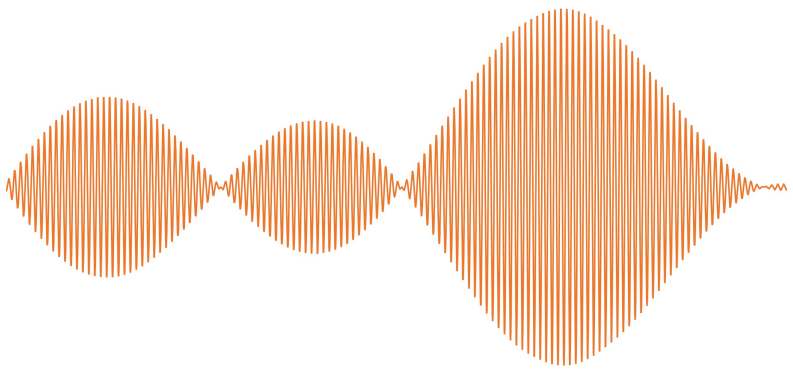
@tyiannak

Theodoros Giannakopoulos

Intro to Audio Analysis: Recognizing Sounds Using Machine Learning



Like my articles? Feel free to vote for me

as ML Writer of the year here

.

Sound analysis is a challenging task, associated to various modern applications, such as speech analytics, music information retrieval, speaker recognition, behavioral analytics and auditory scene analysis for security, health and environmental monitoring. This article provides a brief introduction to basic concepts of **audio feature** extraction, sound **classification** and **segmentation**, with demo examples in applications such as musical genre classification, speaker clustering, audio event classification and voice activity detection.

Python examples are provided in all cases, mostly through the pyAudioAnalysis library. All examples are also provided in this github repo.
With regards to the involved ML methodologies, this article focuses on hand-crafted audio features and traditional statistical classifiers such as SVMs. Deep audio methods are to follow in a future article, as the present article is more about learning to extract audio features that make sense to your classifiers even when you have some tens of training samples.

Prerequisites

Before proceeding deeper to audio recognition, the reader needs to know the basics of audio handling and signal representation: sound definition, sampling, quantization, sampling frequency, sample resolution and the basics of frequency representation. These topics are covered in this article.

Audio Feature Extraction: short-term and segment-based

So you should already know that an audio signal is represented by a **sequence** of **samples** at a given "sample resolution" (usually 16bits=2 bytes per sample) and with a particular **sampling frequency** (e.g. 16KHz = 16000 samples per second).
We can now proceed to the next step: use these samples to **analyze** the corresponding sounds. By "analyze" we can mean anything from: recognize between different types of sounds, segment an audio signal to homogeneous parts (e.g split voiced from unvoiced segments in a speech signal) or group sound files based on their content similarity.
In all cases, we first need to find a way to go from the low-level and voluminous audio data samples to a higher-level

representation of the audio content. This is the purpose of **feature extraction (FE)**, the most common and important task in all machine learning and pattern recognition applications.
FE is about extracting a set of features that are informative with respect to the desired properties of the original data. In our case, we are interested to extract audio features that are capable of discriminating between different audio classes, i.e. different speakers, events, emotions or musical genres, depending on the application subdomain.
The most important concept of audio feature extraction is **short-term windowing** (or **framing**): this simply means that the audio signal is split into short-term windows (or **frames**). The frames can be optionally overlapping.
The length of the frames usually ranges from 10 to 100msecs depending on the application and types of signals. For the non-overlapping case, the **step** of the windowing procedure is equal to the window's **length** (also called "size").
If, on the other hand, step < size, then the frames are overlapping: e.g., a 10msec step for a 40msec window size means a 75% overlap. Usually, a window function (such as hamming) is also applied to each frame.
For each frame (let N be the total number of frames), we extract a set of (short-term) audio features. When the features are directly extracted from the audio sample values, they are called time-domain. If the features are calculated on the FFT values, they are called frequency-domain features. Finally, cepstral features (such as MFCCs) are features that are based on the cepstrum.
As an example, let's assume we only extract the signal's energy (mean of squares of the audio samples) and spectral centroid (the centroid of the FFT's magnitude). This means, that *during this framing procedure, the signal is represented by **a sequence of 2-D short-term feature vectors*** (or two equally-lengthed feature sequences, if you like).

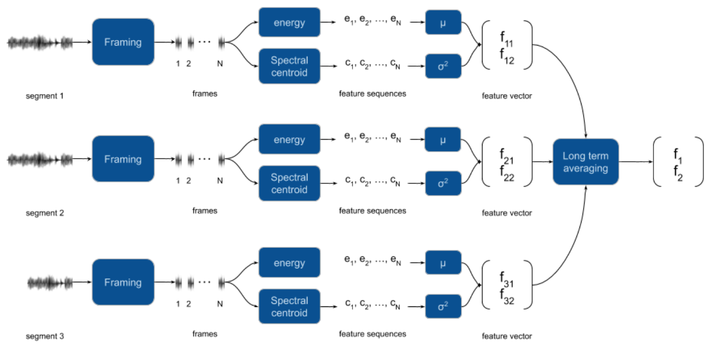
So how can we use these arbitrary–size sequences to analyze the respective signal? Imagine you want to build a classifier to discriminate between two audio classes, say speech and silence. Your initial training data are audio files and corresponding class labels (one class label per **whole** audio file). If these files have the same duration, the corresponding short–term feature vector sequences will have the same length. What happens, though, in the general case of arbitrary durations?

How do we represent arbitrary–sized audio segments?

One solution would be to zero pad the feature sequences up to the maximum duration of the dataset and then concatenate the different short–term feature sequences to a single feature vector. But that would (a) lead to very high dimensionality (and therefore the need for more data samples to achieve training) and (b) be very dependent on the temporal positions of the feature values (as each feature would correspond to a different timestamp).
A more common approach followed in traditional audio analysis is to extract a set of **feature statistics** per **fix–sized segment**. *The segment–level statistics extracted over the short–term feature sequences are the representations for each fix–sized segment. The final signal representation can be the long–term average of the segment statistics.*

> ... segment feature statistics is the simplest way to go

As an example, consider an audio signal of 2.5 seconds. We select a short–term window of 50 msecs and a 1–sec segment. According to the above, the energy and spectral centroid sequences will be extracted for each 1–sec segment. The length of the sequences N will be equal to 1 / 0.050 = 20. Then, the μ and σ of each sequence are extracted for each 1–sec segment, as the segment feature statistics. These are finally long–term averaged, resulting in the final signal representation. (Note that the last segment is 0.5 long, so the statistics are extracted on a shorter segment)



Notes:

1. Short–term **frame sizes** usually range from 10 to 100 msecs. Longer frames mean better frequency representations (more samples to compute FFT and therefore each FFT bin corresponds to fewer Hz). But longer frames also mean losing in time resolution, as audio signals are nonstationary (consider a frame so long, that it encapsulates two different audio events: the frequency resolution would be very high, but what would the respective features represent?)
2. The long–term averaging step of the segment feature statistics of a signal (described above) is optional, usually adopted during training / testing an audio classifier. It is used to map the whole signal to a single feature vector. This could not be desired in another setup, e.g. when we are interested in segmenting the initial signal.

