce the trialmask. The output is a new Scores object.

transform Transforms the scores by sending them through a function. It takes one input: a function handle. The output is a Scores object with each score in *scoremat* being the result of applying the function to the corresponding score in the input *scoremat*.

validate Checks a Scores object against a list of class invariants.

and five static methods:

merge Merges Scores objects to form a new Scores object. It either takes two Scores objects as input or an object array of Scores. The score objects being merged do not need to have disjoint modelsets and segsets.

read Creates an instance of the class from the contents of a file. The file type is determined from the extension. It takes one argument: the name of the input file.

read_hdf5 Creates an instance of the class from the contents of an HDF5 file. This method takes one argument: the name of the input file.

read_mat Creates an instance of the class from the contents of a mat file. It takes one argument: the name of the input file.

read_txt Creates an instance of the class from the contents of a text file. It takes one argument: the name of the input file.

The following constructors are available:

1. `Scores()`
2. `Scores(score_struct)`
3. `Scores(score_matrix,model_list,seg_list)`
4. `Scores(score_matrix,model_list,seg_list,score_mask)`

with the following behaviours:

1. Creates an empty object.
2. Constructs a Scores object from a struct that contains the same fields as the object.
3. Constructs a Scores object having the given scoremat, modelset and segset and with the scoremask set to all true.
4. Constructs a Scores object having the given scoremat, modelset, segset and scoremask.

4.4 Seg_Quality

This class stores quality measures for segments (or models). It can store several different quality measures for each segment (so you don't need a separate file for each type of quality measure), but for each segment, the number of quality measures, meaning of each one and their order of placement must be identical.

The class has two fields:

ids A list of names (m of them).

values A q-by-m matrix of quality measures. values$(:, m)$ is a vector of q quality measures for the mth segment.

It has five dynamic methods:

align_with_ids Re-orders the *ids* (and the *values*) to match the order in a list of ids. Any ids missing from the input list are dropped from the object (along with corresponding columns in *values*). This method takes one argument: a list of ids.

save Saves the object to a file. The type of the file is determined by the extension. It takes one argument: the name of the output file.

save_hdf5 Saves the object to an HDF5 file. This method takes one argument: the name of the output file.

save_mat Saves the object to a Matlab mat file. It takes one argument: the name of the output file.

save_txt Saves the object to a text file. It takes one argument: the name of the output file.

validate Checks a Seg_Quality object against a list of class invariants.

and four static methods:

merge Merges two Seg_Quality objects. This method takes two inputs—the two Seg_Quality objects to merge.

read Creates an instance of the class from the contents of a file. The file type is determined from the extension. It takes one argument: the name of the input file.

read_hdf5 Creates an instance of the class from the contents of an HDF5 file. This method takes one argument: the name of the input file.

read_mat Creates an instance of the class from the contents of a mat file. It takes one argument: the name of the input file.

read_txt Creates an instance of the class from the contents of a text file. It takes one argument: the name of the input file.

The following constructors are available:

1. `Seg_Quality()`
2. `Seg_Quality(seg_quality_struct)`
3. `Seg_Quality(ids,values)`

with the following behaviours:

1. Creates an empty object.
2. Constructs a Seg_Quality object from a struct that contains the same fields as the object.
3. Constructs a Seg_Quality object having the given ids and values.

4.5 Quality

This class stores quality measures for both models and test segments at the same time. It is used by the fusion code for doing fusion of scores with quality measures. In this fusion code, segment based quality measures need to be expanded to trial based quality measures and having the quality measures for the two sides of the trial in this format makes the expansion easier.

Rather use the *Seg_Quality* class for storing quality data. Where a *Quality* object is required by the code, it can be created from *Seg_Quality* objects using utility functions.

The Quality class has seven fields:

modelset A list of model names (m of them).

segset A list of test segment names (n of them).

hasmodel A vector (of length m) indicating whether there are quality measures for the mth model.

hasseg A vector (of length n) indicating whether there are quality measures for the nth segment.

modelQ A q-by-m matrix of quality measures. modelQ$(:, m)$ is a vector of q quality measures for the mth model.

segQ A q-by-n matrix of quality measures. segQ$(:, n)$ is a vector of q quality measures for the nth test segment.

trialmask A matrix used by the code when expanding the segment based quality measures to trial based quality measures. This matrix indicates for which trials there are quality measures.

