Speaker Diarization for Linguistics

Isaac L. Bleaman Ronald L. Sprouse

2023-03-17

Introduction

Linguists working in a variety of subfields (sociolinguistics, phonetics, language documentation, etc.) often require high-quality time-aligned transcripts of speech recordings. Transcription typically involves two separate processes: (1) identifying and marking off the segments of the recording that contain speech (vs. silence), and (2) transcribing each of the speech segments. When multiple people are speaking in the recording, as is the case for interviews, these two processes are carried out on a separate tier for each speaker.

Improvements in automatic speech recognition (ASR) technology have enabled researchers to segment and transcribe their recordings with commercial and open-source software tools. However, there are many cases in which ASR is either unavailable (e.g., for most minority languages) or not reliable enough for research applications. In these situations, researchers often carry out the entire workflow—both segmentation and transcription—by hand. Figuring out a way to automate even a small part of this workflow could save researchers a great deal of time, effort, and money.

This tutorial provides instructions on the use of open-source software for **speaker diarization**: the task of determining *who* is speaking *when* and marking off these segments with timestamps. The tutorial assumes that the user has a set of audio recordings with two speakers, e.g., from sociolinguistic interviews, and wishes to transcribe each speaker on a separate tier in a program such as ELAN or Praat. We piloted these steps as part of the transcription workflow for the Corpus of Spoken Yiddish in Europe, but your recordings can be in any spoken language. The number of speakers and other parameters can also be modified below.

The tutorial is written in Python and can be followed on a personal or cluster computer, assuming you have conda installed. The tutorial relies on the speaker diarization toolkit and pretrained pipeline distributed through pyannote (Bredin et al. 2020; Bredin & Laurent 2021). Diarization is computationally intensive, and so for best results we recommend using a GPU device. If you would like to test this process on your own speech recordings without needing to install any software locally, we have also put together an interactive Google Colab notebook that can be executed in the browser with a GPU runtime.

If you find this tutorial useful and incorporate these steps into your transcription workflow, we kindly ask that you cite it so that others may follow along. This material is based upon work supported by the National Science Foundation under Grant No. BCS-2142797. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Getting started

Clone GitHub repository

Clone our GitHub repository and navigate to it in the command line:

```
# at commandline
git clone https://github.com/ibleaman/Speaker-Diarization-for-Linguistics
cd Speaker-Diarization-for-Linguistics
```

Obtain access token

If you do not already have a free account with Hugging Face, register for one. Otherwise, make sure you are signed in.

In order to use the pretrained speaker diarization pipeline from pyannote, visit https://huggingface.co/pyannote/speaker-diarization and https://huggingface.co/pyannote/segmentation and accept the user conditions at both pages (if requested).

Next, visit https://huggingface.co/settings/tokens to create a User Access Token. Give it any name you want (e.g., "diarize") and READ access. Reveal your token and copy it into a text file saved somewhere on your machine as: `pyannote-auth-token.txt`

Keep this file **confidential**, ideally outside the directory where your code is located. You do not want to post this anywhere public, such as a GitHub repository.

Install dependencies

Create a conda environment using the included `environment.yml` file. This will ensure that you are using the same software and versions that we used to create this tutorial. Then open `index.ipynb` (this tutorial) to continue.

To do this, run the following from the `Speaker-Diarization-for-Linguistics` directory:

```
# at commandline
conda env create -f environment.yml --name diarize
conda activate diarize
jupyter notebook index.ipynb
```

Import libraries

Import the libraries that will be used during audio processing and speaker diarization.

```
import sys
from pathlib import Path
import diarize_utils as utils
from pyannote.audio import Pipeline
from phonlab.utils import dir2df
from audiolabel import df2tg
```

Prepare and diarize the audio files

In this section, we will prepare a set of audio files for diarization and then run a diarization pipeline on them to produce annotation files: either `.eaf` for use with ELAN or `.TextGrid` for use with Praat, according to your needs. The workflow produces speaker tiers in the annotation files that contain intervals marking locations in the audio where that speaker is talking.

As mentioned above, we initially designed this workflow to diarize interviews for the Corpus of Spoken Yiddish in Europe. The source audio for that project consists of stereo files, where each speaker was recorded on a separate lapel microphone and could thus be heard predominantly on either the left or right channel. However, there was also substantial "bleed" across the channels because the speakers were sitting near each other in the same room. The approach we found to work best was to diarize the left and right channels separately; this resulted in four "speaker" tiers, two for the interviewer (left and right) and two for the subject (left and right). We then calculated the average duration of speech on each tier to determine which two to label as the "interviewer" and which two to label as the "subject"; for each speaker, we further compared the mean intensity of the two channels in order to guess which tier (left or right) was likely to be more reliable. The resulting `.eaf` file thus contained four tiers, labeled "Interviewer probable," "Interviewer unlikely," "Subject probable," and "Subject unlikely." Our transcribers would then open the files in ELAN, verify that the "probable" tiers have reasonable segments (or else use the "unlikely" tiers), and begin transcription.

