Unit 2

Speech Recognition

Speech Recognition - Reading audio data - Plotting Audio data - Transforming audio signals into the frequency domain - Apply fourier transform signal & plot - generating audio signals with custom parameters - Generate the time axis - Synthesizing music - Construct audio samples - amplitude and frequency - Extracting frequency domain features - MFCC & filter bank features - Building HMM - HMM training and prediction - Building a speech recognizer - MFCC features - Case Study.

Introduction

→ Speech recognition refers to the process of recognizing & understanding spoken language.

→ Input comes in the form of audio data, and the speech recognizer will process this data to extract meaningful information from it.

→ this has lot of practical issues such as voice controlled devices, transcription of spoken language into words, security systems, and so on.

→ Speech signals are very versatile in nature. There are many variations of speech in same language

→ There are different elements to speech, such as language, emotion, tone, noise, accent and so on.

→ It's difficult to rigidly define a set of rules that can constitute speech.

⇒ Famous tasks: Automatic speech recognition, has been the focal point of attention for many researchers.

Reading & plotting audio data:

* Let's see how to read an audio file and visualize the signal,

→ create a new python file, import following packages

1.
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

2. # Read the i/p file

Sampling_freq, audio = wavfile.read('input_read.wav')

3. Parameters of this signal (print)

⇒
```
print 'Shape:', audio.shape
print 'Datatype:', audio.dtype
```

print 'Duration:', round(audio.shape[0] / float(sampling_freq), 3, 'second'.

4. Store audio signal as 16-bit signed integer data and normalize the values:

```
#Normalize the values
audio = audio/(2.**15)
```

5. Extract the first 30 values to plot, as follows:

```
#Extract the first 30 vals to plot
audio = audio[:30]
```

6. The x-axis is the time axis.

```
#Build axis.
x_values = np.arange(0, len(audio), 1) / float
(sampling_freq)
```

7. Convert the units to seconds:

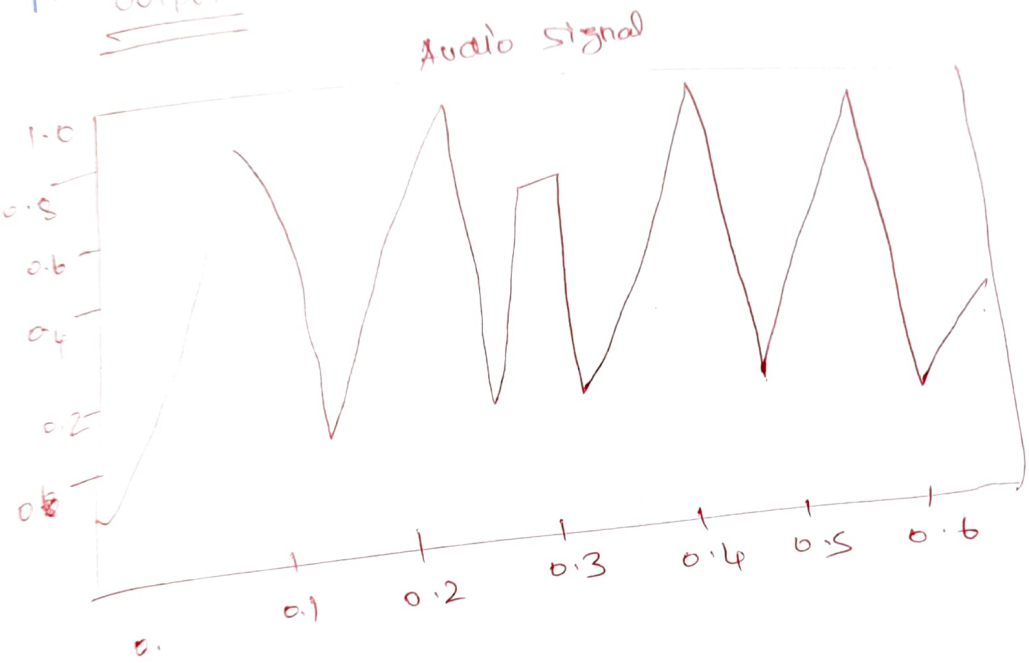
```
#conver to seconds
x_values *= 1000
```

8. Let's plot this as follows.

```
#Plotting the chopped audio signal.
plt.plot(x_values, audio, color = 'black')
```

```
plt.xlabel('Time(ms)')
plt.ylabel('Amplitude')
plt.title('Audio signal')
plt.show().
```

eg. OUTPUT:



Shape: (132300,)

Datatype: int16.