Audio Feature Extraction: code examples

Example1 uses pyAudioAnalysis to read a WAV audio file and extract short–term feature sequences and plots the energy sequence (just one of the features). Please see inline comments for an explanation, along with these two notes:

1. `read_audio_file()`

   returns the sampling rate (Fs) of the audio file and a NumPy array of the raw audio samples. To get the duration in seconds, one simply needs to divide the number of samples by Fs

2. `ShortTermFeatures.feature_extraction()`

   function returns (a) a 68 x 20 short-term feature matrix, where 68 is the number of short-term features implemented in the library and 20 is the number of frames that fit into the 1-sec segments (1-sec is used as mid-term window in the example) (b) a 68-length list of strings that contain the names of each feature implemented in the library.

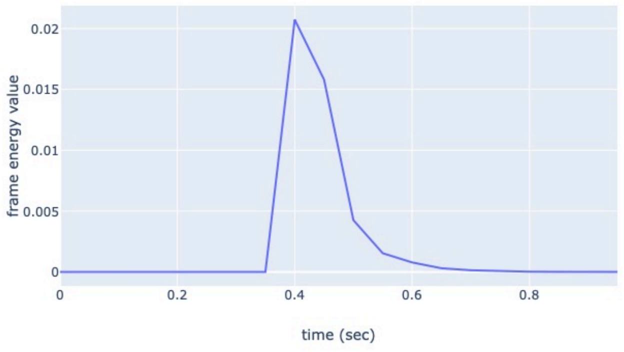
```
# Example 1: short-term feature extraction
from pyAudioAnalysis import ShortTermFeatures as aF
from pyAudioAnalysis import audioBasicIO as aIO
import numpy as np
import plotly.graph_objs as go
import plotly
import IPython
# read audio data from file
# (returns sampling freq and signal as a numpy array)
fs, s = aIO.read_audio_file("data/object.wav")
# play the initial and the generated files in notebook:
IPython.display.display(IPython.display.Audio("data/obje
# print duration in seconds:
duration = len(s) / float(fs)
print(f'duration = {duration} seconds')
# extract short-term features using a 50msec non-overlap
win, step = 0.050, 0.050
[f, fn] = aF.feature_extraction(s, fs, int(fs * win),
int(fs * step))
print(f'{f.shape[1]} frames, {f.shape[0]} short-term fea
print('Feature names:')
for i, nam in enumerate(fn):
print(f'{i}:{nam}')
# plot short-term energy
# create time axis in seconds
time = np.arange(0, duration - step, win)
# get the feature whose name is 'energy'
energy = f[fn.index('energy'), :]
mylayout = go.Layout(yaxis=dict(title="frame energy valu
xaxis=dict(title="time (sec)"))
plotly.offline.iplot(go.Figure(data=[go.Scatter(x=time,
y=energy)],
layout=mylayout))
```

results in (cropped due to length):

```
duration = 1.03 seconds
20 frames, 68 short-term features
Feature names:
0:zcr
1:energy
2:energy_entropy
3:spectral_centroid
4:spectral_spread
5:spectral_entropy
6:spectral_flux
7:spectral_rolloff
8:mfcc_1
...
31:chroma_11
32:chroma_12
33:chroma_std
34:delta zcr
35:delta energy
...
```

```
66:delta chroma_12
67:delta chroma_std
```

and the following graph:

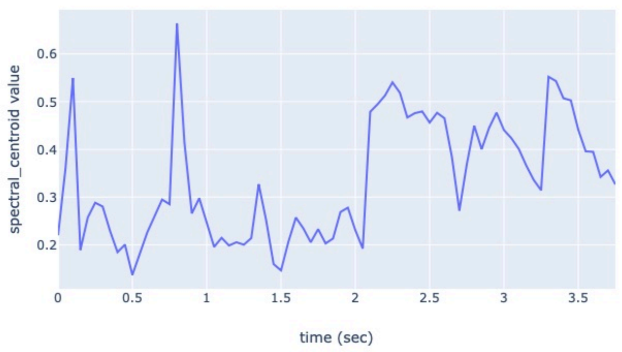


Example2 demonstrates the spectral centroid short-term feature. Spectral centroid is simply the centroid of the FFT magnitude, normalized in the [0, Fs/2] frequency range (e.g, if Spectral Centroid = 0.5 this is equal to Fs/4 measured in Hz).

```
# Example 2: short-term feature extraction:
# spectral centroid of two speakers
from pyAudioAnalysis import ShortTermFeatures as aF
from pyAudioAnalysis import audioBasicIO as aIO
import numpy as np
import plotly.graph_objs as go
import plotly
import IPython
# read audio data from file
# (returns sampling freq and signal as a numpy array)
fs, s = aIO.read_audio_file("data/trump_bugs.wav")
# play the initial and the generated files in notebook:
IPython.display.display(IPython.display.Audio("data/trum
# print duration in seconds:
duration = len(s) / float(fs)
print(f'duration = {duration} seconds')
# extract short-term features using a 50msec non-overlap
win, step = 0.050, 0.050
[f, fn] = aF.feature_extraction(s, fs, int(fs * win),
int(fs * step))
```

```
print(f'{f.shape[1]} frames, {f.shape[0]} short-term fea
# plot short-term energy
# create time axis in seconds
time = np.arange(0, duration - step, win)
# get the feature whose name is 'energy'
energy = f[fn.index('spectral_centroid'), :]
mylayout = go.Layout(yaxis=dict(title="spectral_centroid
xaxis=dict(title="time (sec)"))
plotly.offline.iplot(go.Figure(data=[go.Scatter(x=time,
y=energy)],
layout=mylayout))
```

In this example, the Spectral Centroid sequence is calculated for a recording that contains a 2-sec Donald Trump speech sample, followed by the Bugs Bunny's "What's up doc?" phrase. It is obvious that the spectral centroid sequence can be used to discriminate these two speakers in this particular example (higher spectral centroid values correspond to higher frequencies and therefore "brighter" sounds).



In total, 34 short-term features are extracted in pyAudioAnalysis, for each frame, and the ShortTermFeatures.feature_extraction() function also (optionally) extracts the respective delta features. In that, case the total number of features extracted for each short-term frame is 68. The complete list and description of the short-term features can be found in the library's wiki and this publication.

