THE ALLIES EVALUATION
HOW TO RUN YOUR SYSTEM ON THE BEAT PLATFORM

A PREPRINT

Anthony Larcher
LIUM - Le Mans Université
Avenue Olivier Messiaen
F-72085 – LE MANS CEDEX 9
anthony.larcher@univ-lemans.fr

Olivier Galibert
LNE
Olivier.Galibert@lne.fr

Andre Anjos, Samuel Gaist
IDIAP
andre.anjos@idiap.ch, samuel.gaist@idiap.ch

March 9, 2020

ABSTRACT

This document describes how to run the lifelong diarization task for the ALLIES evaluation. For this task, sources have to be uploaded to the BEAT platform available at IDIAP. This guide, dedicated to ALLIES evaluation's participants describes how to set up a local BEAT platform in order to develop your own speaker diarization system and how to push it to the online platform where the evaluation is to be run.

This documentation and all usefull files are available on this ALLIES evaluation webpage.

1 How to setup a local platform for system development

1.1 Before installation

1. Download, the two documents available on the ALLIEs evaluation webpage
   - the Rules for Supervisor available here
   - the licence to access the development data that is available here

   You have then to complete those documents and send them to the organizers[1].
2. You will then receive credentials to download the training and development data from this link[2]
3. Install a local BEAT platform on your facilities. This environment is required to:
   - download the baseline system and run it on the development data;
   - develop your own system and integrate it in the BEAT framework;
   - push your algorithm on the online BEAT platform in order to submit your system for evaluation.

1.2 Install a local BEAT platform

To develop your system, we recommend to set up a local installation of the BEAT platform. This installation requires three steps:

1. Install CONDA on your computer.

[1] lifelong-speaker-evaluation@univ-lemans.fr
[2] https://allies.kervella.org

2. Setup a conda environment dedicated to the BEAT platform by following the following instructions
3. Pull the baseline system and run a lifelong diarization evaluation

1.2.1 Install CONDA

CONDA's documentation can be found at urlhttps://docs.conda.io/en/latest/.

1.2.2 Setup a CONDA environment for BEAT

Download the file `beat.yml` from here. Run the following command from a terminal:

```
conda env create -f beat.yml
```

Note that the environment created in this process includes the same exact python modules that are available on the on-line platform. In case other modules are required to run your system please contact the organizers.

Activate the CONDA environment (named "beat")

```
$ conda activate beat
(beat) $
```

All further instructions assume you have the environment where BEAT packages are installed properly activated.

If your machine has a graphical interface, you can check the installation of the BEAT platform is working by launching:

```
beat editor start
```

This will start the graphical editor. At this stage, you should not visualize any element in the EDITOR.

1.2.3 Pull the baseline system and run

The BEAT editor allows you to implement and test your solution locally on either a reduced or different dataset. Once you have everything working as expected, push the resulting code to the platform and execute it there, for validation.

The following steps describe the procedure to start from an existing experiment and build on that.

```
beat exp pull anthony_larcher/anthony_larcher/diarization_ll_dev/1/diarization_ll_dev
```

This will pull the experiment as well as all its dependencies from the platform so that it can be edited (and run) locally.

You now have a BEAT `prefix`

Now, you need to specify the path to your development data to the BEAT experience. To do this, open the file

`databases/allies-development-data-sd/1.json`

and change the line:

```
"root_folder": "/lium/corpus/audio/bn/fr/ALLIES/dev"
```

to set your path to the ALLIES development data as follow

```
"root_folder": "mypath_to_allies_data/
```

To run the ALLIES speaker diartization enter the following command:

```
beat exp run anthony_larcher/anthony_larcher/diarization_ll_dev/1/diarization_ll_dev &> output.log
```

The `allies-baseline_output.log` should be similar to the one available here. Given the quantity of data to process, the experiment might run for 10 hours depending on the number of CPU available on your facilities.