Duration: 3.0 seconds

(2) Transforming audio signal into the freq. domain.

→ Audio signal consist of a complex mixture of the waves of different frequencies, amplitudes, and phases.

→ sine waves are referred to as sinusoids.

→ so, there is a lot of information that is hidden in

on the frequency content of an audio signal.

→ The whole world of speech and music is based on this fact. Before that we need some knowledge on fourier transforms.

Implementation:

(1)
```
import numpy as np
from scipy.io import wavfile
import matplotlib.pyplot as plt
```

(2) Read the input_freq.wav

```
# Read i/p file.
Sample_freq, audio = wavfile.read('input_freq.wav')
```

(3) Normalize the signal, as follows:

```
# Normalize the values
audio = audio / (2.**15)
```

(4) the audio signal is just a numpy array. So, you can extract the length using the following code:

```
# Extract length
len_audio = len(audio)
```

(5) # Apply Fariey transform

```
transformed_signal = np.fft.fft(audio)
half_length = np.ceil((len_audio+1)/2.0)
transformed_signal = abs(transformed_signal[0:half
                                            _length])
transformed_signal /= float(len_audio)
transformed_signal **= 2.
```

(6) Extract the length of the signal:

```
# Take care of even/odd cases
-> len_ts = len(transformed_signal)
if len_audio % 2:
    transformed_signal[1:len_ts] *= 2
else
    transformed_signal[1:len_ts-1] *= 2.
```

(7) # Extract power in dB.

```
Power = 10 * np.log10(transformed_signal)
```

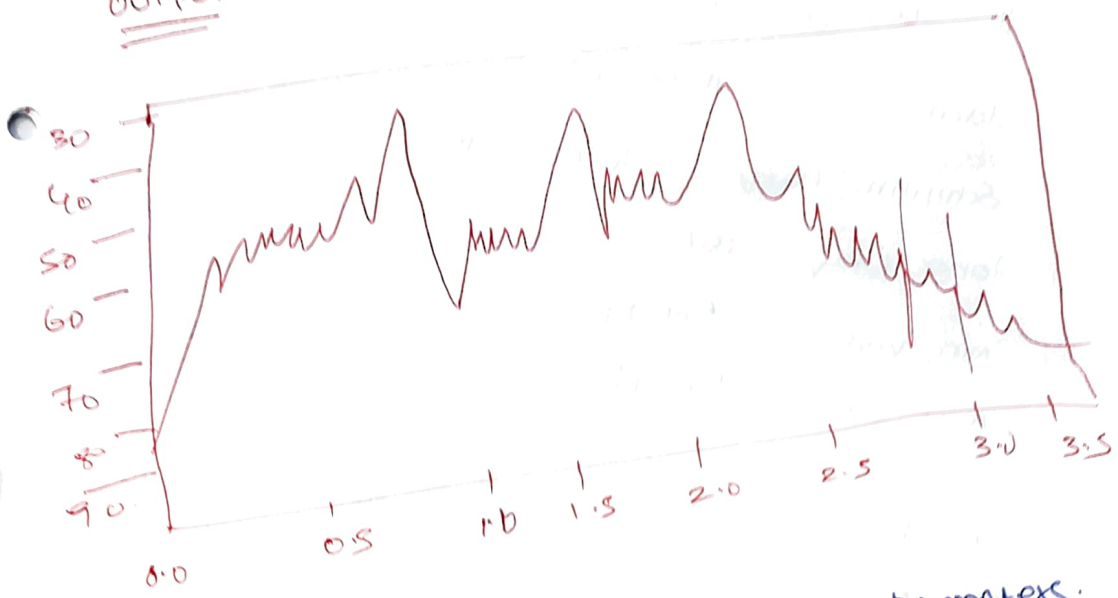
(8) # Build the time axis:

```
X_values = np.arange(0, half_length, 1) * (sampling_freq/
                                len_audio)/1000.0.
```

