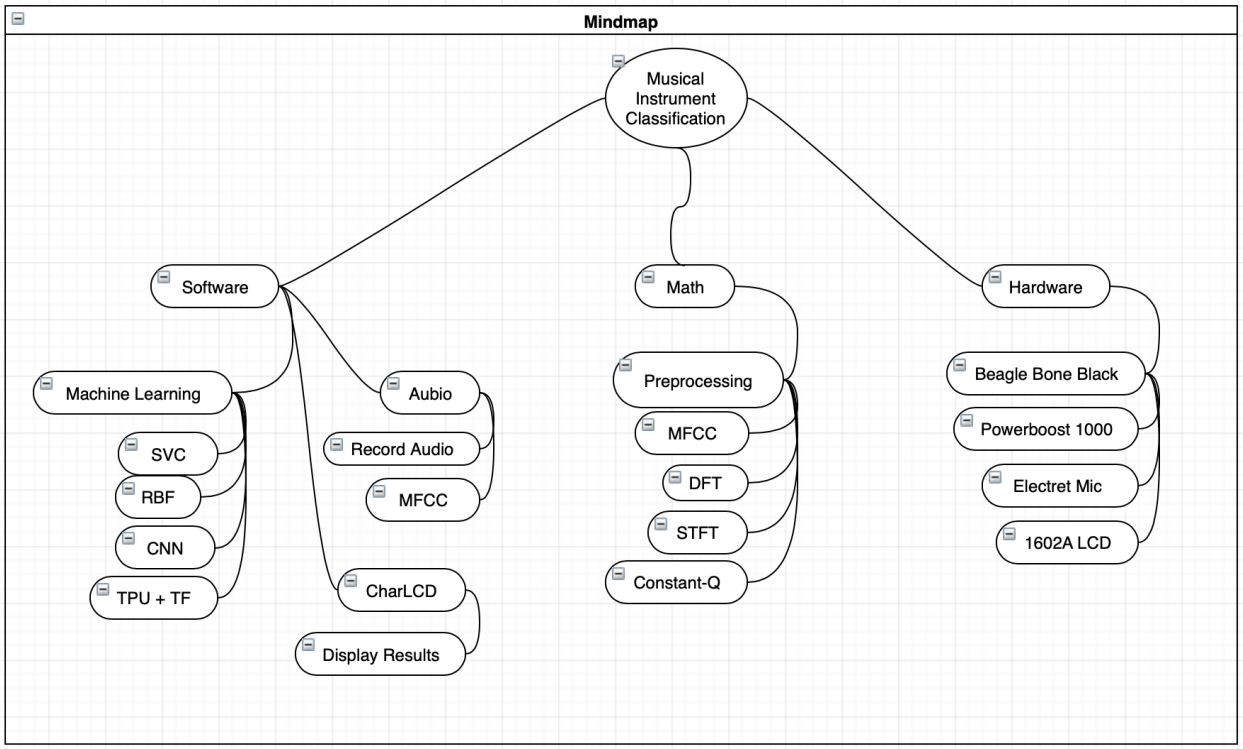


Software:
I was successful in recreating the work of our reference model. I was able to convert the mp3 files to wav files using the pydub framework. In the feature extraction, the resulting svm code used the mfcc output for the classification. I recorded a 99.1% accuracy as opposed to their PDF which reported a 99.3% accuracy. I am now exploring how to make an Gaussian SVM in Tensorflow. The reference model used the C-Support Vector Classification model. https://stackoverflow.com/questions/27912872/what-is-the-difference-between-svc-and-svm-in-scikit-learn

I want to explore the idea of using a constant-Q transform instead of MFCC and try Deep Learning Classification with the results. https://www.tensorflow.org/tutorials/estimators/cnn