2 Implement your own system in the ALLIEs environment

You now have a running baseline on the ALLIEs development data. The following steps will show you:

1. how to fork the baseline experiment in order to create your own;
2. how to edit the code to implement your system

Note that values within angle brackets (< and >) will depend on the asset name. Replace these values by appropriate actual values.

2.0.1 Create your own experiment and algorithms

Run the following command to create a fork of the baseline experiment.

```
(beat) $  beat experiment fork anthony_larcher/anthony_larcher/diarization_ll_dev/1/diarization_ll_dev
 <your-beat-user-id>/<experiments>/<toolchain-author>/<toolchain-name>/<toolchain-version>/<experiment-name>
```

for instance:
```
beat --prefix ./ experiment fork anthony_larcher/anthony_larcher/diarization_ll_dev/1/diarization_ll_dev
 anthony_larcher/anthony_larcher/diarization_ll_dev/1/allies_test_fork
```

creates a new exeperiment `anthony_larcher/anthony_larcher/diarization_ll_dev/1/allies_test_fork`

Fork the algorithm you want to modify amongst the following:

- `anthony_larcher/features_extractor/1`
- `anthony_larcher/diar_train/1`
- `anthony_larcher/diar_lifelong/1`

Here is an example that shows how to fork the main block of lifelong learning.

```
(beat) $ beat algorithm fork anthony_larcher/diar_train/1 <your-beat-user-id>/
<algorithm>/<version>
```

For instance: `beat --prefix ./ algorithms fork anthony_larcher/diar_lifelong/1 anthony_larcher/diar_lifelong/2`

This will allow you to use your own modified algorithm for your fork.

Note that you will have to update your experiment fork in order to use your own algorithm(s). This can be done through the BEAT editor or by editing the JSON file of your own experiment.

Note that you do not need to fork an algorithm, you can use a different one as long as its inputs and outputs match the one you want to replace.

2.0.2 Edit the code:

```
(beat) $ beat algorithm edit <author>/<algorithm>/<version>
```

This will open your default configured editor to edit the code of the algorithm passed in parameter. If not editor is set, use the following command to set a default editor:

```
(beat) $beat conf set editor vim
```

Note that you can visualize the BEAT configuration by entering:

```
(beat) $beat conf show
```

To edit any other asset, just start the editor, optionally pointing to the prefix of interest

```
(beat) $ beat --prefix <path/to/prefix> editor start
```

This will start the BEAT editor so that you will have an easy access to all locally available assets. You'll be able to edit them visually. The editor also allows to start your favorite editor for code or documentation modification.

2.0.3 Run an experiment locally

You can run your own exeperiment locally by entering:

```
$ beat exp run <experiment-author>/experiments/<toolchain-author>/<toolchain-name>/
<toolchain-version>/<experiment-name>
```

2.0.4 Submit your experiment on the on-line platform

Once the experiment runs successfully on your machine, you can upload it to the online platform and run it there.

```
$ beat exp push <experiment-author>/experiments/<toolchain-author>/<toolchain-name>/
<toolchain-version>/<experiment-name>
$ beat exp start <experiment-author>/experiments/<toolchain-author>/<toolchain-name>/
<toolchain-version>/<experiment-name>
```

You can monitor the execution of your experiment at any moment with:

```
$ beat exp monitor <experiment-author>/experiments/<toolchain-author>/<toolchain-name>/
<toolchain-version>/<experiment-name>
```

We recommend to run your expriment on the on-line platform in order to make sure everything runs smoothly.

3 Overview of the system and environment

The toolchain developed to evaluate the autonomous systems is described in Figure 1 this toolchain can be described in four parts:

- the input datasets (red on Figure 1);
- the three blocks of the system (green on Figure 1 to be modified to include your own system);
- the user simulation (white on Figure 1);
- the evaluation blocks (blue on Figure 1)

Note that input datasets, user simulation and evaluation blocks are fixed and guaranty reproducibility of the experiments. Participants are free to edit the three blocks of the system in order to include their own code. Once your code is included in this toolchain, the system will run automatically and the BEAT platform is responsible for managing the data exchanges between the different blocks of the architecture.