The two first examples used function

`ShortTermFeatures.feature_extraction()`

to extract 68 features per short-term frame. As described in the previous section, in many cases, such as segment-level classification, we also extract segment-level statistics. This is achieved through the

`MidTermFeatures.mid_feature_extraction()`

function, as shown in **Example3**:

```
# Example 3: segment-level feature extraction
from pyAudioAnalysis import MidTermFeatures as aF
from pyAudioAnalysis import audioBasicIO as aIO
# read audio data from file
# (returns sampling freq and signal as a numpy array)
fs, s = aIO.read_audio_file("data/trump_bugs.wav")
# get mid-term (segment) feature statistics
# and respective short-term features:
mt, st, mt_n = aF.mid_feature_extraction(s, fs, 1 * fs,
0.05 * fs, 0.05 * fs)
print(f'signal duration {len(s)/fs} seconds')
print(f'{st.shape[1]} {st.shape[0]}-D short-term feature
print(f'{mt.shape[1]} {mt.shape[0]}-D segment feature st
print('mid-term feature names')
for i, mi in enumerate(mt_n):
print(f'{i}:{mi}')
```

results in:

```
signal duration 3.812625 seconds
76 68-D short-term feature vectors extracted
4 136-D segment feature statistic vectors extracted
mid-term feature names
0:zcr_mean
1:energy_mean
```

```
2:energy_entropy_mean
3:spectral_centroid_mean
4:spectral_spread_mean
5:spectral_entropy_mean
6:spectral_flux_mean
7:spectral_rolloff_mean
8:mfcc_1_mean
...
131:delta chroma_9_std
132:delta chroma_10_std
133:delta chroma_11_std
134:delta chroma_12_std
135:delta chroma_std_std
```

```
MidTermFeatures.mid_feature_extraction()
```

extracts 2 statistics, namely the **mean** and **std** of each short-term feature sequence, using the provided "mid-term" (segment) window size of 1 sec for the example above. Since the duration of the signal is 3.8 sec, and the mid-term window step and size is 1 sec, we expect that **4** mid-term segments will be created and for each one of them a feature statistics vector will be calculated. Also, these segment statistics are computed on the short-term feature sequences of 3.8 / 0.05 = **76** short-term frames. Also, note that the mid-term feature names also contain the segment statistic, e.g. zcr_mean is the mean of the zero-crossing-rate short-term feature.
The first 3 examples showed how we can extract short-term features and mid-term (segment) feature statistics. Function

```
MidTermFeatures.directory_feature_extraction()
```

extracts audio features for all files in the provided folder, so that these data can be used for training a classifier etc.
So it actually calls

```
MidTermFeatures.mid_feature_extraction()
```

for each WAV file and it performs long-term averaging to go from segment feature statistic vectors to a single feature vector. Also, this function is capable of extracting two music beat-related features that are appended in the averaged segment statistics. As an example, let's suppose we want to analyze a song of 120 seconds, with a short-term window (and step) of 50 msecs and a mid-term (segment) window and step of 1 second.
The following steps will occur during the

```
MidTermFeatures.directory_feature_extraction()
```

call:

1. 120 / 0.05 = 2400 68-D short-term feature vectors are extracted
2. 120 136-D feature statistics (mean and std of the 68-D vector sequences) are computed
3. the 120 136-D are long-term averaged for the whole song and (optionally) two beat-features are appended (beat has to be computed in a file-level as it needs long-term information), leading to a final feature vector of 138 values.

Example4 demonstrates the usage of

```
MidTermFeatures.directory_feature_extraction()
```

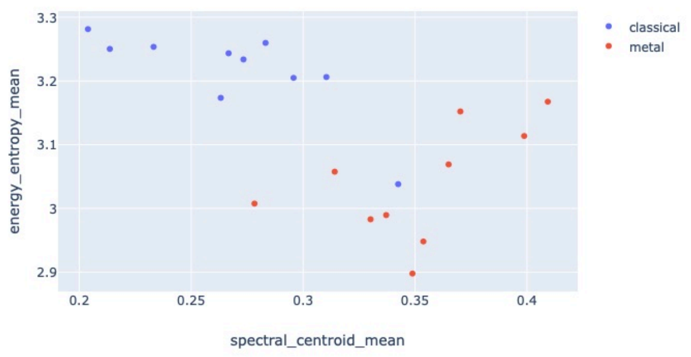
to extract file-level features (averages of segment feature statistics) for 20 2-sec music samples (separate WAV files) from two musical genre categories, namely classical and heavy metal. For each of the 2-sec song segment

```
MidTermFeatures.directory_feature_extraction()
```

extracts the 138-D feature vector, as described above. Then we select to plot 2 from these features, using different colors for the two audio classes (classical and metal):

```
# Example4: plot 2 features for 10 2-second samples
# from classical and 10 from metal music
from pyAudioAnalysis import MidTermFeatures as aF
import os
import numpy as np
import plotly.graph_objs as go
import plotly
dirs = ["data/music/classical", "data/music/metal"]
class_names = [os.path.basename(d) for d in dirs]
m_win, m_step, s_win, s_step = 1, 1, 0.1, 0.05
# segment-level feature extraction:
features = []
for d in dirs: # get feature matrix for each directory (
f, files, fn = aF.directory_feature_extraction(d, m_win,
s_win, s_step)
features.append(f)
# (each element of the features list contains a
# (samples x segment features) = (10 x 138) feature matr
print(features[0].shape, features[1].shape)
# select 2 features and create feature matrices for the
f1 = np.array([features[0][:, fn.index('spectral_centroi
features[0][:, fn.index('energy_entropy_mean')]])
f2 = np.array([features[1][:, fn.index('spectral_centroi
features[1][:, fn.index('energy_entropy_mean')]])
# plot 2D features
plots = [go.Scatter(x=f1[0, :],  y=f1[1, :],
name=class_names[0], mode='markers'),
go.Scatter(x=f2[0, :], y=f2[1, :],
name=class_names[1], mode='markers')]
mylayout = go.Layout(xaxis=dict(title="spectral_centroid
yaxis=dict(title="energy_entropy_mean"))
plotly.offline.iplot(go.Figure(data=plots, layout=mylayo
```

This example plots the size of the feature matrices for the two classes (10 x 138 as explained above) and returns the following plot of the distributions for the two selected features (mean of spectral centroid and mean of energy entropy):



It can be seen that the two features can discriminate between the two classes with very high accuracy (only one classical song sample will be misclassified by a simple linear classifier). In particular, the mean of spectral centroid values has higher values for the metal samples, while the mean of energy entropy higher values for the energy entropy samples. Of course, this is just a small demo on a very simple task and with few samples. As we will see in the next Section, classification based on audio features is not always easy and requires more than two features…

Audio classification: train the audio classifier

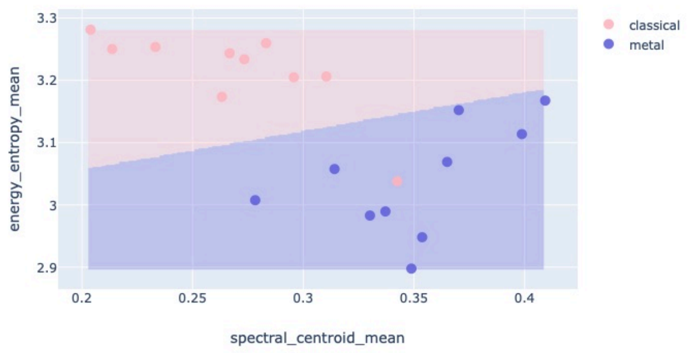
Having seen how to extract audio feature vectors per short-term frame, segment and for whole recordings, we can now proceed to building supervised models for particular classification tasks. All we need to have is a set of audio files and respective class labels. pyAudioAnalysis assumes that audio files are organized in folders and each folder represents a different audio class.