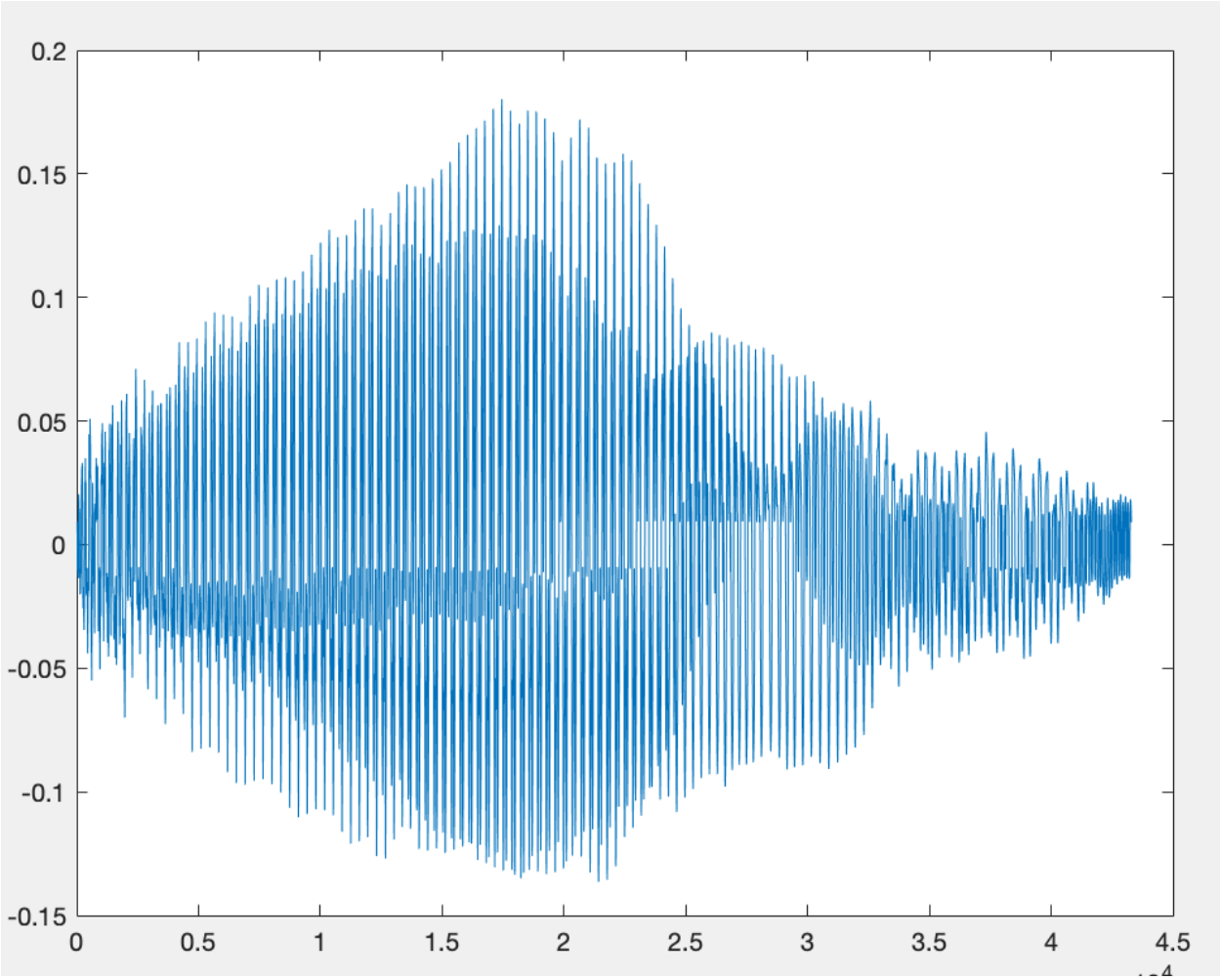
I was also planning to implementing both Constant-Q transform and Convolution Neural Network in C through the aubio framework.

I was able to get $100 in Google Cloud Credits, so I was thinking that if I get a model running on BBB with a basic TF model, then I could throw all the data sets at it and train a strong model on a TPU in the cloud and transfer it over to the BBB. A prototype of a CNN can be made in TF.

I have explored Aubio for reading from an audio stream or wav/mp3 and MFCC, but I am stuck at how to configure Hamming Windows. The MFCC performs dct and idct. I was unable to find a silence removal algorithm in Aubio.

I was successful in displaying arbitrary strings of 32 characters in length. There is an automatic CR/NL after 16 characters.