It has five dynamic methods:

align_with_ndx Re-orders the modelset and segset (and the dependent fields) to match the order in a Key or Ndx. Any models or test segments missing from the Key or Ndx are dropped from the Quality object (along with corresponding rows and columns in the dependent fields). This method takes one argument: a Key or Ndx object.

save Saves the object to a file. The type of the file is determined by the extension. It takes one argument: the name of the output file.

save_hdf5 Saves the object to an HDF5 file. This method takes one argument: the name of the output file.

save_mat Saves the object to a Matlab mat file. It takes one argument: the name of the output file.

validate Checks a Quality object against a list of class invariants.

and four static methods:

merge Merges two Quality objects. This method takes two inputs—the two Quality objects to merge.

read Creates an instance of the class from the contents of a file. The file type is determined from the extension. It takes one argument: the name of the input file.

read_hdf5 Creates an instance of the class from the contents of an HDF5 file. This method takes one argument: the name of the input file.

read_mat Creates an instance of the class from the contents of a mat file. It takes one argument: the name of the input file.

Note that Quality doesn't have a text format, so there are no *read_txt* and *save_txt* methods.

The following constructors are available:

1. `Quality()`
2. `Quality(quality_struct)`
3. `Quality(model_list,seg_list,model_quality_values,seg_quality_values)`
4. `Quality(model_list,seg_list,model_quality_values,seg_quality_values, has_model,has_seg)`

5. `Quality(model_list,seg_list,model_quality_values,seg_quality_values, has_model,has_seg,trial_mask)`

with the following behaviours:

1. Creates an empty object.
2. Constructs a Quality object from a struct that contains the same fields as the object.
3. Constructs a Quality object having the given modelset, segset, modelQ and segQ and with hasmodel, hasseg and trialmask set to all true.
4. Constructs a Quality object having the given modelset, segset, modelQ, segQ, hasmodel, and hasseg and with trialmask set to all true.
5. Constructs a Quality object having the given modelset, segset, modelQ, segQ, hasmodel, hasseg and trialmask.

4.6 Results

This class contains methods that calculate various measures of goodness of the system. This class is just syntactic sugar. The *stats functions* (see section 9) can be used to do the same calculations, and the methods of this class call those functions internally.

The class methods are:

get_act_dcf Given an effective prior, returns the (unnormalized) actual DCF value for the system.

get_min_dcf Given an effective prior, returns the (unnormalized) minimum DCF value for the system.

get_norm_act_dcf Given an effective prior, returns the normalized actual DCF value for the system.

get_norm_min_dcf Given an effective prior, returns the normalized minimum DCF value for the system.

get_prbep Returns the precision-recall break even point.

get_eer Returns the equal error rate.

num_tar Returns the number of target scores.

num_non Returns the number of non-target scores.

The following two constructors are available:

1. `Results(scr,key)`
2. `Results(tar,non)`

The first form takes a Scores object and a Key object as input, and the second form takes an array of target scores and an array of non-target scores.

4.7 Id_Map

This class stores two lists of strings and can be used to map between them. One use for it is to store a *train file* i.e. a file that gives the mapping between model names and enrolment segments, in the case of one enrolment segment per model. Duplicate entries are permitted in both lists. The two lists must be the same length.

The class has two fields:

leftids A list of strings.

rightids Another list of strings.

The class has nine dynamic methods:

filter_on_left Removes some of the pairs in the map. This method takes two arguments: a list of strings which will be compared with the leftids, and a boolean indicating whether the list's strings must be retained in or discarded from the map. It returns the filtered map.

filter_on_right Removes some of the pairs in the map. This method takes two arguments: a list of strings which will be compared with the rightids, and a boolean indicating whether the list's strings must be retained in or discarded from the map. It returns the filtered map.

map_left_to_right Takes in a list of strings, compares them with the leftids and returns the corresponding rightids.

map_right_to_left Takes in a list of strings, compares them with the rightids and returns the corresponding leftids.

save Saves the object to a file. The type of the file is determined by the extension. It takes one argument: the name of the output file.

save_hdf5 Saves the object to an HDF5 file. This method takes one argument: the name of the output file.

save_mat Saves the object to a Matlab mat file. It takes one argument: the name of the output file.

save_txt Saves the object to a text file. It takes one argument: the name of the output file.

validate Checks an Id_Map object against a list of class invariants.

and five static methods:

merge Merges two Id_Map objects. This method takes two inputs—the two Id_Map objects to merge. The leftids of the second map is appended to the leftids of the first and the rightids of the second is appended to the rightids of the first.

read Creates an instance of the class from the contents of a file. The file type is determined from the extension. It takes one argument: the name of the input file.

read_hdf5 Creates an instance of the class from the contents of an HDF5 file. This method takes one argument: the name of the input file.

read_mat Creates an instance of the class from the contents of a mat file. It takes one argument: the name of the input file.

read_txt Creates an instance of the class from the contents of a text file. It takes one argument: the name of the input file.