In the example of the previous Section, we've seen how two features differentiated for two musical genre classes, from respective WAV files organized in two folders. **Example5** shows how the same features can be used to train a simple SVM classifier: each point of a grid in the 2-D feature space is then classified to either of the two classes. This is a way of visualizing the **decision surface** of the classifier.

```
# Example5: plot 2 features for 10 2-second samples
# from classical and 10 from metal music.
# also train an SVM classifier and draw the respective
# decision surfaces
from pyAudioAnalysis import MidTermFeatures as aF
import os
import numpy as np
from sklearn.svm import SVC
import plotly.graph_objs as go
import plotly
dirs = ["data/music/classical", "data/music/metal"]
class_names = [os.path.basename(d) for d in dirs]
m_win, m_step, s_win, s_step = 1, 1, 0.1, 0.05
# segment-level feature extraction:
features = []
for d in dirs: # get feature matrix for each directory (
f, files, fn = aF.directory_feature_extraction(d, m_win,
s_win, s_step)
features.append(f)
# select 2 features and create feature matrices for the
f1 = np.array([features[0][:, fn.index('spectral_centroi
features[0][:, fn.index('energy_entropy_mean')]])
f2 = np.array([features[1][:, fn.index('spectral_centroi
features[1][:, fn.index('energy_entropy_mean')]])
# plot 2D features
p1 = go.Scatter(x=f1[0, :],  y=f1[1, :], name=class_name
marker=dict(size=10,color='rgba(255, 182, 193, .9)'),
mode='markers')
p2 = go.Scatter(x=f2[0, :], y=f2[1, :],  name=class_name
marker=dict(size=10,color='rgba(100, 100, 220, .9)'),
mode='markers')
mylayout = go.Layout(xaxis=dict(title="spectral_centroid
yaxis=dict(title="energy_entropy_mean"))
y = np.concatenate((np.zeros(f1.shape[1]), np.ones(f2.sh
f = np.concatenate((f1.T, f2.T), axis = 0)
# train the svm classifier
```

```
cl = SVC(kernel='rbf', C=20)
cl.fit(f, y)
# apply the trained model on the points of a grid
x_ = np.arange(f[:, 0].min(), f[:, 0].max(), 0.002)
y_ = np.arange(f[:, 1].min(), f[:, 1].max(), 0.002)
xx, yy = np.meshgrid(x_, y_)
Z = cl.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx
# and visualize the grid on the same plot (decision surf
cs = go.Heatmap(x=x_, y=y_, z=Z, showscale=False,
colorscale= [[0, 'rgba(255, 182, 193, .3)'],
[1, 'rgba(100, 100, 220, .3)']])
mylayout = go.Layout(xaxis=dict(title="spectral_centroid
yaxis=dict(title="energy_entropy_mean"))
plotly.offline.iplot(go.Figure(data=[p1, p2, cs], layout
```

results in:



In the two examples above (4 and 5) the f1 and f2 matrices can be used to for our classification task, as in any other case when we are using scikit-learn and most ML similar libraries: a matrix X of feature vectors is required (this can be generated by merging f1 and f2 as shown in Example5), along with an array y of the same number of rows with X, corresponding the target values (the class labels).
Then, as with any other classification task, X can be split into train and test (using either random subsampling, fold cross-validation or leave-one-out) and for each train/test split an evaluation metric (such as F1, Recall, Precision, Accuracy or even the whole confusion matrix) is computed. Finally, we report

the overall evaluation metric, as the average among all train/test splits (depending on the method). This procedure can be followed as described in many tutorials (IMO it is best shown in the scikit-learn's webpage), as soon as we have the audio features, as described in the previous examples.
Alternatively, pyAudioAnalysis provides a wrapped functionality that includes both feature extraction and classifier training. This is done in

```
audioTrainTest.extract_features_and_train()
```

function, which
(a) first calls function

```
MidTermFeatures.multi_directory_feature_extraction()
```

, which calls

```
MidTermFeatures.directory_feature_extraction()
```

, to extract feature matrices for all given folders of audio files (assuming each folder corresponds to a class)
(b) generates X and y matrices, aka the feature matrix for the classification task and the respective class labels.
(c) evaluates classifier for different parameters (e.g. C if SVM classifiers are selected)
(d) returns printed evaluation results and saves the best model to a binary file (to be used by another function for testing as shown later)
The following example trains an SVM classifier for the classical/metal music classification task:

```
# Example6: use pyAudioAnalysis wrapper
# to extract feature and train SVM classifier
# for 20 music (10 classical/10 metal) song samples
from pyAudioAnalysis.audioTrainTest import extract_featu
mt, st = 1.0, 0.05
```

```
dirs = ["data/music/classical", "data/music/metal"]
extract_features_and_train(dirs, mt, mt, st, st, "svm_rb
```

final results are here:

```
                    classical                         metal
C         PRE       REC       f1        PRE       REC       f1
0.001     79.4      81.0      80.2      80.6      79.0      79.8
0.010     77.2      78.0      77.6      77.8      77.0      77.4
0.500     76.5      75.0      75.8      75.5      77.0      76.2
1.000     88.2      75.0      81.1      78.3      90.0      83.7
5.000     100.0     83.0      90.7      85.5      100.0     92.2
10.000    100.0     78.0      87.6      82.0      100.0     90.1
20.000    100.0     75.0      85.7      80.0      100.0     88.9
Confusion Matrix:
cla       met
cla       41.50     8.50
met       0.00      50.00
Selected params: 5.00000
```

The overall confusion matrix fo the best C param (in that case C=5), indicates that there is (on average for all subsampling experiments) an almost 9% probability that a classical segment will be classified as metal (which, by the way, makes sense if we remember the feature distribution plots we have seen above).

> *audioTrainTest.extract_features_and_train() from the pyAudioAnalysis lib,* is a wrapper that (a) reads all audio files organized in a list of folders and extracts long–term averaged feature statistics (b) then **trains** a classifier assuming that folder names represent audio classes.

The trained model will be saved in

```
svm_classical_metal
```

(last argument of the function), along with the feature extraction parameters (short-term and segment window sizes and steps) . Note that another file is also created called

```
svm_classical_metalMEANS
```

, that stores the normalization parameters, i.e. the mean and std used to normalize the audio features before training and testing. Finally, apart from SVMs the wrapper supports most scikit-learn classifiers such as decision trees and gradient boosting.

Audio classification: apply the audio classifier

So we have trained an audio classifier to distinguish between two audio classes (classical and metal) based on averages of feature statistics as described before. Now let's see how we can use the trained model to **predict** the class of an **unknown** audio file. Towards this end, we are going to use pyAudioAnalysis'

```
audioTrainTest.file_classification()
```

as shown in **Example7**:

```
# Example7: use trained model from Example6
# to classify an unknown sample (song)
from pyAudioAnalysis import audioTrainTest as aT
files_to_test = ["data/music/test/classical.00095.au.wav
"data/music/test/metal.00004.au.wav",
"data/music/test/rock.00037.au.wav"]
for f in files_to_test:
print(f'{f}:')
c, p, p_nam = aT.file_classification(f, "svm_classical_m
print(f'P({p_nam[0]}={p[0]})')
```

```
    print(f'P({p_nam[1]}={p[1]})')
    print()
```

results in:

```
data/music/test/classical.00095.au.wav:
P(classical=0.6365171558566964)
P(metal=0.3634828441433037)
data/music/test/metal.00004.au.wav:
P(classical=0.1743387880715576)
P(metal=0.825661211928442б)
data/music/test/rock.00037.au.wav:
P(classical=0.2757302369241449)
P(metal=0.7242697630758552)
```

We can see that we have asked the classifier to predict for three files. The first was a classical music segment (not used in the training dataset obviously) and indeed the estimated posterior of classical was higher than that of metal. Similarly, we tested against a metal segment and it was classified as metal with a posterior of 86%. Finally, the 3rd test file does not belong to the two classes (it is a rock song segment), however, the result makes some sense as it is classified to the closer class.