This is the workflow we adopt in this tutorial, though it can be modified to accommodate other kinds of speech recordings. Each step is designed to be easily repeated whenever new inputs to the step are added to the project. For step 1, this would be when new source audio files are added. Step 2 should be run whenever newly prepared audio files (i.e., from step 1) become available. Step 3 combines the diarized outputs (in the case of stereo files) into a single `.eaf` or `.TextGrid` file.

- Step 1: Prepare audio: Loop over every source audio file, extract the left/right channels (if stereo), and downsample the audio.
- Step 2: Diarize the prepared audio: Run the speaker diarization pipeline on each downsampled mono audio file.
- Step 3: Combine diarized outputs: *For stereo recordings only:* Combine the diarized outputs into a single `.eaf` or `.TextGrid` file.

Before we begin, set the variable `projroot` to be the location on your machine where you intend to store the input and output files in subdirectories:

```
projroot = Path('/path/to/my/project_directory') # update this
```

We also need to load in your Hugging Face access token, saved in an earlier step:

```
tokenfile = '/path/to/pyannote-auth-token.txt' # update this to the location of your secre

with open(tokenfile, 'r') as tf:
    auth_token = tf.readline().strip()
```

For simplicity, we will assume that you have source audio files (`.wav`) that you wish to diarize, and you have placed them in the `audio/source` directory inside the project root. If you wish, this directory can be organized as a flat folder of `.wav` files, or with further subdirectories corresponding to each interview, date, experimental group, etc.

The `dir2df` function produces a dataframe of the files found in the source audio directory. The speaker subdirectories are shown by the value of the `relpath` column, and the filenames are stored in `fname`. The `barename` column contains the filename without its extension.

```
dir2df(
    projroot/'audio'/'source',
    addcols=['barename']
)

# If you see an empty table here, no source audio files could be found.
```

Overwrite the following parameters according to the particulars of your audio files and project:

```
# How many channels do your audio files have?
# (2 for stereo, 1 for mono)
num_channels = 2

# How many speakers can be heard on each channel?
# (If you recorded stereo files using one lapel microphone per speaker, but there is some
# amount of "bleed" of the voices across the channels, select 2.)
num_speakers = 2

# What file type do you want for the diarized output?
output_type = 'eaf' # 'eaf' or 'TextGrid'

# The speech segments found by pyannote sometimes cut off certain high-frequency sounds ne
# beginnings and ends of segments.
# We recommend extending each speech segment, whenever possible, by some fraction of a sec
# (Use 0.250 seconds if you are unsure.)
buffer = 0.250

# If your output is .eaf, ignore this.
# If your output is .TextGrid, how should we label the speech segments?
# (The default is to mark speech with * and to leave silent segments blank.)
speech_label = '*' if output_type == 'TextGrid' else ''
```

Step 1: Prepare audio

To prepare the audio files, we will extract each channel from the source `.wav` file and downsample it.

The variable `channelmap` defines a mapping of input audio channels to output directories. Here channel `1` will map to the `audio/left` subdirectory and channel `2` to `audio/right`.

For mono input files, a simpler `channelmap` will be created that maps channel `1` to the subdirectory `audio/downsampled`.

```
channelmap = {
    1: 'left',
    2: 'right'
}
if num_channels == 1:
    channelmap = {
        1: 'downsampled'
```

```
    }
```

Extract and downsample audio

Next we use sox to extract audio files consisting of a single channel (left or right) from the source audio that has been downsampled to 16000 Hz.

The `compare_dirs` function finds `source` files that do not yet have corresponding `left` or `right` output files (or `downsampled` for mono). The `ext1` and `ext2` values ensure that `compare_dirs` only looks for `.wav` files in the corresponding directories. `compare_dirs` returns a dataframe in which each row contains a file that requires processing.

We iterate over the rows of the `todo` dataframe and use `prep_audio` to extract one channel of audio and downsample. The resulting `.wav` file is stored in a `left` or `right` subdirectory. The inclusion of `relpath` in the output filepath also ensures that the `source` directory structure is replicated in the output directory.

```
verbose = True # Set to false to suppress progress messages.

# Loop over the channels defined in `channelmap`.
for chan_num, chan_name in channelmap.items():
    srcdir = projroot/'audio'/'source'
    chandir = projroot/'audio'/chan_name

    # Find input stereo files that don't have a corresponding
    # left|right pre-processed file.
    todo = utils.compare_dirs(
        dir1=srcdir, ext1='.wav',
        dir2=chandir, ext2='.wav'
    )

    # Loop over the files that require processing.
    for row in todo.itertuples():
        infile = srcdir/row.relpath/row.fname
        outfile = chandir/row.relpath/row.fname

        # Create pre-processed output file for left|right channel.
        if verbose:
            print(f'prep_audio: {outfile}')
        utils.prep_audio(infile, outfile, chan_num)
```