Math:
Example data provided for cello_A2_1_forte_arco-normal.wav in wav.zip, in the cello folder:
https://drive.google.com/drive/folders/1Q7KeeMlNrKtexKcZi_FYn-Mc74NeN1dg
Here is a plot of the wav file:

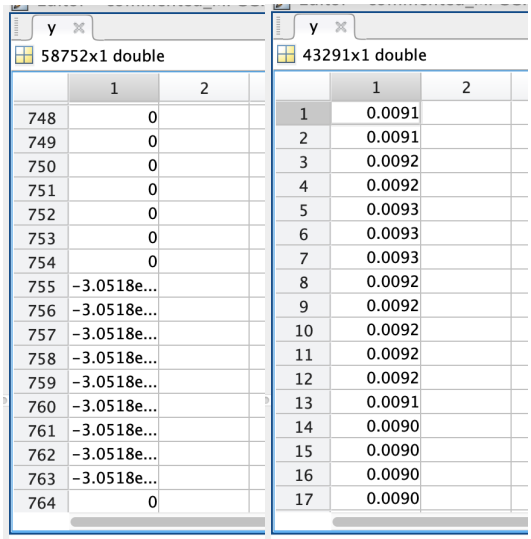


1. Calculate MFCC

For Each wav file:

- Read the file
    - Collected the sampled data
        - Y = 58,752
    - Collected sample weight
        - Fs = 44,100
- 1) Normalization
- Silence removal (58,752 - 15,461 = 43,291)
    - For each sample in Y:
        - If the absolute value of sample is less than 0.009

        - Remove it from Y
    - Y = 43,291 samples after silence removal
    - Beginning silence ends at the 754th sample data and gets removed



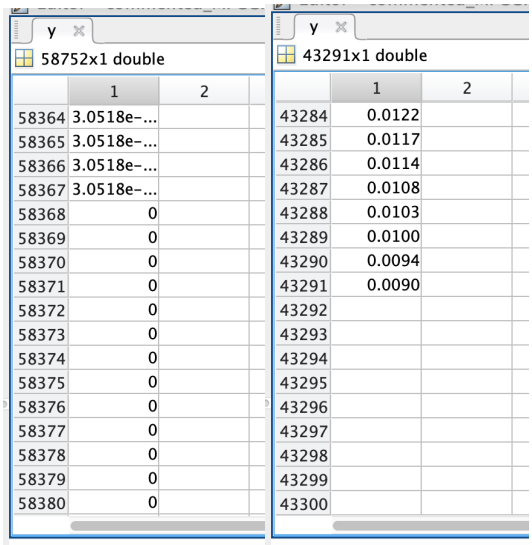
y — 58752x1 double

	1	2
748	0	
749	0	
750	0	
751	0	
752	0	
753	0	
754	0	
755	-3.0518e...	
756	-3.0518e...	
757	-3.0518e...	
758	-3.0518e...	
759	-3.0518e...	
760	-3.0518e...	
761	-3.0518e...	
762	-3.0518e...	
763	-3.0518e...	
764	0	

y — 43291x1 double

	1	2
1	0.0091	
2	0.0091	
3	0.0092	
4	0.0092	
5	0.0093	
6	0.0093	
7	0.0093	
8	0.0092	
9	0.0092	
10	0.0092	
11	0.0092	
12	0.0092	
13	0.0091	
14	0.0090	
15	0.0090	
16	0.0090	
17	0.0090	

    - Ending silence begins at the 58,367th sample data and gets removed.
        - Overall Y is reduced by 15,461 samples



y — 58752x1 double

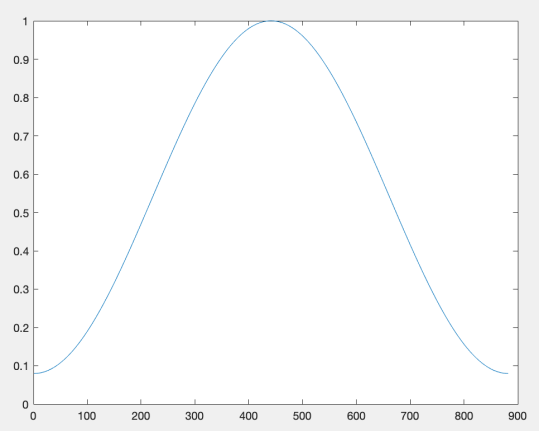
	1	2
58364	3.0518e-...	
58365	3.0518e-...	
58366	3.0518e-...	
58367	3.0518e-...	
58368	0	
58369	0	
58370	0	
58371	0	
58372	0	
58373	0	
58374	0	
58375	0	
58376	0	
58377	0	
58378	0	
58379	0	
58380	0	

y — 43291x1 double

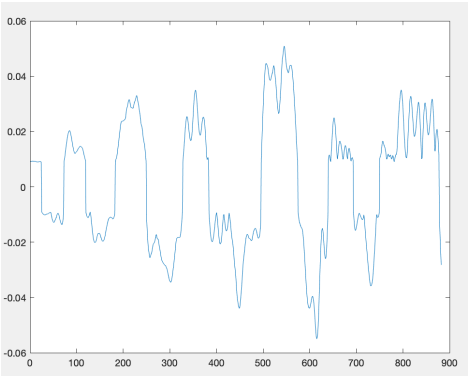
	1	2
43284	0.0122	
43285	0.0117	
43286	0.0114	
43287	0.0108	
43288	0.0103	
43289	0.0100	
43290	0.0094	
43291	0.0090	
43292		
43293		
43294		
43295		
43296		
43297		
43298		
43299		
43300		