> audioTrainTest.file_classification() gets the trained model, and the path of an unknown audio file and does feature extraction and classifier prediction for the unknown file, returning the (predicted) winner class, the classes posteriors and the respective classes names.

The previous example showed how we can apply the trained audio classifier to an unknown audio file to predict its audio label. In addition to that, pyAudioAnalysis provides function

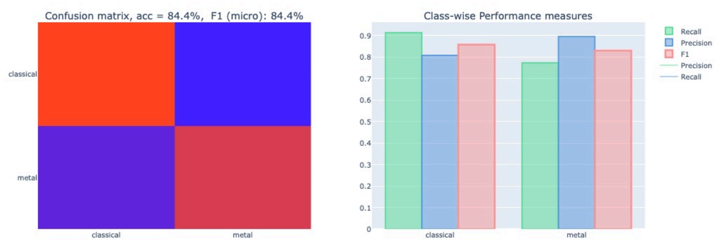
`audioTrainTest.evaluate_model_for_folders()`

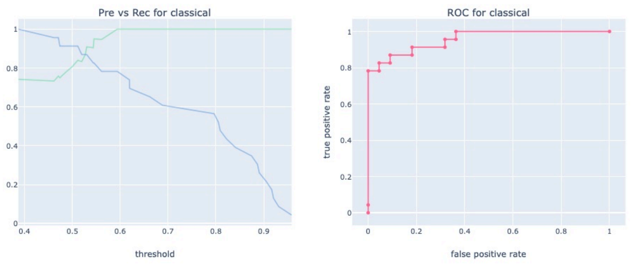
, which accepts a list of folders, assuming their basenames are class names (as we do during training), and repetitively applies a pre-trained classifier on the audio files of each folder. A the end it outputs performance metrics such as confusion matric and ROC curves:

```
# Example8: use trained model from Example6
# to classify audio files organized in folders
# and evaluate the predictions, assuming that
# foldernames = classes names as during training
from pyAudioAnalysis import audioTrainTest as aT
aT.evaluate_model_for_folders(["data/music/test/classica
"data/music/test/metal"],
"svm_classical_metal",
"svm_rbf",
"classical")
```

results in the following figures of the confusion matrix, precision/recall/f1 per class, and Precision/Recall curve and ROC curve for a "class of interest" (here we have provided classical). Note that the 3rd and 4th subplots evaluate the classifier as a **detector** of the class "classical" (last argument). For example, the last plot shows the true positive rate vs the false positive rate, and this is achieved by simulating thresholding of the posterior of the class of interest (classical): as the probability threshold rises, both true positive and false negative rates rise, the question is: how "steep" is the true positive rate's increase? More info about the ROC curve can be found here. The precision/recall curve is equivalent to ROC, but it shows both metrics on the same graph on the y-axis for different thresholds of the posterior which is shown on the x-axis (more info here).
In our example, we can see that for a probability threshold of, say, 0.6 we can have a 100% Precision with around 80% Recall

for classical: this means that all files detected will be indeed classical, while we will be "losing" almost 1 out of 5 "classical" song as metal. Another important note here is that *there is no "best" operation point of the classifier, that depends on the individual application*.





Audio regression: predicting continuous target values

Regression is the task of training a mapping function from a feature space to a continuous target variable (instead of a discrete class). Some applications of regression of audio signals include: speech and/or music emotion recognition using non-discrete classes (emotion and arousal) and music soft attribute estimation (e.g. to detect Spotify's "danceability").
In this article, we demonstrate how regression can be used to detect a choral singing segment's pitch, without using any signal processing approach (e.g. the autocorrelation method). Towards this end, we have used part of the Choral Signing Dataset, which is a set of **acapella** recordings with respective **pitch** annotations. Here, we have selected to use this dataset to produce segment-level pitch annotations: we split the singing recordings to small (0.5 sec) segments and for each segment,

we calculate the mean and standard deviation of the pitch (which provided by the dataset). These two metrics are f0_mean and f0_std respectively and are the two target regression values demonstrated in the following code. Below you can listen to a 0.5–sec sample with a low f0 and low f0 deviation:
and a 0.5–sec sample with a high f0 and high f0 deviation:
Getting 0.5–sec segments from the Choral Signing Dataset leads to thousands of samples, but for demonstration purposes of this article, we have used around 120 training and 120 testing samples, available under

```
data/regression/f0/segments_train
```

and

```
data/regression/f0/segments_train folders
```

of the github repo. Again, to demonstrate training and testing of the regression model we are using pyAudioAnalysis which treats audio segmentation in the following way:

1. given a path that contains audio files and
2. a set of <regression_task>.csv files of the format <audio_filename>, <value>
3. pyAudioAnalysis trains one regression model for each <regression_task>.csv file

In our example,

```
data/regression/f0/segments_train
```

contains 120 WAV files and two 120–line CSV files named f0.csv and f0_std.csv. Each CSV corresponds to a separate regression task and each line of the CSV corresponds to the ground truth of the respective audio file. To train, evaluate, and save the two regression models for our example, the following code is used:

```
# Example9:
# Train two linear SVM regression models
# that map song segments to pitch and pitch deviation
# The following function searches for .csv files in the
# input folder. For each csv of the format <filename>,<v
# a separate regresion model is trained
from pyAudioAnalysis import audioTrainTest as aT
aT.feature_extraction_train_regression("data/regression/
0.5, 0.5, 0.05, 0.05,
"svm", "singing", False)
```

Since

`data/regression/f0/segments_train`

contains two CSVs, namely

`f0.csv`

and

`f0_std.csv`

, the above code results in two models:

`singing_f0`

and

`singing_f0_std`

(

`singing`

prefix is provided as a 7th argument in the function above and is used for all trained models).
Also, this is the result of the evaluation process executed internally by the

