CS328 - Assignment 2 Part 2

Speaker Identification

**Group Assignment**

**Deadline: May 5, 2023 11:59 PM**

# Assignment Description and Goals

In this assignment, you will implement speaker identification by (1) collecting labeled audio data, (2) extracting informative audio features, (3) training and evaluating a classifier.

**Background**

Speaker identification is an important problem in many domains. It can be used for authentication, forensic analysis and, in the case of mobile health analytics, it can be used to identify conversation patterns, which can be used to infer social habits over time.

Speaker identification can be formulated as a classification problem, much like the activity recognition task. Informative features are extracted over windows of audio data and feature-label pairs are then used to train a classifier, which can later be used to predict labels for unlabelled audio data.

**Dependencies**

Implementing audio features by yourself in Python would be far more time-consuming than we can allow for an undergraduate course in mobile health. You should understand what the features are and have a high-level idea of how they are being computed, but *most* of the code is provided for you.

Python does not have built-in audio processing techniques, so we have to look to external libraries to get things working. In this assignment, we will make use of the following libraries:

**Python Speech Features**: Speech feature extraction in Python. You can install it using the following command:

pip install python\_speech\_features

**Audiolazy**: We need this to calculate formants (explained later). You can install it using the following command:

pip install audiolazy

# Collecting Data

You should collect audio data for **at least 3** speakers, as well as for **no speaker**, for a total of 4 classes. Each session should be at least 3 minutes long, containing mostly speech of the corresponding subject. Speak naturally so that it will generalize later! Try to capture variance, for instance, by varying the background noise or the location (acoustics are different in different rooms), the microphone’s distance from the speaker, etc. For the no-speaker session, try to capture different acoustic environments.

Use this online tool to record speech: [https://voice-recorder-online.com](https://voice-recorder-online.com/)

Name your recorded speech data file in the following format:

“speaker-data-\*-#.wav” where \* is replaced by the speaker’s name or other identifier, and # is the index of that speaker’s data, in case you do multiple data collection sessions per user. The label is a unique integer corresponding to each subject, e.g., 0 refers to Tom, 1 refers to Jerry, etc.

Example: If Tom is recording his voice, then he will rename the record.wav (generated from online tool) as speaker-data-Tom-1.wav

Next, use the script convert\_wav\_to\_csv.py to convert each of “.wav” files to “.csv”. You will be working with csv files for feature engineering, training, and testing. Instructions on how to use convert\_wav\_to\_csv.py can be found as header comments in the convert\_wav\_to\_csv.py file itself.

# Features

Extracting features from audio data is a very specialized domain, as it requires many sophisticated signal processing techniques and precise modeling of speech transfer (how does the speech wave change between the larynx to the microphone?). We cover the details of some of these features below.

To begin, open features.py. You will not need to compute any of them, but you will need to manipulate their outputs. More specifically, you need to fill out the following methods:

\_compute\_formant\_features()

\_compute\_delta\_coefficients()

The rest of the code (including the fitting and prediction) has been implemented for you, most of it is in speaker\_identification\_train.py. We changed features.py to satisfy an object-oriented programming design by creating a FeatureExtractor class. Method calls are quite similar. From outside the class, calls are made by instantiating a FeatureExtractor instance, say feature\_extractor. Then we call the extract\_features() method as follows:

feature\_extractor = FeatureExtractor()

x = feature\_extractor.extract\_features(window)

This is just for reference, you should not need to modify this code.

## **Formant Features: \_compute\_formant\_features()**

A formant of a speech wave refers to a frequency band which has a high concentration of acoustic energy. They correspond to tones produced by the vocal tract and are often used to characterize phonemes, especially vowels, which have nearly distinct frequency patterns.

We provide you with an implementation for extracting formants from audio. However, these can’t be used as features, because there are not a constant number of formats. One audio buffer window may contain 5 formants, while another contains only 4.