The following constructors are available:

1. `Id_Map()`
2. `Id_Map(idmap_struct)`
3. `Id_Map(left_ids,right_ids)`

with the following behaviours:

1. Creates an empty object.
2. Constructs an Id_Map object from a struct that contains the same fields as the object.
3. Constructs an Id_Map object having the given leftids and rightids.

4.8 Logger

This class writes messages to files and/or to the screen. Use it in your scripts if you want output to go to both the terminal and a log file.

The class has ten methods:

add_fid Adds an open file (specified by its file id) to the list of files to log in. It takes two inputs: an fid and a logging level. The user must close the file sometime after removing its fid from the list.

add_filename Creats a file (with the given name) and adds it to the list of files to log in. It takes two inputs: a filename and a logging level. The logger will close the file when the file is removed from the list.

add_stderr Adds *stderr* to the logging list. The method takes one input: the logging level.

add_stdout Adds *stdout* to the logging list. The method takes one input: the logging level.

clear Removes all fids and filenames from the list. Files that were added using their name are closed.

log Writes a message to all files in the list (possibly including stdout and stderr). The method takes two or more inputs: the logging level, a string containing the message (possibly containing C conversion specifications), and further arguments corresponding to the number of conversion specifications in the string.

rm_fid Removes an fid from the logging list. The file is not closed by this method.

rm_filename Removes a filename from the logging list. The file is closed by this method.

rm_stderr Removes *stderr* from the logging list.

rm_stdout Removes *stdout* from the logging list.

validate Checks a Logger object against a list of class invariants.

The constructor for this class takes no arguments. See the comments at the top of the class definition file for a description of how to use a logger in your script.

4.8.1 logging levels

Logging levels are used to control which messages are written to which files. Each file in the logging list has a logging level and every call to the *log* method includes a logging level. If the *bitand* of the message logging level with a file's logging level is not zero, the message will be written to that file. The user can set the logging level to any non-negative integer, but for convenience, the Logger class provides the following named constants: None, Major_Error, Error, Warning, Minor_Warning, Info, Minor_Info, Debug_1, Debug_2, All. These constants can be combined using *bitor* so that all combined conditions are true e.g. a file with a logging level of bitor(Warning,Error) will accept all messages with a logging level of Warning or Error or both (or All).

4.9 DET_Plot

This class is used for making detection error trade-off plots (see section 2.6 of the User Guide). It has the following dynamic methods:

display_legend Displays the legend for the plot. This function must be called only after all curves have been plotted.

plot_DR30_both Calls both plot_DR30_fa and plot_DR30_miss.

plot_DR30_fa Draws a vertical line indicating the point beyond which the estimate of the false alarm rate is unreliable.

plot_DR30_miss Draws a horizontal line indicating the point beyond which the estimate of the miss rate is unreliable.

plot_mindcf_point Places a min-dcf point on the DET curve.

plot_rocch_det An function to plot a DET curve using the ROCCH curve.

plot_steppy_det Function to plot a DET curve from the ROC.

save_as_pdf Saves the plot to a PDF file.

set_system Inputs scores to the class to be used by subsequent calls to plotting methods of this class. This function must be called before curves are plotted for a system, but it can be called several times with different systems (with calls to plotting methods in between) so that curves for different systems appear on the same plot. It takes three inputs: a vector of target scores for the system, a vector of non-target scores, and an optional system name which will be prepended to the curve names for the system in the legend.

set_system_from_scores This function has the same purpose as set_system, but instead of taking vectors of target and non-target scores as input, it takes a Scores object and a Key object and produces the target and non-target vectors itself which it then passes to set_system.

It has two static methods which are used to create a *Plot_Window* object to pass to the constructor:

make_plot_type_from_string The class has four predefined window types which we have found useful in our work. This function takes the name of one of these windows as input and returns an object describing that window. The four window names are: old, new, big, sre10. See the class definition file for the window settings.

make_plot_type_from_values This function takes the values needed for constructing the window directly. The user is therefore not limited to using one of the four predefined windows.

and a static method for testing the class:

test_this Makes a plot comparing *plot_rocch_det* with *plot_steppy_det*.

The following constructors are available:

1. `Det_Plot(plot_window)`
2. `Det_Plot(plot_window,plot_title)`

with the following behaviours:

1. Constructs a Det_Plot object with the axes described by a Plot_Window object and with an empty plot title.
2. Constructs a Det_Plot object with the axes described by a Plot_Window object and with the given plot title.

4.10 Norm_DCF_Plot

This class is used for making normalized Bayes error rate plots (see section 2.5.4 of the User Guide).