- 2) Compute the N-point symmetric Hamming window filter
    - Let Y_len = len(Y) = 43,291
    - Let win_size = 882
    - Make a variable to hold a duplicate of Y to serve as a filter
        - Y' = Y
    - Calculate the number of frames
        - Floor(Y' / win_size) = Floor(43,291 / 882) = 49
        - n_frames = 49
    - Make a column vector of win_size for the filter data
        - W = vector<double>[win_size][1]
    - They used Matlab's hamming(win_size) function to calc data

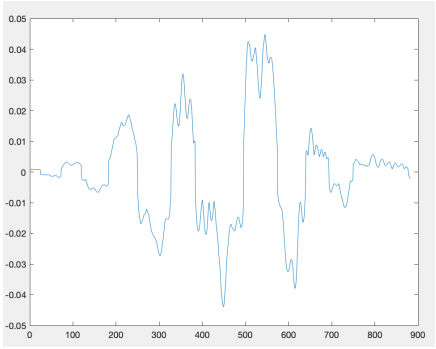
- Zero: Left = [1 .. 42], Right = [841 .. 882]
- One: Left = [43 .. 104], Right = [779 .. 840]
- Two: Left = [105 .. 144], Right = [739 .. 778]
- Three: Left = [145 .. 177], Right = [706 .. 738]
- Four: Left = [178 .. 209], Right = [674 .. 705]
- Five: Left = [210 .. 239], Right = [644 .. 673]
- Six: Left = [240 .. 271], Right = [612 .. 643]
- Seven: Left = [272 .. 305], Right = [578 .. 611]
- Eight: Left = [306 .. 347], Right = [536 .. 577]
- Nine: Left = [248 .. 439], Right = [444 .. 535]
- Ten: Middle = [440 .. 443]



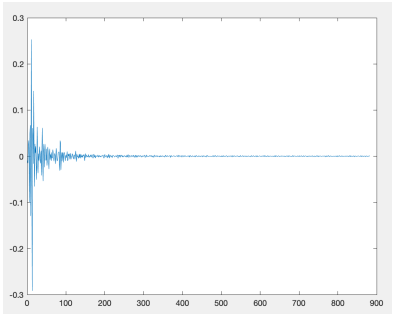
- 3) Apply the Hamming Filter to the normalized sample data
  - For i in range(1,n_frames,1):
    # Example for i = 1
    - Calculate all frame range values
      - t = range(((i - 1) * win_size + 1) : (i * win_size))
      - Computers the following ranges: ([0 .. 882], [883 .. 1,765], [1,766 .. 2,648], …, [42336 .. 43218])
        # Example for t = [0 .. 882]
    - Make a variable to store the results of the filter application
      - Y' = vector<double>[43218][1]
    - Isolate the specific frame interval
      - interval = Y'(t) = Y'([0 .. 882])



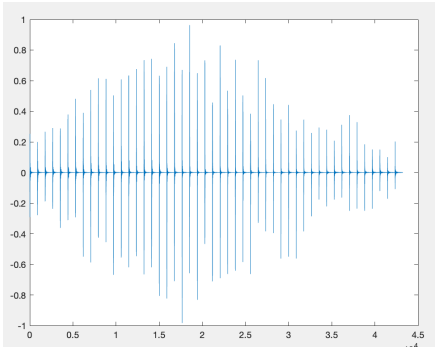
- Calculate the element wise multiplication of interval of normalized sample data and interval of the Hamming Window filter
    - filtered_data = interval .* W
        - filtered_data[0] = interval[0] * W[0]
        - = 0.009094238281250 * 0.080000000000000
        - = 7.275390625000001e-04
        - ...
        - filtered_data[882] = interval[882] * W[882]
        - = 0.028228759765625 * 0.080000000000000
        - = 0.002258300781250