Thus, you don't need to take care about the communication between blocks, especially, the interaction between the system and the user simulator is automatic.

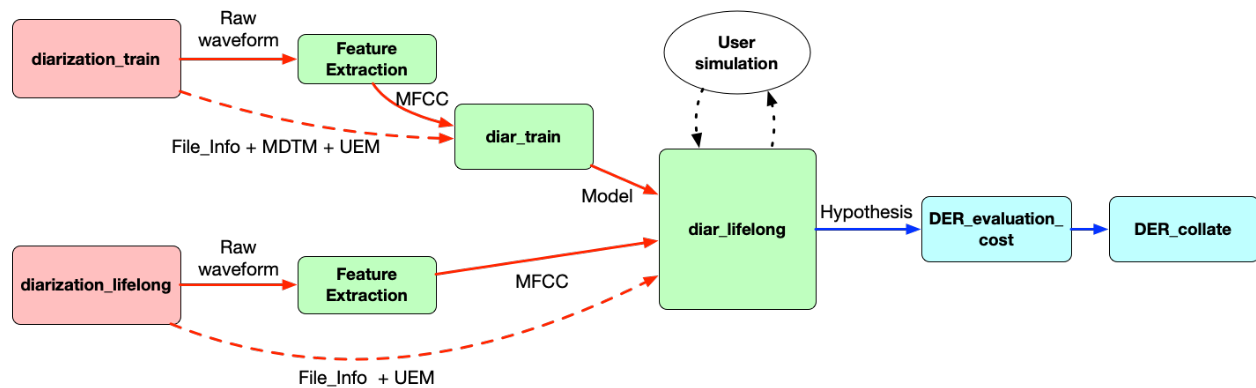

Figure 1: Architecture of the ALLIES toolchain for autonomous lifelong learning system evaluation.

3.1 Input datasets

Two datasets are available:

diarization train this dataset includes audio files together with information about the TV shows and a manual segmentation available for supervised training. This dataset is available for the initial training of the system and later for system adaptation all along the system life-cycle. Audio files from this dataset can be access anytime on-demand.

diarization lifelong This dataset is available in a sequential manner and used for evaluation of the system. Each audio file is provided to the system without any other information. For each file, the system has to return an hypothesis that will be evaluated.

3.2 System blocks (the core part)

This section describes the part of the toolchain your system has to be included in. The architecture of the system has been developed according to standard speaker diarization architectures. In order to facilitate the development of your system and to provide a baseline, a complete implementation of a speaker diarization system using SIDEKIT [1] and S4D [2] is provided on the allies evaluation web page

The lifelong learning system is composed of three modules. Inputs and output formats of the modules are described in the following sections.

3.2.1 Feature extraction module

This module is referred to as *feature_extraction* and is implemented in the file `algorithms/allies/features_extractor/1.py`. As for every module, the main function is **process**. Prototype of this function MUST not be modified in order to guaranty your toolchain will execute properly.

All audio files will go through this block that is duplicated in the toolchain in order to guaranty that training files and test files are processed the exact same manner;

input MUST NOT be modified

output a *numpy.ndarray*

```
def process(self, inputs, dataloader, outputs):

    # Get the incoming wav form as a numpy.ndarray
    speech = inputs["speech"].data.value

    label, energy, cep, _ = self.fe.extract_from_signal(speech, 16000)
    # Concatenate energy and cep
    feat = numpy.hstack([energy[:, None], cep])
    # Add dynamic features
    feat = self.fs._delta_and_2delta(feat)
    # Smooth the labels
    label = sidekit.frontend.vad.label_fusion(label)

    outputs["features"].write({"value": feat})

    # always return True, it signals BEAT to continue processing
    return True
```

The code above shows how to retrieve incoming wave form from the *dataloader* object and how to return the output features for the next modules. Note that both inputs and outputs (variable **speech** and **feat**) in this code are *numpy.ndarray*.