It has 15 public methods:

display_legend Displays the legend for the plot. This function must be called only after all curves have been plotted.

plot_curves A convenience method that calls a selection of the plotting methods depending on the values set in an input mask.

plot_dcf_curve_act Plots the Bayes error-rate curve for the current system (set using set_system).

plot_dcf_curve_min Plots the minimum Bayes error-rate curve for the current system.

plot_DR30_both Calls both plot_DR30_fa and plot_DR30_miss.

plot_DR30_fa Plots the Doddington 30 point for the false alarm rate for the minimum Bayes error-rate curve for the current system.

plot_DR30_miss Plots the Doddington 30 point for the miss rate for the minimum Bayes error-rate curve for the current system.

plot_fa_rate_act Plots the false alarm contribution to the actual Bayes error-rate curve for the current system.

plot_fa_rate_min Plots the false alarm contribution to the minimum Bayes error-rate curve for the current system.

plot_miss_rate_act Plots the misses contribution to the actual Bayes error-rate curve for the current system.

plot_miss_rate_min Plots the misses contribution to the minimum Bayes error-rate curve for the current system.

plot_operating_point Plots a vertical line on the plot. This is useful for indicating threshold values (the negative of the operating point) for calibrated scores. This function is independent of *set_system*.

save_as_pdf Saves the plot to a PDF file.

set_system Inputs scores to the class to be used by subsequent calls to plotting methods of this class. This function must be called before curves are plotted for a system, but it can be called several times with different systems (with calls to plotting methods in between) so that curves for different systems appear on the same plot. It takes three inputs: a vector of target scores for the system, a vector of non-target scores, and an optional system name which will be prepended to the curve names for the system in the legend.

set_system_from_scores This function has the same purpose as set_system, but instead of taking vectors of target and non-target scores as input, it takes a Scores object and a Key object and produces the target and non-target vectors itself which it then passes to set_system.

The following constructors are available:

1. `Norm_DCF_Plot()`
2. `Norm_DCF_Plot(plot_axes)`
3. `Norm_DCF_Plot(plot_axes,plot_title)`

with the following behaviours:

1. Constructs a Norm_DCF_Plot object with the axes set to `[-10,0,0,1.2]` and with an empty plot title.

2. Constructs a Norm_DCF_Plot object with the axes described by a vector of four numbers and with an empty plot title.

3. Constructs a Norm_DCF_Plot object with the axes described by a vector of four numbers and with the given plot title.

The elements in the vector for the axis limits are in the standard Matlab order: $[x_{min}, x_{max}, y_{min}, y_{max}]$.

5 File Formats

The classes Key, Ndx, Scores, Seg_Quality, Id_Map and Quality all have both Matlab mat and HDF5 file formats. All of those classes except Quality also have a text format.

To save an object, call the *save* method, which determines the file format from the extension in the filename given to the method. Otherwise, the classes also have three methods that are hardcoded to save the object in a particular format: *save_hdf5*, *save_mat* and *save_txt*.

To load an object, use the static *read* method which determines the format from the file extension, or use *read_hdf5*, *read_txt* or *read_mat*. You must use the *read* methods like this:

```
scr = Scores.read(filename);
```

In the following two subsections, the text and hdf5 file formats are described to allow interoperability between the toolkit and other programs (or future versions of the BOSARIS toolkit) written in other programming languages.

If other Matlab tools need to create mat files, then they should either

1. create objects as described above and then use the Matlab *struct* function to convert them to structures (to see how to do this, look at any of the *save_mat* methods), and then save them, or

2. create the structures that correspond to the objects described above directly i.e. a structure that has the same fields as the object and save the structure to a mat file.

5.1 Text format

This subsection describes the text file format for each class that implements it.

5.1.1 Ndx

A text file for an index must contain two columns:

1. a list of model names

2. a list of test segment names

Each line in the file describes a trial. Therefore, model and segment names may be repeated in the columns, but lines must be unique.

5.1.2 Key

The text format for a key is similar to that for an index except it contains a third column whose entries are all one of the words *target* or *nontarget*, to indicate whether the pair form a target or non-target trial. Again, the file should not contain duplicate trials (described by the entries in the first two columns).

5.1.3 Scores

The text format for a score file also contains three columns with the first two columns having the same meaning as for keys and indexes. The third column contains a score (a real number).

5.1.4 Seg_Quality

The initial line in the text format for a Seg_Quality object must contain a single positive integer which indicates the number of quality values per segment. Each of the remaining lines in the file is for a single segment. The first column contains the segment name and the remaining columns (the number of these must be equal to the integer on the first line) contain the quality values (as real numbers) for that segment. The order of these entries in the columns is important. All entries in a column must be for the same *quality measure*.