(9) Plot the signal as follows:-

```
plt.figure()
plt.plot(x_values, powers, color='black')
plt.xlabel('Freq (in kHz)')
plt.ylabel('Power (in dB)')
plt.show()
```

output:



(3) Generating Audio Signal with custom parameters:

→ We can use Numpy to generate audio signals. As we discussed earlier, audio signals are complex mixtures of sinusoids. So keep this in mind when we want to generate our own signal.

required packages

(1)
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write
```

(2) #File where the output will be saved

output_file = 'output_generated.wav'

(3) Generate a 3-second long signal with a sampling frequency of 44100 and a total frequency of 587 Hz. The values on the time axis will go from $-2*pi$ to $2*pi$.

```
# Specify audio parameters
duration = 3 # seconds
sampling_freq = 44100 #Hz
tone_freq = 587
min_val = -2 * np.pi
max_val = 2 * np.pi
```

(4) Lets generate the time axis and the audio signal. The audio signal is a simple sinusoid with the previously mentioned parameters:

```
#Generate audio
t = np.linspace (min_val, max_val, duration
                 * sampling_freq )
audio = np.sin (2 * np.pi * tone_freq * t)
```

(5) Add some noise to the signal:

```
noise = 0.4 * np.random.rand (duration * sampling_freq)
audio += noise.
```

(6) Scale the values to 16-bit integers before we store them:

```
#Scale it to 16-bit integer values
Scaling_factor = pow(2,15) - 1
audio_normalized = audio / np.max(np.abs(audio))
audio_scaled = np.int16(audio_normalized * Scaling_factor)
```

(7) write signal to the o/p file:

```
write(output_file, Sampling_freq, audio_scaled)
```

(8) Plot signals using the first 100 values:-

```
audio = audio[:100]
```

(9) Generate the time axis:

```
#Build the time axis
X_values = np.arrange(0, len(audio),1) / float
(Sampling_freq)
```

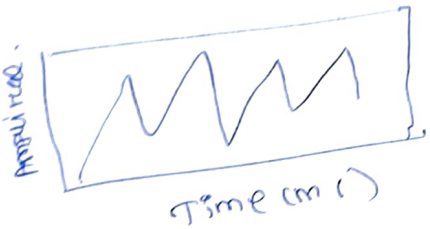
(10) Convert the time axis into seconds.

```
#conver to seconds
X_values *= 1000
```

(11) Plot the signal as follows,

```
plt.plot (y_values, audio, color = 'black')
plt.xlabel ('Time (ms)')
plt.ylabel ('Amplitude')
plt.title ('Audio signal')
plt.show()
```

12) Output:



14. Synthesizing music.

→ # Generate simple music.

(1) → `import json`
`import numpy as np`
`from scipy.io.wavfile import write`
`import matplotlib.pyplot as plt`

(2)
```
# Synthesize tone
def synthesize(freq, duration, amp = 1.0, sampling_
                                    rate = 44100):
```

(3) Build the time axis value:

```
# Build the time axis
t = np.linspace(0, duration, duration * sampling_freq)
```

4 Define main function:-

import notes:-

```
if __name__ == '__main__':
    tone_map_file = 'tone_freq_map.json'
```

5 Load the file:

```
#Read the frequency map
with open(tone_map_file, 'r') as f:
    tone_freq_map = json.loads(f.read())
```

6 Let's assume that we want to generate a G note for a duration of 2 seconds:-

```
#set input parameters to generate 'G' tone
input_tone = 'G'
duration = 2    #seconds
amplitude = 10000
sampling_freq = 44100 #Hz
```

7 Call function with following parameters.