3.2.2 Initial training module

The initial training of the system is implemented in the file `algorithms/allies/diar_train/1.py`. The *process* method is the main one. From this method you can access all data from the *diarization_train* set and use the human annotation provided with the data. This block outputs a model.

Inputs and outputs of this module.

inputs

- **features** the *numpy.ndarray* output from the Feature extraction module;
- **uem** an object including the content of a UEM file (parts of speech which must be processed and could be evaluated. Format of UEM is described in section 5.
- **file_info** a dictionary object that contains three fields: *file_id*, *supervision* and *time_stamp*. Those three fields are strings and give a unique file identification, the kind of supervision that can be performed on the file (only for diar_lifelong dataset) and the date of the first broadcast of the curent TV show respectively.
- **speakers** is composed of three aligned lists that describe all speech segments from the associated wav file. the lists are: *speaker* that contains the speaker ID of each segment, *start_time* and *stop_time* which are the time of beginning and end of each speech segment in seconds

output

- **model** your model as a string. An easy way to serialize your model as a string is to use the pickle module as done in the baseline system. This module allows you to include dictionaries, *numpy.ndarray* and neural networks easily (see the baseline system for an example of how to serialize your model).

3.2.3 Lifelong learning module

This block receives the initial model from the *diar_train* block and process all files from the *diarization_lifelong* dataset, one at a time. This module is the one managing the interaction between the human in the loop (user simulation) and your system.

Inputs and outputs of this module.

inputs

- **model** your model, output from the training module
- **features** the *numpy.ndarray* output from the Feature extraction module, this is the audio data to process
- **processor_uem** an object including the content of a UEM file (parts of speech which must be processed and will be evaluated (described in section 5).
- **processor_file_info** a dictionary object that contains three fields: *file_id*, *supervision* and *time_stamp*. Those three fields are strings and give a unique file identification, the kind of supervision that can be performed on the file and the date of the first broadcast of the curent TV show respectively.

output

- **adapted_speakers** is the final hypothesis returned by your system for the current TV show. It is composed of three aligned lists that describe all speech segments frim the associated wav file. the lists are: *speaker* that contains the speaker ID of each segment, *start_time* and *stop_time* which are the time of beginning and end of each speech segment in seconds

Additionally, this lifelong learning module can access all data from the `diarization_train` dataset. This data can be accessed via the following inputs:

Additional inputs

- **train_features** the *numpy.ndarray* output from the Feature extraction module (features from the training set).
- **train_uem** an object including the content of a UEM file (parts of speech which must be processed and could be evaluated;
- **train_file_info** a dictionary object that contains three fields: *file_id*, *supervision* and *time_stamp*. Those three fields are strings and give a unique file identification, the kind of supervision that can be performed on the file (only for diar_lifelong dataset) and the date of the first broadcast of the curent TV show.
- **train_speakers** is composed of three aligned lists that describe all speech segments frim the associated wav file. the lists are: *speaker* that contains the speaker ID of each segment, *start_time* and *stop_time* which are the time of beginning and end of each speech segment in seconds

Example on how to access lifelong learning data and training data is provided in the baseline system.

Interacting with the human in the loop.

Each file from the lifelong learning dataset comes with a flag stored in the `file_info` variable and named `supervision`, that specifies the mode of human assisted learning for this file. The mode can be:

- **active** the system is allowed to ask questions to the human in the loop;
- **interactive** once the system produces a first hypothesis, the human in the loop provides corrections to the system to improve the hypothesis;
- **none** Human assisted learning is OFF for this file. The system can still adapt the model in an unsupervised manner.

While processing an audio file, the system can perform unsupervised learning and goes through the the Human Assisted Learning process if supervision mode is either active or interactive.