5.1.5 Id_Map

A text file for an Id_Map contains two columns. The left column contains the left ids and the right column contains the right ids. Each row becomes an entry in the map.

5.2 HDF5 format

This subsection describes the HDF5 file format for each class that implements it.

Note: All matrices should be in row-major (*C-style*) order.

5.2.1 Key

Files for storing Key information should have a top level dataset called *trial_mask* and a group called *ID* containing two datasets: *row_ids* and *column_ids*.

trial_mask should be a matrix with *uint8* entries of -1, 0 and 1. On loading, logical target and non-target masks are created. 1s indicate target trials, -1s indicate non-target trials and 0s indicate trials that should not be performed.

row_ids and *column_ids* should be vectors of *h5string*. *row_ids* gives the model names and *column_ids* gives the test segment names.

5.2.2 Ndx

Files for storing Ndx information should have a top level dataset called *trial_mask* and a group called *ID* containing two datasets: *row_ids* and *column_ids*.

trial_mask should be a matrix with *uint8* entries of zero and one (zero for *false* and one for *true*). On loading, a logical matrix is created.

row_ids and *column_ids* should be vectors of *h5string*. *row_ids* gives the model names and *column_ids* gives the test segment names.

5.2.3 Scores

Files for storing Scores information should have two top level datasets called *scores* and *score_mask* and a group called *ID* containing two datasets: *row_ids* and *column_ids*.

scores should be a matrix of scores of type *H5T_IEEE_F64LE*.

score_mask should be a matrix with *uint8* entries of zero and one (zero for *false* and one for *true*). On loading, a logical matrix is created.

row_ids and *column_ids* should be vectors of *h5string*. *row_ids* gives the model names and *column_ids* gives the test segment names.

scores and *score_mask* must be the same size and the number of rows must be the same as the number of *row_ids* and the number of columns must be the same as the number of *column_ids*.

5.2.4 Seg_Quality

Files for storing Seg_Quality information should have two datasets called *ids* and *values*.

values should be a matrix of quality measures (for models and/or segments) of type *H5T_IEEE_F64LE*. Entries in a column are all for the same model or segment, and entries in a row are all of the same type of quality measure.

ids should be a vector of *h5string*. It gives the model or segment names.

5.2.5 Quality

Files for storing Quality information should have two top level datasets called *modelQ* and *segQ* and a group called *ID* containing either two or four datasets: *row_ids* and *column_ids* are compulsory and *has_row* and *has_column* are optional, but must either both be present or both be absent.

modelQ should be a matrix of quality measures for models (or enrolment segments) of type *H5T_IEEE_F64LE*. Entries in a column are all for the same model, and entries in a row are all of the same type of quality measure.

segQ should be a matrix of quality measures for test segments of type *H5T_IEEE_F64LE*. Entries in a column are all for the same test segment, and entries in a row are all of the same type of quality measure.

row_ids and *column_ids* should be vectors of *h5string*. *row_ids* gives the model names and *column_ids* gives the test segment names.

has_row and *has_column* should be vectors with *uint8* entries of zero and one (zero for *false* and one for *true*). On loading, logical vectors are created. *has_row* must be the same length as *row_ids* and *has_column* must be the same length as *column_ids*.

5.2.6 Id_Map

Files for storing Id_Map information should have two datasets called *left_ids* and *right_ids*. Both should be a vectors of *h5string*.

6 Calibration

Now that we have described how data is stored by the toolkit, we begin describing what the toolkit can do with the data. The uses of the toolkit can be divided into four categories: calibrating scores, fusing scores, making plots to evaluate systems, and calculating statistics to measure the accuracy and calibration of systems. This section describes the functions that can be used for calibration—the other uses are explained in the next three sections, respectively.

The toolkit provides six main functions for doing calibration:

linear_calibrate_scores_dev_eval Calibrates scores for two sets of trials, one of which has a key and the other an index. The calibration is trained on the former and the same calibration is applied to both sets of scores. This function does linear calibration (i.e. scores are scaled and shifted).

linear_calibrate_scores_dev_eval_from_files A wrapper function for the function *linear_calibrate_scores_dev_eval* that does the loading and saving of files.

linear_calibrate_scores Calibrates a single set of scores, given the key. This function does linear calibration (i.e. scores are scaled and shifted). The calibration weights are trained on the supplied scores and then applied to the same scores.

linear_calibrate_scores_from_files A wrapper function for the function *linear_calibrate_scores* that does the loading and saving of files.

pav_calibrate_scores_dev_eval Calibrates two sets of scores, one of which has a key and the other an index. The calibration is trained on the former and the same calibration is applied to both sets of scores. This function uses the PAV algorithm for calibration.

pav_calibrate_scores Calibrates a single set of scores, given the key. The calibration is trained on the supplied scores and then applied to the same scores. This function uses the PAV algorithm for calibration.

and two auxiliary functions (which are used by the above main functions): **pav_calibration** and **train_linear_calibration**. The auxiliary functions take a vector of target scores and a vector of non-target scores as input instead of a score object and a key object (which the main calibration functions do).