One way to get a valid constant-length feature vector is by using the bin counts, i.e. the distribution, of the formant frequencies. This can be done using numpy’s [histogram](https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html) function. You have to make certain you specify the bins, not just the number of bins, so that the comparisons are meaningful. This can be done by setting the range parameter appropriately: The maximum formant frequency we might see is 5500Hz.

## **Mel-Frequency Cepstrum Coefficients: \_compute\_delta\_coefficients()**

### **Background (implemented for you)**

A *cepstrum* is a sequence of numbers characterizing a window of speech.

The cepstrum can be computed using the inverse discrete Fourier Transform (IDFT), which takes a frequency-domain signal and gives you a time-domain signal, of the logarithm of the power spectrum .

Here, the power spectrum of a function (restricted to some properties we’ll omit) isthe spectral decomposition of the [autocorrelation](https://en.wikipedia.org/wiki/Autocorrelation) (correlation of the function and *itself* over time) of that function. The power spectrum is computed as follows:

where is the signal. For more details, check [this page](http://www.practicalcryptography.com/miscellaneous/machine-learning/tutorial-cepstrum-and-lpccs/) out.

Cepstrums were used for decades in speech recognition systems until Paul Mermelstein introduced the idea of the Mel-frequency cepstrum, which is the cepstrum computed in the Mel scale. The Mel scale is a logarithmic frequency scale which better characterizes the properties of human hearing and speech. Computing the Mel-frequency cepstrum coefficients (MFCCs) is a similar but slightly more involved algorithm. In case you are interested in the details, the algorithm is as follows:

1. Break the signal into frames.
2. Compute the power spectrum for each frame.
3. Convolve (multiply) the power spectrum with the mel filterbank.
4. Take the logarithm of all filterbank responses.
5. Take the [Discrete Cosine Transform](https://en.wikipedia.org/wiki/Discrete_cosine_transform) (DCT) of the log filterbank responses. The DCT is similar to the DFT, but breaks the signal down into a composition of cosines only (DFT does both cosines and sines)
6. You should get 22 coefficients back; keep only the coefficients 2-13, since those are useful in speech analytics.

While the above algorithm recommends using only 12 coefficients, our implementation of MFCCs in the starter code uses the mfcc() function from python\_speech\_features which, by default, returns 13 cepstral coefficients (details [here](https://python-speech-features.readthedocs.io/en/latest/#functions-provided-in-python-speech-features-module)). These 13 coefficients characterize the speech signal over each frame similarly to the way your ear does. That makes them very promising features for all sorts of speech analytics tasks, such as phoneme detection and speech recognition. Even for speaker identification, this can be very useful!

We have implemented the MFCC computation in \_compute\_mfcc(), which is already called for you in \_compute\_delta\_coefficients(). See the next section for what you will actually need to implement.

The MFCCs are computed over each 25ms frame, not over the entire 1 second window. The step size is 12.5 ms, which is half the frame size, meaning adjacent frames have 50% overlap. Since we have 1 second of audio data in each window, we should have 1000 / 12.5 = 80 frames. However, the last frame must be excluded because the end index would be at 1012.5 ms, which is beyond the window bound. Therefore, we should get 79 MFCC vectors, each of size 13, with these frame size and step size parameters.

While the MFCCs may be enough to detect whether someone is speaking, it’s not descriptive enough to identify *who* is speaking. We instead want to look at how the MFCCs change over time. These features are called the *Deltas* and *Delta Acceleration* or *Delta-Deltas*. We will only be concerned with computing the Deltas in this assignment.

### **Delta Coefficients (you must implement)**

The MFCC feature vector only characterizes the power spectrum over a single frame. This may be enough to detect whether someone is speaking, but it’s not descriptive enough to identify *who* is speaking. We would like to capture some of the dynamics of the MFCCS; in other words how is it changing over time? These features are called the *deltas* and the second-order feature is called the *delta acceleration* or *delta-deltas* (how the deltas change over time). We will only be concerned with computing the deltas.