The code below shows how to interact with the user simulation:

```
if supervision in ["active", "interactive"]:
    human_assisted_learning = True

if not human_assisted_learning:
    # Perform unsupervised learning
    self.model, self.global_diar = unsupervised_model_adaptation(features,
                                   file_id,
                                   data_loaders,
```

```
                                               self.model,
                                               self.global_diar,
                                               current_s4d_segmentation
                                               )

# If human assisted learning mode is on (active or interactive learning)
# loop until the user or the system stop the process
while human_assisted_learning:

    # Create a fake request that is used to initiate interactive learning
    # For the case of active learning, this request is overwritten by your system
    request = {"request_type": "toto", "time_1": 0.0, "time_2": 0.0}

    if supervision == "active":
        # The system can send a question to the human in the loop
        # by using an object of type request
        # The request is the question asked to the system
        request = generate_system_request_to_user(self.model,
                                                  self.global_diar,
                                                  current_s4d_segmentation)

    # package the request to be sent to the user simulation together with
    # the ID of the file of interest and the current hypothesis
    message_to_user = {
        "file_id": file_id,  # ID of the file the question is related to
        "hypothesis": current_hypothesis,  # The current hypothesis
        "system_request": request,  # the question for the human in the loop
    }
    # Send the request to the user simulation and receive the answer
    human_assisted_learning, user_answer = loop_channel.validate(message_to_user)

    # Take into account the user answer to generate a new hypothesis
    # and possibly update the model
    self.model, self.global_diar, current_s4d_segmentation = online_adaptation(self.model,
                       self.global_diar,
                       features,
                       current_s4d_segmentation,
                       request, user_answer)
```

4 From development phase to evaluation

Each participant team can choose to run three different systems for the final evaluation. On the 1st of June 2020, systems will be frozen and participants must indicate the name of the three BEAT experiments to run as their submission.

5 UEM format description

The UEM format is a format describing the time ranges in the source audio files the system should be working on. It is used to give the boundaries of the shows but also to exclude the zones with overlapping speakers. It's a space-separated columns format, with four columns:

1. File name without the extension
2. Channel number (always 1)
3. Start time of zone to diarize
4. End time of zone to diarize

Example extract:

```
TV8_LaPlaceDuVillage_2011-03-14_172834 1 407.621 471.166
TV8_LaPlaceDuVillage_2011-03-14_172834 1 471.666 478.396
TV8_LaPlaceDuVillage_2011-03-14_172834 1 476.920 493.571
TV8_LaPlaceDuVillage_2011-03-14_172834 1 492.927 495.556
```

6 MDTM format description

The MDTM format is a format describing the reference or an hypothesis for the speaker identity in a file. It's a space-separated format, with eight columns:

1. File name without the extension
2. Channel number (always 1)
3. Start time of the speaker range
4. Duration of the speaker range (beware, not end time)
5. Event type (always "speaker")
6. Event subtype (always "na")
7. Gender ("adult_male" or "adult_female", "unknown" for hypothese, not evaluated in any case)
8. Speaker id

In the references, the speaker id is the speaker name of the form "Firstname_LASTNAME", in the hypothesis it is a unique, space-less, identifier per speaker.

Example extract:

```
TV8_LaPlaceDuVillage_2011-03-14_172834 1 407.621 15.040 speaker na adult_male Michel_THABUIS
TV8_LaPlaceDuVillage_2011-03-14_172834 1 422.661 18.148 speaker na adult_male Philippe_DEPARIS
TV8_LaPlaceDuVillage_2011-03-14_172834 1 440.809 30.357 speaker na adult_male Michel_THABUIS
TV8\_LaPlaceDuVillage_2011-03-14_172834 1 471.666 6.730 speaker na adult_male Philippe_DEPARIS
```

References

[1] Anthony Larcher, Kong Aik Lee, and Sylvain Meignier. An extensible speaker identification sidekit in python. In 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 5095–5099. IEEE, 2016.

[2] Pierre-Alexandre Broux, Florent Desnous, Anthony Larcher, Simon Petitrenaud, Jean Carrive, and Sylvain Meignier. S4d: Speaker diarization toolkit in python. In Annual Conference of the International Speech Communication Association (Interspeech), 2018.