`audioTrainTest.feature_extraction_train_regression()`

function:

```
Analyzing file 1 of 120: data/regression/f0/segments_tra
Analyzing file 2 of 120: data/regression/f0/segments_tra
Analyzing file 3 of 120: data/regression/f0/segments_tra
Analyzing file 4 of 120: data/regression/f0/segments_tra
Analyzing file 5 of 120: data/regression/f0/segments_tra
...
...
...
Analyzing file 119 of 120: data/regression/f0/segments_t
Analyzing file 120 of 120: data/regression/f0/segments_t
Feature extraction complexity ratio: 44.7 x realtime
Regression task f0_std
Param           MSE             T-MSE           R-MSE
0.0010          736.98          10.46           661.43
0.0050          585.38          9.64            573.52
0.0100          522.73          9.17            539.87
0.0500          529.10          7.41            657.36
0.1000          379.13          6.73            541.03
0.2500          361.75          5.09            585.60
0.5000          323.20          3.88            522.12
1.0000          386.30          2.58            590.08
5.0000          782.14          0.99            548.65
10.0000         1140.95         0.47            529.20
Selected params: 0.50000
Regression task f0
Param           MSE             T-MSE           R-MSE
0.0010          3103.83         44.65           3121.97
0.0050          2772.07         41.38           3098.40
0.0100          2293.79         37.57           2935.42
0.0500          1206.49         19.69           2999.49
0.1000          1012.29         13.94           3115.49
0.2500          839.82          8.64            3147.30
0.5000          758.04          5.62            2917.62
1.0000          689.12          3.53            3087.71
5.0000          892.52          1.07            3061.10
```

```
10.0000        1158.60        0.47           2889.27
Selected params: 1.00000
```

The 1st column on the results above represents the classifier's parameter evaluated during the experiment. The 2nd column is the Mean Square Error (MSE) of the estimated parameter (f0_std and f0 for the two models), measured on the internal test (validation) data of each experiment. The 3rd column shows the training MSE and the 4th column shows an estimate of the random model (to be used as a baseline metric). We can see that the f0_std target is much harder to estimate through a regression model: the best MSE achieved (320) is just slightly better than the average random MSE (around 550). On the other hand, for the f0 target, the trained regression model achieves a 680 MSE with a baseline error of around 3000. In other words, the model achieves a x5 performance-boosting related to the random model, while for the f0_std this boosting is just around x1.5.
Now, once the two regression models are trained, evaluated and saved, we can use them to map any audio segment to either f0 or f0_std. **Example10** demonstrates how to do this using

```
audioTrainTest.file_regression()
```

:

```
# Example10
# load trained regression model for f0 and apply it to a
# of WAV files and evaluate (use csv file with ground tr
import glob
import csv
import os
import numpy as np
import plotly.graph_objs as go
import plotly
from pyAudioAnalysis import audioTrainTest as aT
# read all files in testing folder:
```

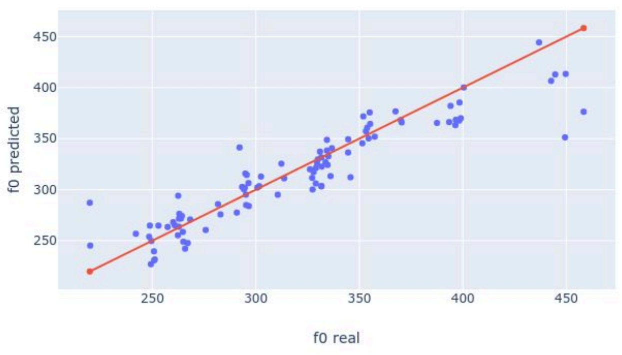
```
wav_files_to_test = glob.glob("data/regression/f0/segmen
ground_truths = {}
with open('data/regression/f0/segments_test/f0.csv', 'r'
reader = csv.reader(file, delimiter = ',')
for row in reader:
ground_truths[row[0]] = float(row[1])
estimated_val, gt_val = [], []
for w in wav_files_to_test: # for each audio file
# get the estimates for all regression models starting w
values, tasks = aT.file_regression(w, "singing", "svm")
# check if there is ground truth available for the curre
if os.path.basename(w) in ground_truths:
# ... and append ground truth and estimated values
# for the f0 task
estimated_val.append(values[tasks.index('f0')])
gt_val.append(ground_truths[os.path.basename(w)])
# compute mean square error:
mse = ((np.array(estimated_val) - np.array(gt_val))**2).
print(f'Testing MSE={mse}')
# plot real vs predicted results
p = go.Scatter(x=gt_val,  y=estimated_val, mode='markers
mylayout = go.Layout(xaxis=dict(title="f0 real"),
yaxis=dict(title="f0 predicted"),
showlegend=False)
plotly.offline.iplot(go.Figure(data=[p,
go.Scatter(x=[min(gt_val+
estimated_val),
max(gt_val+
estimated_val)],
y=[min(gt_val+
estimated_val),
max(gt_val+
estimated_val)])],
layout=mylayout))
```

In this example, we demonstrate how

`audioTrainTest.file_regression()`

can be used for a set of files from a test dataset. Note that this function returns decisions and task names for all available regression tasks that start with the provided prefix (in our case "singing"). The results are shown below: we can see that the real and predicted values are pretty close for the f0 task.

```
Testing MSE=492.7141430351564
```



audioTrainTest.feature_extraction_train_regression() reads a folder of WAV files and assumes that each CSV of (<path>,<value>) format, is a regression ground-truth file. It then extracts audio features, trains and saves the respective number of models using a prefix (provided also as argument). audioTrainTest.file_regression() reads the saved models and returns predicted regression outputs for all tasks.

About Audio Segmentation

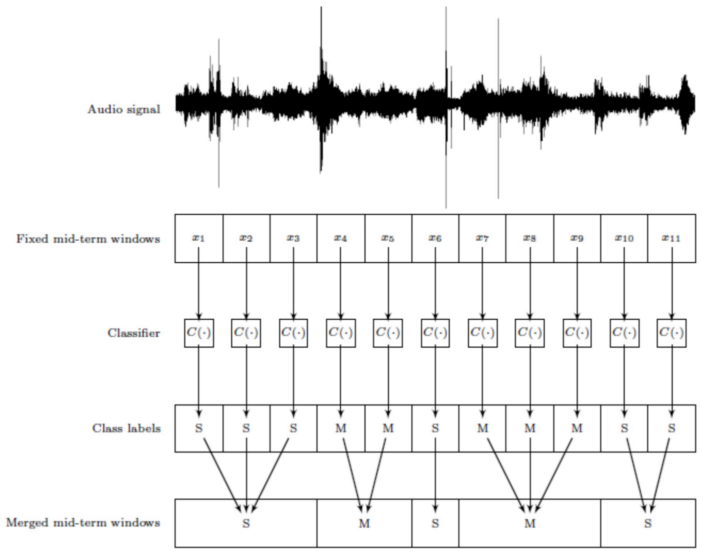
Until now we have seen how to train supervised models that map segment-level audio feature statistics to either class labels (audio classification) or real-valued targets (audio regression). Also, we have seen how to use these models to predict the label of an unknown audio file, e.g. a speech utterance or a whole song or a song's segment. In all these cases, the assumption followed was that the unknown audio signals **belonged to a single label**. For example, a song belongs to a particular genre, a singing segment has a particular pitch value and a speech utterance has a particular emotion. However, in real-world applications, there are many cases in which audio signals are not segments of homogeneous content, but complex audio streams that contain many successive segments of different content labels. A recording of a real-world dialog, for instance, is a sequence of labels of speaker identities or emotions.