For linear calibration, the weights are trained using logistic regression.

7 Fusion

The second use of the toolkit is for fusing scores from different systems. The user can either give the scores to a function that returns fused scores or train a fusion function that fuses input scores (using weights trained in the training step).

The toolkit also enables quality measures to be fused with scores. When fusing with quality measures, the toolkit always first trains fusion weights without including the quality measures and then uses these weights as the initial values when training the weights for the quality fusion.

The toolkit provides eight principal functions for doing fusion:

linear_fusion_dev_eval This function does a linear fusion of scores from several systems. The fusion is trained on the *dev* scores (from all systems) and then applied to both the *dev* and *eval* score sets to produce a single score per *dev* trial and a single score per *eval* trial.

linear_fusion_dev_eval_from_files A wrapper function for the function *linear_fusion_dev_eval* that does the loading and saving of files.

linear_fusion_scores This function does a linear fusion of scores from several systems. The fusion is trained on the scores (from all systems) and then applied to the same score sets to produce a single score per trial.

linear_fusion_scores_from_files A wrapper function for *linear_fusion_scores* that does the loading and saving of files.

linear_and_quality_fusion_dev_eval This function trains a linear fusion of scores from several systems, and then trains a second fusion that includes quality measures. The fusions are trained on the *dev* scores (from all systems) and then applied to both the *dev* and *eval* score sets to produce a single score per *dev* trial and a single score per *eval* trial.

linear_and_quality_fusion_dev_eval_from_files A wrapper function for *linear_and_quality_fusion_dev_eval* that does the loading and saving of files.

linear_and_quality_fusion_scores This function trains a linear fusion of scores from several systems, and then trains a second fusion that includes quality measures. The fusions are trained on the scores (from all systems) and then applied to the same score sets to produce a single score per trial.

linear_and_quality_fusion_scores_from_files A wrapper function for the function *linear_and_quality_fusion_scores* that does the loading and saving of files.

and eight auxiliary functions:

analyse_dev_lin_fusion This function displays the results of a linear fusion on supervised scores. The function displays measures of goodness for each system in the fusion and for the fused scores. See the second last paragraph of this section for a list of the statistics calculated.

analyse_dev_lin_and_qual_fusion This function displays the results of linear and quality fusions on supervised scores. The function displays measures of goodness for each system in the fusions and for the fused scores. See the second last paragraph of this section for a list of the statistics calculated.

analyse_eval_lin_fusion This function compares the results of a linear fusion on unsupervised scores. See the final paragraph of this section for a list of the statistics calculated.

analyse_eval_lin_and_qual_fusion This function compares the results of linear and quality fusions on unsupervised scores. See the final paragraph of this section for a list of the statistics calculated.

apply_linear_fusion_function This function takes a score set and a linear fusion function as input and produces a fused score set as output.

apply_quality_fusion_function This function takes a score set, quality measures and a quality fusion function as input and produces a fused score set as output.

train_linear_fusion_function This function trains a linear fusion function, which can then be applied to score sets to fuse them.

train_quality_fusion_function This function trains a quality fusion function, which can then be applied to score sets to fuse them.

train_linear_fuser This function is called by the *train_linear_fusion_function*. The difference between the two is that the input to train_linear_fusion_function is a key object and an array of score objects, whereas the input to this function is a matrix of scores (with systems in the rows and trials in the columns) and a label vector (containing ones at target trial positions, minus ones at non-target trials positions and zeros at positions of trials that should be ignored).

The statistics calculated for each dev system by the analyse functions are an estimate of the proportion of target trials (to all trials) in the dev set (this can be compared with the true target proportion, which is displayed), the actual DCF value, minimum DCF value (both at the operating point) and PRBEP.

The statistics calculated for each eval system by the analyse functions are an estimate of the proportion of target trials (to all trials) in the eval set, the KL-divergence between the dev score sets for all systems (for comparison with the eval divergences), and the KL-divergence between the eval score sets for all systems.

8 Plotting

The toolkit can make two types of plots: DET plots and normalized Bayes error-rate plots. The ways to make these plots are described in the following two subsections.

8.1 DET plots

DET plots are made using the class *Det_Plot* (see section 4.9). The *demo_main* script (in the *demo* directory) generates two DET plots using this class.