The delta coefficients are defined as follows:



where  is the delta coefficients of frame  computed in terms of the *2N* MFCCs  to . Here,  is typically set to 2.

You will need to implement this calculation, given the MFCCs. First, a word on the return value: you need to return a 1D array with 975 elements. Here’s how that works out - as discussed above, there are 79 MFCC vectors with 13 elements each.

**You will compute the delta coefficients separately for each of the 13 coefficients.** Thus, each *ct+n - ct-n* operation should be subtracting a 13-element vector from another 13-element vector with a 13-element vector as the result. With the code we have provided, each of those would be indexed as mfcc\_feats[i,:], where “i” is the appropriate window number for the numerator summation.  **If you are having trouble figuring out how to do this, please let us know through Piazza.**

The output of the computation will be a 75 x 13 array. Why 75 and not 79? Because you have to drop 2 elements at the beginning and end since n = 2 (see below).

Finally, now that you have a 75 x 13 array, you need to flatten it because a single feature vector must be a 1D array. You can use np.flatten() for this, and then you will have an array 75x13 = 975 elements.

Here is some additional guidance on how to do the computation:

* Try to replace loops with numpy functions (np.sum(), in particular) as much as possible, since it will be significantly faster.
* You will also need an outer for loop that iterates over all *t* of the windows and performs this calculation for each one - the number of windows is provided to you in the code as “n\_frames”.
* The denominator is the same for each frame (it doesn’t depend on *t*), so it doesn’t need to be computed in the loop (i.e., you should compute it before or after and divide the entire result by it in one operation).
* Make sure to be careful of the edges of the array - if n = 2, then you need to start the outer loop at window t = 2 because it needs two previous windows to compute the numerator summation (and similarly for the end). You will know you’re likely doing this wrong if you end up with more or less than 75 windows (of 13 elements each) at the end.
* It may be helpful to work out a few steps in the numerator summation by hand to make sure you understand what’s going on and how to implement it in code. **Remember that each *ct* is a 13-element vector.**

For the corner cases, i.e. where or , this will not work, because or will give an out-of-bounds exception. These are unlikely to come in practice, but to protect against it you can either pad the array with zeros or test for and skip these cases. If you pad with zeros, then you should expect (where is 13) features; if you ignore edge cases, you will get features. It’s up to you how you do that, either way is fine so long as it gives correct results for the normal case.

# Classification

Just like in the activity recognition task, we need to train a classifier to identify speakers. Run speaker-identification-train.py to train on the data you have provided. **You will need to change the variable n\_features to match the number of features returned by your extract\_features() method.**

We have provided code that trains two different classifiers - a decision tree and random forest - and reports results using 10-fold cross-validation. **(Optional:)** Feel free to change the classifiers that are tested (this will help improve your performance), and for the best one, report the average accuracy, precision and recall. Once you have found a candidate, change the best\_classifier variable to be a model with the same parameters. This is currently set to a random forest classifier.

The best classifier will then be trained on \*all\* of the data and saved as classifier.pickle.

**Making Predictions on Test Data**

You will follow a similar protocol from that of Assignment 2 Part 1. Record a new set of WAV files, convert each of them to CSV files, and then test the new data using the pickled classifier. - You need to code this up on speaker-identification.py

# Submission and Grading

You are required to commit and push all your code to Github by the deadline (**11:59 PM on Friday May 5**).

Your submitted repo must include the modified Python scripts, the CSV data files you generated, and your pickled classifier (classifier.pickle). Note: If you are having trouble pushing your CSV files to your repository due to the size limit, you can upload the files to Google Drive and add a link to it in the README.

Your repo should also contain a video demonstrating:

* Average accuracy, precision and recall over all folds for the decision tree and random forest classifiers, as well as any other classifiers you might try
* The results of your classifier in the test setting