> Real-world recordings are not segments of homogeneous content but sequences of segments of different labels

For that reason, audio **segmentation** is an important step of audio analysis and it is about segmenting a long audio recording to a sequence of segments that are of homogeneous content. The definition of homogeneity is relative to the application domain: if, for example, we are interested in speaker recognition, a segment is considered homogeneous if it belongs to the same speaker.

Audio Segmentation: using pretrained models (supervised)

hTAudio segmentation algorithms can be divided into two categories: (a) supervised and (b) unsupervised or semisupervised. Supervised segmentation is based upon a pretrained segment model that is able to classify homogeneous segments. In this section we will show how to achieve segmentation using a simple fix-size segment and a pre-trained model. Suppose you have trained a segment model to distinguish between classes S and M. The segmentation method presented in this Section is as simple as that: split the audio recording to non-overlapping fix-sized segments with the same length as the one used to train the model. Then classify each fix-sized segment of the audio stream using the trained model and finally merge successive segments that contain the same class label. This process is illustrated in the following diagram.



In Example6 we had trained a model that classifies unknown music segments to "metal" and "classical" (model was saved in file

`svm_classical_metal`

). Let's use this model to segment a 30-sec recording that contains both metal and classical (non-overlapping) parts. This recording is stored in

`data/music/metal_classical_mix.wav`

of the article's code. Also,

`data/music/metal_classical_mix.segment`

contains the respective **ground-truth** annotation file of the format <start_segment_sec>\t<end_segment_sec>\t<segment_label>. This is the ground truth file:

```
0       7.5     classical
7.5     15      metal
```

```
15      19      classical
19      29      metal
```

The fix-window supervised segmentation functionality is implemented in function

`audioSegmentation.mid_term_file_classification()`

, as shown in **Example11**:

```
# Example 11
# Supervised audio segmentation example:
#  - Apply model "svm_classical_metal" to achieve fix-si
#    on file data/music/metal_classical_mix.wav
#  - Function audioSegmentation.mid_term_file_classifica
#    the mid-term step that has been used when training
#  - data/music/metal_classical_mix.segments contains th
from pyAudioAnalysis.audioSegmentation import mid_term_f
from pyAudioAnalysis.audioTrainTest import load_model
labels, class_names, _, _ = mid_term_file_classification
"svm_classical_metal", "svm_rbf",  True,
"data/music/metal_classical_mix.segments")
print("\nFix-sized segments:")
for il, l in enumerate(labels):
print(f'fix-sized segment {il}: {class_names[int(l)]}')
# load the parameters of the model (actually we just wan
cl, m, s, m_classes, mt_win, mt_step, s_win, s_step, c_b
# print "merged" segments (use labels_to_segments())
print("\nSegments:")
segs, c = labels_to_segments(labels, mt_step)
for iS, seg in enumerate(segs):
print(f'segment {iS} {seg[0]} sec - {seg[1]} sec: {class
```

`audioSegmentation.mid_term_file_classification()`

returns a list of label ids (one for each fix-sized segment window), a list of class names and the accuracy and confusion matrix (if ground truth is also provided, as in the example above). The

```
labels
```

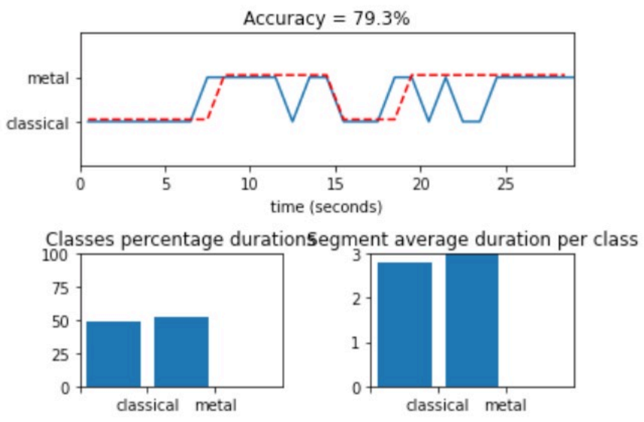
list corresponds to fix-sized segments of length equal to the segment step used during training of the model (1 second in the above example, according to Example6). That's why we use

```
audioTrainTest.load_model()
```

, to load the segment window directly from the model file. Also, we use

```
audioSegmentation.labels_to_segments()
```

to generate the list of final segments, based on the simple merging route (i.e concatenate successive 1-sec segments that have the same label).
The output of the above code is the following (red corresponds to ground truth and blue to predicted segment labels):



```
Overall Accuracy: 0.79
Fix-sized segments:
fix-sized segment 0: classical
fix-sized segment 1: classical
fix-sized segment 2: classical
fix-sized segment 3: classical
fix-sized segment 4: classical
fix-sized segment 5: classical
fix-sized segment 6: classical
```

```
fix-sized segment 7: metal
fix-sized segment 8: metal
fix-sized segment 9: metal
fix-sized segment 10: metal
fix-sized segment 11: metal
fix-sized segment 12: classical
fix-sized segment 13: metal
fix-sized segment 14: metal
fix-sized segment 15: classical
fix-sized segment 16: classical
fix-sized segment 17: classical
fix-sized segment 18: metal
fix-sized segment 19: metal
fix-sized segment 20: classical
fix-sized segment 21: metal
fix-sized segment 22: classical
fix-sized segment 23: classical
fix-sized segment 24: metal
fix-sized segment 25: metal
fix-sized segment 26: metal
fix-sized segment 27: metal
fix-sized segment 28: metal
fix-sized segment 29: metal
Segments:
segment 0 0.0 sec - 7.0 sec: classical
segment 1 7.0 sec - 12.0 sec: metal
segment 2 12.0 sec - 13.0 sec: classical
segment 3 13.0 sec - 15.0 sec: metal
segment 4 15.0 sec - 18.0 sec: classical
segment 5 18.0 sec - 20.0 sec: metal
segment 6 20.0 sec - 21.0 sec: classical
segment 7 21.0 sec - 22.0 sec: metal
segment 8 22.0 sec - 24.0 sec: classical
segment 9 24.0 sec - 29.0 sec: metal
```

`audioSegmentation.labels_to_segments()`

returns the more "compact" and useful information for the end "user". Also, note that the segmentation **errors** are either:

(a) due to **misclassifications** of the **segment classifier** (e.g. segments 22 and 23 are misclassified as classical when their true label is metal or
(b) due to **time resolution** issues: e.g. according to ground truth, the 1st segment classical music ends at 7.5 sec, while our model is applied every 1 second, so the best this fix-window methodology will achieve is to either recognize classical until 7 or 8 sec. Obviously, this can be handled through a smaller step in the segment window (i.e. by introducing a segment overlap), however this will be with significant increase in computational demands (more segment-level predictions will take place).