There is also an example script, *detplot*, in the *plotting* directory which uses the *Det_Plot* class to plot a single DET curve and does the loading of the key and scores and the saving of the plot.

8.2 normalized Bayes error-rate plots

These plots are made using the class *Norm_DCF_Plot* (see section 4.10). The *demo_main* script plots one of these plots which contains several curves and shows how various transformations on a set of scores affect calibration.

There is also an example script, *dcfplot*, in the *plotting* directory which loads development and evaluation scores and plots several curves for both sets of scores on the same plot and saves the plot.

9 Stats functions

The final use of the toolkit is to run evaluation functions on scores to calculate statistics for comparing systems or judging the calibration of a system.

The following nine functions are available for evaluating systems:

area_under_rocch Calculates the area under the ROC convex hull given the rocch co-ordinates.

cllr Calculates C_{llr} (see section 2.5.3 of the User Guide), from target and non-target scores.

dcf_at_threshold Calculates the actual DCF values for a score set at a number of thresholds (specified by the user).

eer Calculates the *equal error rate*.

effective_prior Calculates the effective prior (see section 2.4.2 of the User Guide) from the triple (target_prior, cost_of_miss, cost_of_false_alarm).

fast_actDCF Computes the actual dcf value (or a vector of actual dcf values) given target and non-target scores and the logit of the effective prior (or a vector of logit prior values). The input scores need to be calibrated.

fastEval a wrapper function for fast_actDCF and fast_minDCF that returns the actual DCF value, the minimum DCF value, PRBEP and equal error rate.

fast_minDCF Computes the minimum dcf value (or a vector of minimum dcf values) given target and non-target scores and the logit of the effective target prior (or a vector of logit prior values). The input scores do not need to be calibrated.

min_cllr calculates minimum C_{llr} which is the value of C_{llr} that one would get if the scores were perfectly calibrated.

prbep Calculates the *precision-recall break even point*.

10 Utility functions

This section lists functions that do not implement primary functions of the toolkit, but rather are used by other functions or are useful in scripts.

The functions are divided into the following subdirectories:

data Functions for reading from files and doing formatted printing.

det Helper functions for the DET plotting code.

manip Object manipulation functions for constructing objects from objects of other types, etc.

maths Some mathematical functions.

Optimization_Toolkit This directory contains a toolkit that performs optimization. The BOSARIS toolkit uses this toolkit to train logistic regression weights for calibration and fusion. The Optimization Toolkit is not documented in this manual.

10.1 data functions

load_qual_files Loads quality measures from files into an array of Quality objects.

load_score_files Loads scores from files into an array of Scores objects.

logprint Writes a message to all log files (and possibily stdout and stderr) if the logger exists; otherwise does nothing.

log_error Calls *logprint* with the logging level set to *Error*.

log_info Calls *logprint* with the logging level set to *Info*.

log_warning Calls *logprint* with the logging level set to *Warning*.

read_list Reads a list (single column of strings) from a file and outputs a cell array of strings.

read_map Reads a map (two columns of strings, entries in first column unique) from a file and outputs a struct with two cell arrays (of the same length) of strings—one for the keys and one for the values.

save_list Saves a cell array of strings to a text file with one string per line.

sprintfmatrix Takes a matrix as input and returns a string containing the formatted matrix with row labels and newlines added.

10.2 DET functions

compute_roc Computes the points on the ROC (see section 2.6 of the User Guide) given vectors of target and non-target scores.

filter_roc Removes redundant points from the ROC so that plotting an ROC "curve" will be faster. The output ROC curve will be identical to the one plotted from the ROC.

pavx This is the implementation of the PAV algorithm (see section 3.2 of the User Guide) in the toolkit.

rocchdet Given vectors of target and non-target scores, returns x and y vectors that can be passed to Matlab's plot function to plot a DET curve.

rocch Computes the points on the ROCCH (see section 2.6.1 of the User Guide) given vectors of target and non-target scores.

10.3 object manipulation functions

make_objective_function_from_name Takes the name of a type of objective function (as a string) and produces an objective function which can be passed to the training code.

maplookup Produces a list of values given a list of keys and a map from keys to values.

map_mod_names The purpose of this function is to map the modelset of a Key or Ndx. The model name is an indirect reference to the enrolment segment. An index could use one set of model names to refer to enrolment segments while a score file uses another set of names to refer to the same segments. This function facilitates the conversion of the model names in the Ndx to the set used by the score file.

seg_qual_to_qual Merges two objects of type Seg_Quality (one for the models and one for the test segments) to form an object of type Quality.

10.4 maths functions

logit Maps probability to log odds.

neglogsigmoid The negative of the natural logarithm of the logistic function of the input.

probit Computes the probit function i.e. the inverse of the cumulative distribution function for the standard normal distribution.

sigmoid Applies the logistic function to the input.