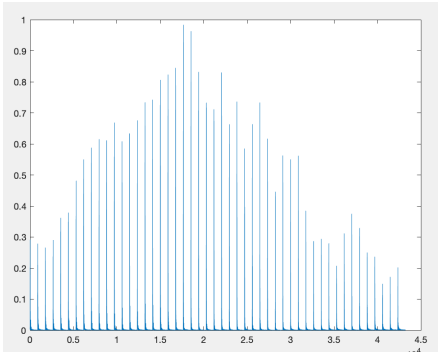
- 4) Calculate the Discrete Cosine Transform of the interval of the filtered data that was normalized
    - Make a variable to hold the results of the fourier filtering in a row vector
        - Fourier = vector<double>[1][43218]
        - Note: the row length of the vector is equal to n_frames * win_size
    - calculate discrete cosine transform of the filter above
        - Fourier(t) = dct(filtered_data)



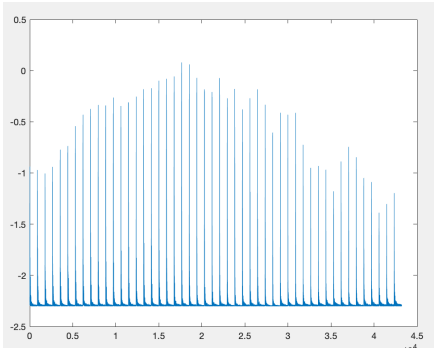
- 5) Calculate the inverse discrete cosine transform
    - inv_transform = idct(log(abs(fourier)+0.1));
    - Start with the completed fourier data



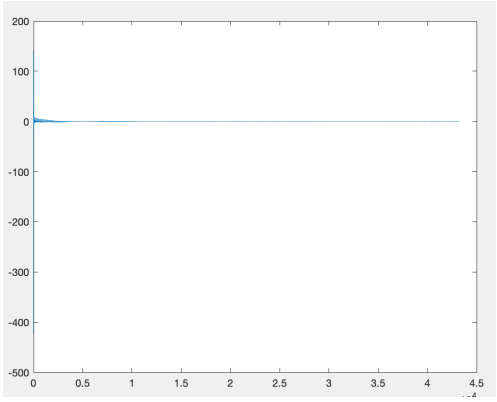
- Take the absolute value of the fourier data



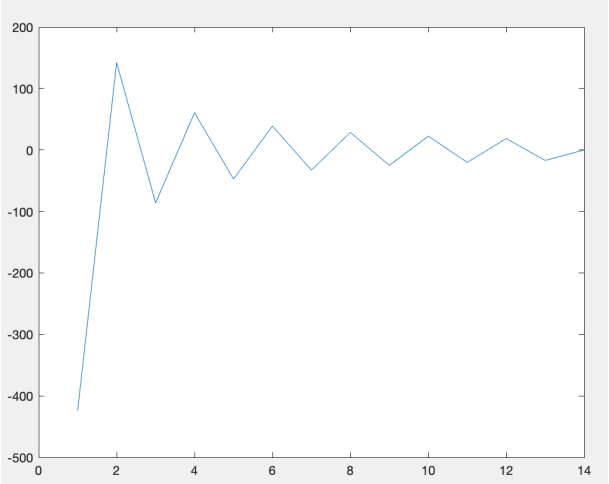
- Take the natural log of the results



- Calculate the idct



- First 13 columns of the MFCC data



- I was able to successfully recreate their work and get similar results. I managed to recreate the above algorithm in Python.
- The example shown above what would happen for a single file. Each file will represent a single row in the mfcc_results.csv.
- https://drive.google.com/drive/folders/1Q7KeeMlNrKtexKcZi_FYn-Mc74NeN1dg
- Each grouping of rows get labeled [1 .. n_instrument_folders] which will be used as the data for Supervised Learning.

2. Calculate the Gaussian SVM

http://ciml.info/dl/v0_99/ciml-v0_99-ch11.pdf

You can also use this property to show that the following **Gaussian kernel** (also called the **RBF kernel**) is also psd:

$$K_\gamma^{(\text{RBF})}(\boldsymbol{x}, \boldsymbol{z}) = \exp\left[-\gamma ||\boldsymbol{x} - \boldsymbol{z}||^2\right] \tag{11.18}$$

Here γ is a hyperparameter that controls the width of this Gaussian-like bumps. To gain an intuition for what the RBF kernel is doing, consider what prediction looks like in the perceptron:

$$f(\boldsymbol{x}) = \sum_n \alpha_n K(\boldsymbol{x}_n, \boldsymbol{x}) + b \tag{11.19}$$

$$= \sum_n \alpha_n \exp\left[-\gamma ||\boldsymbol{x}_n - \boldsymbol{z}||^2\right] \tag{11.20}$$

In this computation, each training example is getting to "vote" on the label of the test point $\boldsymbol{x}$. The amount of "vote" that the nth training example gets is proportional to the negative exponential of the distance between the test point and itself. This is very much like an RBF neural network, in which there is a Gaussian "bump" at each training example, with variance $1/(2\gamma)$, and where the α_ns act as the weights connecting these RBF bumps to the output.