Audio Segmentation: unsupervised

So we've seen how, given a pretrained audio segment model, we can split an audio recording into segments of homogeneous content. In many cases though, we are not aware of the exact classification problem, or we do not have the data to train such a classifier. Such cases require unsupervised or semi-supervised solutions as shown in the use-cases below:
Music segmentation
Extracting structural parts from a music track is a typical use-case where unsupervised audio analysis can be used. Since it is obviously rather difficult to have a classifier that distinguishes between song parts, we can answer the question: *can you group song segments so that segments of the same group sound like they belong to the same song part?* In the following example, M. Jacksons "Billie Jean" is used as input to the previously described segment-level feature extraction process and a simple k-means clustering is applied on the resulting feature vector sequences. Then, the segments of each cluster are concatenated into an artificial recording and saved to audio files. Each artificial "cluster recording" shows how song parts

can be grouped and if this grouping makes some sense in terms of music structure. The code is shown in **Example12**:

```
# Example 12: Unsupervised Music Segmentation
#
# This example groups of song segments to clusters of si
import os, sklearn.cluster
from pyAudioAnalysis.MidTermFeatures import mid_feature_
from pyAudioAnalysis.audioBasicIO import read_audio_file
from pyAudioAnalysis.audioSegmentation import labels_to_
from pyAudioAnalysis.audioTrainTest import normalize_fea
import numpy as np
import scipy.io.wavfile as wavfile
import IPython
# read signal and get normalized segment feature statist
input_file = "data/music/billie_jean.wav"
fs, x = read_audio_file(input_file)
mt_size, mt_step, st_win = 5, 0.5, 0.1
[mt_feats, st_feats, _] = mT(x, fs, mt_size * fs, mt_ste
round(fs * st_win), round(fs * st_win * 0.5))
(mt_feats_norm, MEAN, STD) = normalize_features([mt_feat
mt_feats_norm = mt_feats_norm[0].T
# perform clustering
n_clusters = 5
x_clusters = [np.zeros((fs, )) for i in range(n_clusters
k_means = sklearn.cluster.KMeans(n_clusters=n_clusters)
k_means.fit(mt_feats_norm.T)
cls = k_means.labels_
# save clusters to concatenated wav files
segs, c = labels_to_segments(cls, mt_step)  # convert fl
for sp in range(n_clusters):
count_cl = 0
for i in range(len(c)):     # for each segment in each c
if c[i] == sp and segs[i, 1]-segs[i, 0] > 2:
count_cl += 1
# get the signal and append it to the cluster's signal (
cur_x = x[int(segs[i, 0] * fs): int(segs[i, 1] * fs)]
x_clusters[sp] = np.append(x_clusters[sp], cur_x)
x_clusters[sp] = np.append(x_clusters[sp], np.zeros((fs,
# write cluster's signal into a WAV file
print(f'cluster {sp}: {count_cl} segments {len(x_cluster
wavfile.write(f'cluster_{sp}.wav', fs, np.int16(x_cluste
IPython.display.display(IPython.display.Audio(f'cluster_
```

The above code saves artificial cluster sounds to WAV files and also displays them in a mini player in the notebook itself, but I've also uploaded the cluster sounds to YouTube (couldn't think of an obvious way to embed them in the article). So, let's listen to the resulting clusters and see if they correspond to homogeneous song parts:
This is clearly the **chorus** of the song, repeated twice (second time is much longer though as it includes more successive repetitions and a small solo)
Cluster **2** has a single segment that corresponds to the song's **intro**.
Cluster **3** is the **pre-chorus** of the song
The **4th** cluster contains segments from the **verses** of the song (if you exclude the small segment in the beginning). The 5th cluster is not shown as it just included a very short almost-silent segment at the beginning of the song. In all cases, clusters represented (with some errors of course) structural song components, even using this very simple approach, and without making use of any "external" supervised knowledge, other than similar features may mean similar music content.

> Clusters of song segments may correspond to stuctural song elements if appropriate audio features are used

Finally, note that, executing the code above may result in the same clustering but with different ordering of cluster IDs (and therefore order in the resulting audio files). This is probably due to the k-means random seed.
Speaker diarization
This is the task that, given an unknown speech recording, answers the question: "who speaks when?". For the sake of

simplicity let's assume that we already know the number of speakers in the recording. What is the most straightforward way to solve this task? Obviously, first extract segment-level audio features and then perform some type of clustering, hoping that the resulting clusters will correspond to speaker IDs. In the following example (13), we use the exact same pipeline as the one followed in Example12, where we clustered a song to its structural parts. We have only changed the segment window size to 2 sec with a step of 0.1 sec and a smaller short-term window (50msec), since speech signals are, in general, characterized with faster changes in their main attributes, due to the existence of very different phonemes, some of which last just a few seconds (on the other hand musical note last several msecs, even in the fastest types of music).
So Example13, uses the same rationalle of clustering of audio feature vectors. This time the input signal is a speech signal with 4 speakers (this is known beforehand), so we set our kmeans cluster size to 4:

```
import os, sklearn.cluster
from pyAudioAnalysis.MidTermFeatures import mid_feature_
from pyAudioAnalysis.audioBasicIO import read_audio_file
from pyAudioAnalysis.audioSegmentation import labels_to_
from pyAudioAnalysis.audioTrainTest import normalize_fea
import numpy as np
import scipy.io.wavfile as wavfile
import IPython
# read signal and get normalized segment feature statist
input_file = "data/diarization_example.wav"
fs, x = read_audio_file(input_file)
mt_size, mt_step, st_win = 2, 0.1, 0.05
[mt_feats, st_feats, _] = mT(x, fs, mt_size * fs, mt_ste
round(fs * st_win), round(fs * st_win * 0.5))
(mt_feats_norm, MEAN, STD) = normalize_features([mt_feat
mt_feats_norm = mt_feats_norm[0].T
# perform clustering
n_clusters = 4
```

```
x_clusters = [np.zeros((fs, )) for i in range(n_clusters
k_means = sklearn.cluster.KMeans(n_clusters=n_clusters)
k_means.fit(mt_feats_norm.T)
cls = k_means.labels_
# save clusters to concatenated wav files
segs, c = labels_to_segments(cls, mt_step)  # convert fl
for sp in range(n_clusters):
count_cl = 0
for i in range(len(c)):     # for each segment in each c
if c[i] == sp and segs[i, 1]-segs[i, 0] > 2:
count_cl += 1
# get the signal and append it to the cluster's signal (
cur_x = x[int(segs[i, 0] * fs): int(segs[i, 1] * fs)]
x_clusters[sp] = np.append(x_clusters[sp], cur_x)
x_clusters[sp] = np.append(x_clusters[sp], np.zeros((fs,
# write cluster's signal into a WAV file
print(f'speaker {sp}: {count_cl} segments {len(x_cluster
wavfile.write(f'diarization_cluster_{sp}.wav', fs, np.in
IPython.display.display(IPython.display.Audio(f'diarizat
```

This is the initial recording
And these are the 4 resulting clusters (results are also written in inline audio clips in jupiter notebook again):
In the above example speaker clustering (or speaker diarization as we usually call it) was quite successfull with a few errors at the begining of the segments, mainly due to time resolution limitations (2-sec window has been used). Of course this is not always the case: speaker diarization is a hard task, especially if (a) a lot of background noise is present (b) the number of speakers is unknown beforehand (c) the speakers are not balanced (e.g. a speaker speaks 60% of the time and another speaker just 0.5% of the time).
The code of this article is provided as a jupiter notebook in this GitHub repo.

Tags: machinelearning,machine-learning-tutorials,python,ai,speech-recognition,artificial-intelligence,audio-content,data-science