11 Demo Scripts

There are two demos in the *demo* directory of the toolkit. They both use synthetic data. One, *demo_main*, works with all the data in memory, whereas the other, *demo_with_files*, creates mat files on disk to demonstrate how the toolkit works with files.

11.1 demo_main

This script demonstrates four features of the toolkit:

1. the normalized Bayes error-rate plot
2. the DET plot
3. fusion
4. evaluation functions

11.1.1 normalized Bayes error-rate plot

The script calls the sub-level script *demo_plot_nber* to produce a plot that shows the effect of various score transformations on calibration. The figure produced by this script is included in the User Guide in section 2.5.4. See that section for a description of the NBER plot.

11.1.2 DET plot

The sub-level script *demo_plot_det* creates a DET plot containing curves for two systems. For each curve, the plot displays the minDCF point, the DR30 point for misses and the DR30 point for false alarms.

11.1.3 fusion

The demo_main script creates train and test scores for two systems and then calls the sub-level script *demo_fusion* which trains a fusion function (using *train_linear_fuser*) on the train scores and applies it to the test scores. demo_main then displays a second DET plot that has curves for the test scores for the two systems and the fused test scores.

11.1.4 evaluation functions

The final few lines of demo_main calculate and display statistics for the two sets of test scores that were fused and for the test scores resulting from the fusion. The statistics calculated are equal error rate, minimum DCF and actual DCF. Note that the two systems are not calibrated before fusion, so their minimum and actual DCF values are quite far apart.

11.2 demo_with_files

This script repeats the last two stages of demo_main, but works from files instead of scores stored in memory. It creates files containing train and test scores for two systems and train and test keys and indexes. Next it calls *linear_fusion_dev_eval_from_files* to train a fusion function and apply that function to the test scores. It then shows the results by creating a DET plot and displaying the equal error rates, minimum DCF values and actual DCF values for the test scores of the two systems and the fused test scores.

12 Code Examples

This section is meant to give example code listings to show how toolkit functions can be used to perform tasks that may be required by general users of the toolkit. It currently contains one example which illustrates how to convert from Seg_Quality data to Quality data so that the quality measures can be used by the fusion code.

12.1 converting from Seg_Quality to Quality

Assume we have a single score file that we want to fuse with quality measures. We have the quality measures for all segments in the database stored in a file

containing a Seg_Quality object. The names of the test segments in the Scores and Key are in the same form as those in the Seg_Quality object, but the model names are different. We have an Id_Map between the model names (leftids) and the segment names (rightids).

The function *linear_and_quality_fusion_scores* can be used to do the fusion, but it takes an array of Quality objects as one of its inputs. So we need to create a Quality object from our Seg_Quality object. This can be done with the function *seg_qual_to_qual* which takes two Seg_Quality objects as input: one for the models and one for the test segments.

We are only interested in trials for which we have scores (because we want to fuse with them), so the models and test segments we need are those from the Scores object. Then we need to get the quality measures corresponding to these lists and create new Seg_Quality objects which we then pass to *seg_qual_to_qual*. The code snippet below performs these steps.

```
% read in the information we have
key = Key.read(keyfilename);
scr = Scores.read(scorefilename);
all_seg_qual = Seg_Quality.read(qualfilename);
idmap = Id_Map.read(trnfilename);

% Select the quality measures for the test segments.
test_seg_qual = all_seg_qual.align_with_ids(scr.segset);

% The following three steps make a Seg_Quality object for
% the models.

% Select the quality measures for the model segments.
model_seg_qual = all_seg_qual.align_with_ids(idmap.rightids);

% Replace model segment names with model names.
model_seg_qual.ids = idmap.leftids;

% Select only the models for which we have scores.
model_seg_qual = model_seg_qual.align_with_ids(scr.modelset);

% Now make the Quality object.
qual = seg_qual_to_qual(model_seg_qual,test_seg_qual);

% Pack them into arrays for the fusion function.
scores_obj_array = Scores.empty(1,0);
scores_obj_array(1) = scr;
qual_obj_array = Quality.empty(1,0);
qual_obj_array(1) = qual;

% filter the key so that it contains only trials for which
% we have scores.
key = key.filter(scr.modelset,scr.segset,true);

% call the fusion function.
```

```
Qtrans = [1];
obj_func = [];
[scr_lin,scr_qual] = linear_and_quality_fusion_scores(key,...
                        scores_obj_array,qual_obj_array,...
                        Qtrans,prior,niters,obj_func,...
                        true,true);

% save the resulting scores
scr_qual.save(outfilename);
```