Step 2: Diarize the prepared audio

Instantiate the pipeline

In this step, we will download and instantiate the pretrained speaker diarization pipeline from pyannote. We will also manually override one of the hyper-parameters, `pipeline.segmentation.min_duration_off`, which specifies the minimum duration of a non-speech segment. The original default value of 0.582 meant that if the diarization found a <0.582 second silence in the middle of a speech segment, it would be ignored and treated as part of that speech segment. In other words, a silence needs to be >0.582 seconds before it is treated as a pause boundary between speech segments. We found that this default value produced very long speech segments that were unwieldy to transcribe, and so we preferred a shorter value for `min_duration_off` (0.3). You may want to tweak this for your own purposes.

Note that `pipeline` only needs to be instantiated once. The later cell that performs diarization can be executed repeatedly without reinstantiating `pipeline`.

```
pipeline = Pipeline.from_pretrained(
    "pyannote/speaker-diarization",
    use_auth_token=auth_token
)

parameters = {
    "segmentation": {
        "min_duration_off": 0.3,
    },
}

pipeline.instantiate(parameters)
```

Diarize the channels

The outputs of diarization are annotation files for each of the prepared audio files.

First the `compare_dirs` function finds left and right `.wav` files that do not have a corresponding `.eaf` or `.TextGrid` (whichever desired format you specified earlier). The `ext1` and `ext2` values ensure that `compare_dirs` only looks for `.wav` and `.eaf`/`.TextGrid` files in their corresponding directories.

Each `.eaf`/`.TextGrid` is created by the `diarize` function while iterating over `todo`. Note that the output filename is constructed by `barename` (the input file's filename without extension) plus `.eaf`/`.TextGrid` as the extension.

```
fill_gaps = '' if num_channels == 1 else None
# Loop over the channels defined in `channelmap`.
for chan_num, chan_name in channelmap.items():
    wavdir = projroot/'audio'/chan_name
    outdir = projroot/'diarized'/output_type/chan_name

    # Find input pre-processed files that don't have a corresponding
    # left|right .eaf/.TextGrid.
    todo = utils.compare_dirs(
        dir1=wavdir, ext1='.wav',
        dir2=outdir, ext2=f'.{output_type}'
    )

    # Loop over the files that require diarizing.
    for row in todo.itertuples():
        wavfile = wavdir/row.relpath/row.fname
        outfile = outdir/row.relpath/f'{row.barename}.{output_type}'

        # Diarize to create .eaf/.TextGrid for left|right prepared audio file.
        if verbose:
            print(f'diarize: {outfile}')
        diarization = utils.diarize(
            wavfile, pipeline, outfile, num_speakers, buffer, speech_label,
            fill_gaps
        )
```

Step 3: Combine diarized outputs

In this step we combine the tiers in the `left` and `right` output files into a single `.eaf` or `.TextGrid` file. *This step only needs to be run if your source audio files are stereo.*

We also use a sorting algorithm to assign labels to the combined tiers. For the interviews in our corpus project, we expected the person doing the most talking to be the person being interviewed. We also expected the average intensity of each person's speech to be greater in the channel corresponding to the closer microphone. On the basis of this sorting we assign the names `Subject probable`, `Subject unlikely`, `Interviewer probable`, and `Interviewer unlikely` to the annotation tiers.

If the sorting algorithm fails to cleanly find the subject and interviewer on separate channels, then `unknown` names are assigned.

```
if num_channels == 1:
    sys.stderr.write('You do not need to run this step if you are working with mono audio

else:
    # Find existing left|right label files that need to be combined.
    annodir = projroot/'diarized'/output_type
    (annodir/'combined').mkdir(parents=True, exist_ok=True)
    todo = utils.compare_dirs(
        dir1=annodir/'left', ext1=output_type,
        dir2=annodir/'combined', ext2=output_type
    )

    # Loop over label files to be combined and sort the tiers.
    for row in todo.itertuples():
        tierdfs, tiernames = utils.sort_tiers(
            annodir,
            list(channelmap.values()),
            projroot/'audio'/'source',
            row.relpath,
            row.fname
        )
        outfile = annodir/'combined'/row.relpath/f'{row.barename}.{output_type}'
        outfile.parent.mkdir(parents=True, exist_ok=True)
        if verbose:
            print(f'sorted: {outfile}')
        if output_type == 'TextGrid':
            df2tg(
                tierdfs,
                tnames=tiernames,
                lbl='label',
                ftype='praat_short',
                fill_gaps='',
                outfile=outfile
            )
        elif output_type == 'eaf':
            utils.write_eaf(tierdfs, tiernames, outfile, speech_label, 't1', 't2')
```

The diarization process is now complete. You should have diarized `.eaf` or `.TextGrid` files in your project directory.