```
#Generate the tone
synthesized_tone = synthesizes(tone_freq_
                    map[input_tone],
duration, amplitude, sampling_freq)
```

(8) Write the generated signal into the output file:

```
# Write the o/p file.
write('output_tone.wav', sampling_freq,
                         synthesized_tone)
```

(9) Open this file in a media player and listen to it. There's the G note! Let's do something more interesty. Let's generate some notes in sequence to give it a musical feel. Define a note sequence along with their duration in seconds

```
# Tone-duration sequence
tone_seq = [('D', 0.3), ('G', 0.6), ('C', 0.5), ('A', 0.3),
            ('Asharp', 0.7)]
```

(10) Iterate through the list and call the synthesizer function for each of them:

```
# Construct the audio signal based on the chord
sequence
output = np.array([])
for item in tone_seq:
    input_tone = item[0]
    duration = item[1]
    synthesized_tone
    = synthesize(tone_freq_map[input_tone],
                 duration, amplitude, sampling_freq)
output = np.append(output, synthesized_tone, axis=0)
```

write the signal to o/p file:

#write to the o/p file.

write ('output_tone_seq.wav', sampling_freq, output)

(5) Extracting frequency domain features.

-> After converting a signal into the frequency domain, you need to convert it into a usable form. MFCC is a good way to do this. MFCC takes the power spectrum of a signal & then uses a combination of filter banks, and discrete cosine transformation to extract features.

Code:-

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.import wavfile
from features import mfcc, logfbank
```

-> Read the input_freq.wav input file that is already provided:

```
sampling_freq, audio = wavfile.read('input_freq.wav')
```

->
```
# Extract MFCC & Filter bank features
mfcc_features = mfcc(audio, sampling_freq)
filter_bank_features = logfbank(audio, sampling_freq)
```

-> Check the number of windows generated:

```
#print parameters
print('\nMFCC: \n Number of windows =', mfcc_features.
                                        shape[0])
print('Length of each feature =', mfcc_features.shape[1])
print('\n Filter bank: \n Number of windows =',
filterbank_features.shape[0])
print('Length of each feature =', filterbank_features.shape[1])
```

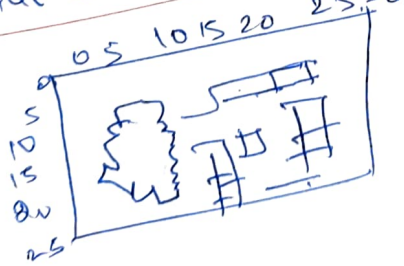
-> Visualize MFCC features.

```
#Transform the matrix so that the time domain is
horizontal:
#plot the features
mfcc_features = mfcc_features.T
plt.matshow(mfcc_features)
plt.title('MFCC')
```

-> Visualize the filter bank features

```
filterbank_features = filterbank_features.T
plt.matshow(filterbank_features)
plt.title('Filter bank')
plt.show()
```

-> O/P:



MFCC:
Number of windows = 40
Length of each feature = 13

Filter bank:
Number of windows = 40
Length of each feature = 26.

5 Building Hidden markov models

→ HMMs detecting suits our needs. An HMM is a model that represents probability distributions over sequences of observations. We assume that the outputs are generated by hidden states. Goal: Find the hidden states so that we can model the signal.

Code:

1. # class to handle all HMM related processing.
   class HMMTraining (object).

2. Initialize the class. HMMs are used to model the data, the n_components parameter defines the number of hidden states. the cov_type defines the type of covariance in our transition matrix, and (n_iter) indicates the number of iterations it will go through before it stops training.

```
def __init__(self, model_name='GaussianHMM',
             n_components=4, cov_type='diag', n_iter=10):
```

3. Initialize the variables,

```
self.model_name = model_name
self.n_components = n_components
self.cov_type = cov_type
self.n_iter = n_iter
self.models = []
```

4. Define the model with tailored parameters

```
if self.model_name == 'GaussianHMM':
```

```
self.model = hmm.GaussianHMM(n_components = self.n_
                             covariance_type = self.
                                 cov_type,
                             n_iter = self.n_iter)
else:
    raise TypeError('Invalid model type')
```