Showing that this kernel is positive definite is a bit of an exercise in analysis (particularly, integration by parts), but otherwise not difficult. Again, the proof is provided in the appendix.

MFCC data -> test_train_split -|-> train_data -> SVM(kernel='rbf') -> model -|-> prediction

|----------> test_dala -^

- Read mfcc_results.csv and split the data into features and labels
    - samples (x_data) = [1 .. 13]
    - labels (y_data) = [14]
- Stratified Shuffle Split the Data into 5 parts with the train data being 87.5% of the total data and the test data being 12.5% of the total data
- Extract the train and test data
- Perform the SVM classification
- SVM:
    - Penalty parameter (C) = 50, Kernel Type (kernel) = ‘rbf’, Kernel Coefficient (gamma) = 0.005, Decision function shape = One-vs-Rest (ovr) = (n_samples, n_classes)
    - The function classifies a portion of the data as specified by the Stratified Shuffle Split.
    - Number of re-shuffling & splitting iterations (n_splits) = 5, test_size=0.125
    - Fit the data
    - predict the results
    - calculate the metrics
- Take the average of all 5 parts of the C-Support Vector Classification to get the average accuracy

I managed to port the algorithm to Python. Check out the colab: https://colab.research.google.com/github/tmartin293/Musical_Instrument_Classification/blob/master/Feature_Extraction/Example/Feature_Extraction.ipynb

My next goals are to get the Gaussian Support Vector Machine working in Tensorflow with Python. Tensorflow has SVMs and a lite version: https://www.tensorflow.org/lite

I am exploring the idea of replacing Python functions for Aubio’s functions.

Hardware:
I was able to power the BBB and the LCD with the power boost 1000 and a Li-Po Battery. I was not successful in connecting the Electret Condenser Mic module from Adafruit. I was successful in recording an audio file using a simple USB mic.

Hardware Updates:
I am currently working to interface and control the 16x2 LCD screen using I2C on the BBB (Circuit Python library unable to select correct I2C bus and device address, so I may need to do this manually).

I was successful in connecting a condenser microphone (Amazon Basics mic) to the BBB and recording an audio file with it. I will focus on taking that audio file and performing the necessary mathematical operations on it to get the MFCC coefficients for instrument classification.

```
root@beaglebone:/# amixer -c 1 contents
numid=2,iface=MIXER,name='Mic Capture Switch'
  ; type=BOOLEAN,access=rw------,values=1
  : values=on
numid=3,iface=MIXER,name='Mic Capture Volume'
  ; type=INTEGER,access=rw---R--,values=1,min=0,max=47,step=0
  : values=40
  | dBminmax-min=-8.00dB,max=15.00dB
numid=1,iface=PCM,name='Capture Channel Map'
  ; type=INTEGER,access=r----R--,values=1,min=0,max=36,step=0
  : values=0
  | container
    | chmap-fixed=MONO

root@beaglebone:/# amixer -c 1 controls
numid=2,iface=MIXER,name='Mic Capture Switch'
numid=3,iface=MIXER,name='Mic Capture Volume'
numid=1,iface=PCM,name='Capture Channel Map'
root@beaglebone:/# arecord -L
null
    Discard all samples (playback) or generate zero samples (capture)
default:CARD=Mic
    AmazonBasics Portable USB Mic, USB Audio
    Default Audio Device
sysdefault:CARD=Mic
    AmazonBasics Portable USB Mic, USB Audio
    Default Audio Device
front:CARD=Mic,DEV=0
    AmazonBasics Portable USB Mic, USB Audio
    Front speakers
```

```
>>> for i in range(audio.get_device_count()):
...     print(audio.get_device_info_by_index(i).get('name'))
...
TI BeagleBone Black: - (hw:0,0)
AmazonBasics Portable USB Mic: Audio (hw:1,0)
sysdefault
default
dmix
```