5. the input data is a numpy array, where each element is a feature vector consisting of k-dimensions:

```
# X is a 2D numpy array where each row is
13D def train(self, X):
    np.seterr(all='ignore')
    self.models.append(self.model.fit(X))
```

6. Define a method to extract the score, based on the model.

```
# Run the model on input data
def get_score(self, input_data):
    return self.model.score(input_data)
```

7. we built a class to handle HMM training and prediction, but we need some data to see it in action. we will use it in the next recipe to build a speech recognizer.

-> Building a Speech Recognizer:

-> A database containing 7 different words, where each word has 15 audio files associated with it. This is a small dataset, but this sufficient to understand how to build a speech recognizer that can recognize seven different words. We need to build an HMM model for each class.

-> import os
import argparse
import numpy as np
from scipy.io import wavfile
from hmmlearn import hmm
from utilities import main

(2) # Parse the input arguments in the command line:

# Function to parse i/p arguments
def build_arg_parser():
parser = argparse.ArgumentParser(description='Trains the HMM classifier')
parser.add_argument('--input-folder', dest='input_folder', required=True,
help="Input folder containing the audio files in subfolders")
return parser

(3) Define the main function, parse the i/p arguments:

if __name__=='__main__':
args = build_arg_parser().parse_args()
input_folder = args.input_folder

(4) Initiate the variables that will hold all the HMM models:
hmm_models = []

(5) # Parse the i/p directory
for dirname in os.listdir(input_folder):

(6) Extract the name of the subfolder:
# Get the name of the subfolder
subfolder = os.path.join(input_folder, dirname)
if not os.path.isdir(subfolder):
continue

(7) The name of the subfolder is the label of this class.

Extract the label

label = subfolder[subfolder.find('/')+1:]

(8) Initialize the variables for training:

Initialize variables

X = np.array([])

y_words = []

(9) Iterate through the list of audio files in each subfolder.

Iterate through the audio files

for filename in [x for x in os.listdir(subfolder) if x.endswith('.wav')][:-1]:

(10) Read the each audio file, as follows:

Read the i/p file.

filepath = os.path.join(subfolder, filename)

sampling_freq, audio = wavfile.read(filepath)

(11) Extract the MFCC features.

mfcc_features = mfcc(audio, sampling_freq)

(12) Keep appending this to the X variable:

Append to the variable X.

if len(X) == 0:

X = mfcc_features

else:

X = np.append(X, mfcc_features, axis=0)

(13) Append the corresponding label too:

Append the label

y_words.append(label)

(14) # Train and save HMM model.

hmm_trainer = HMMTrainer()

hmm_trainer.train(X)

hmm_models.append((hmm_trainer, label))

hmm_trainer = None

(15) Get a file to test.

Test files

input_files = ['data/...', 'data/...']

(16) parse the inputs:

```
#classify input data
for input_file in input_files
```

(17) Read in each audio file:

```
#Read input file
sampling_freq, audio = wavfile.read(input_
```

(18) Read in each audio file:

```
sampling_freq, audio = wavfile.read(input_file)
```

(19)

```
Extract the MFCC features:
#Extract MFCC feats
mfcc_features = mfcc(audio, sampling_freq)
```

(20)

```
#max_score
max_score = None
output_label = None
```

(21)

```
#Iterate through all HMM models and pick
The one with highest score

for item in hmm_models:
    hmm_model, label = item
```

(22)

```
Extract the score & store max.score
score = hmm_model.get_score(mfcc_features)
if score > max_score:
```

```
MAX_score = score
output_label = label.
```

(23) Print true & predicted labels:

```
# print output
print "\n True:",
input_bill [input_bill.find('/') + 1 : input_
                                    bill.rfind('/')]
print "predicted", output_label
```

(24) Output

```
True: pineapple
predicted: pineapple

True: orange
predicted: orange

True: apple
predicted: apple.
```