SPEECH RECOGNITION

Rajalakshmi Santhakumar
DS-SEA 08

INTRODUCTION

Automatic speech recognition is the technology that allows human beings to use their voices to speak with a computer interface and is currently invading our lives. It's built into our phones, our game consoles and our smart watches. It's even automating our homes. Though the ASR technology is getting better day-by-day in terms of technological development, it is still a challenging task due to the high variations in the speech signals with respect to the accent, voice modulation and background noises.

Project Question:

The current model aims to transcribe, with accuracy, the spoken words (in English) to text by pre-processing and modeling audio files.

DESCRIPTION OF DATA SET

The data for this model was obtained from the Google Speech Commands Dataset. This is a set of one-second .wav audio files, each containing a single spoken English word. These words are from a small set of commands, collected using crowdsourcing and are spoken by a variety of different speakers. Twenty core command words were recorded, with most speakers saying each of them five times. The core words are "Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", "Go", "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", and "Nine". To help distinguish unrecognized words, there are also ten auxiliary words, which most speakers only said once. These include "Bed", "Bird", "Cat", "Dog", "Happy", "House", "Marvin", "Sheila", "Tree", and "Wow". The audio files are organized into folders based on the word they contain, and no details were recorded for any of the participants and random ids were assigned to each individual. These ids are encoded in each file name as the first part before the underscore. If a participant contributed multiple utterances of the same word, these are distinguished by the number at the end of the file name.

DATA PREPROCESSING AND VISUALIZATION

Sound is transmitted as waves and are one-dimensional i.e. at every moment in time, they have a single value based on the height of the wave. To turn this sound wave into meaningful numbers that can be used for Data modelling, we need to preprocess the audio files. I explored 3 different preprocessing measures for processing audio files, which are described below:

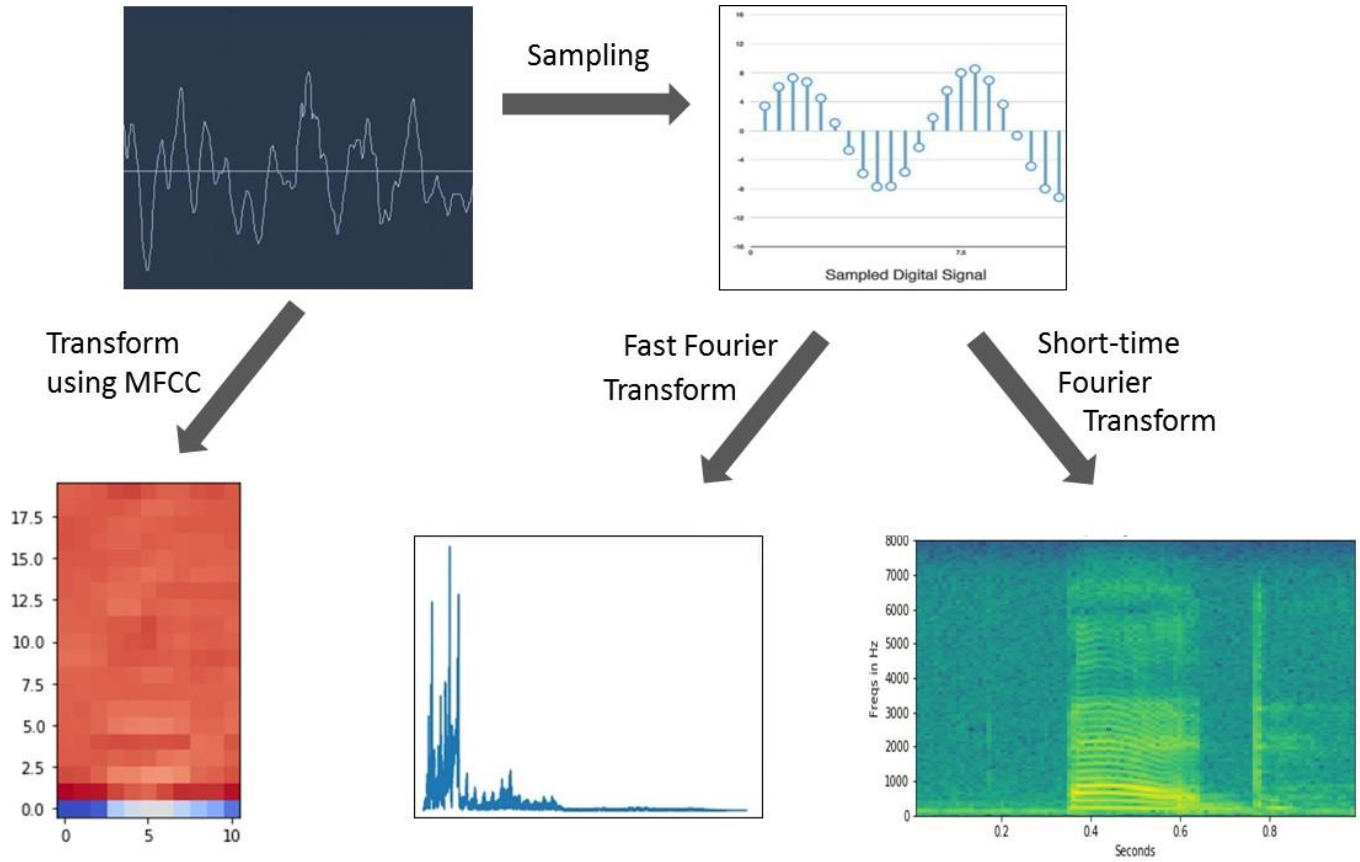


Fig.1: Data Preprocessing

1. SAMPLING AND FAST FOURIER TRANSFORMATION:

In signal processing, sampling is the reduction of a continuous-time signal to a discrete-time signal. In order to transform an audio wave to a sequence of samples, we record of the height of the wave at equally-spaced points, in this case every 1/16,000th second. However, this data is complex mix of different frequencies of sound and therefore, processing these samples directly by neural networks might be difficult and more time consuming. Fast Fourier transformation breaks apart this complex sound wave into its component parts and provides a score that represents how much energy was in each 50 Hz band of our audio clip.

In the current dataset, each audio file was sampled using 'wavfile.read' and was fast-Fourier transformed using 'fft' function from the 'scipy' package. Sampling generates 16,000 records for each audio file which becomes 8000 after Fourier transformation since FFT is symmetrical. This corresponds to the energy in each 50 Hz band and is stored as a numpy array.

```
# Importing all the required packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy
import scipy.io.wavfile
import os
from scipy.fftpack import fft
import scipy.signal as signal
import keras
import tensorflow
import librosa
import sklearn
```

```
# Defining a function to calculate the fast-Fourier transform
def custom_fft(data, rate):
    S = 1.0 / rate
    N = data.shape[0]
    yf = fft(data)
    xf = np.linspace(0.0, 1.0/(2.0*S), N//2)
   # FFT results in a complex number and we need just the real part (abs)
    value = 2.0/N * np.abs(yf[0:N//2])  # FFT is symmetrical, so we can just take the first half
    return xf, value
```

```
# Performing sampling and fast-Fourier transformation on one of the audio files from 'bed'
rate, data =
scipy.io.wavfile.read(r'C:\Users\srajalak\Documents\GA\Project\speechData1\bed\00f0204f_nohash_0.w
av')
xf, value=custom_fft(data, rate)
fft_data_sample= pd.DataFrame (value, xf)
fft_data_sample.head(10)
```

	0
0.000000	0.171375
1.000125	0.082871
2.000250	0.252714
3.000375	0.092358
4.000500	0.145415
5.000625	0.131781
6.000750	0.229318
7.000875	0.289879
8.001000	0.133706
9.001125	0.095457

Fig.2: Preprocessed Data obtained after sampling and fast Fourier transformation for the audio file from 'bed'

```
# Plotting the data
fft_data_sample[[0]].plot()
plt.show()
```

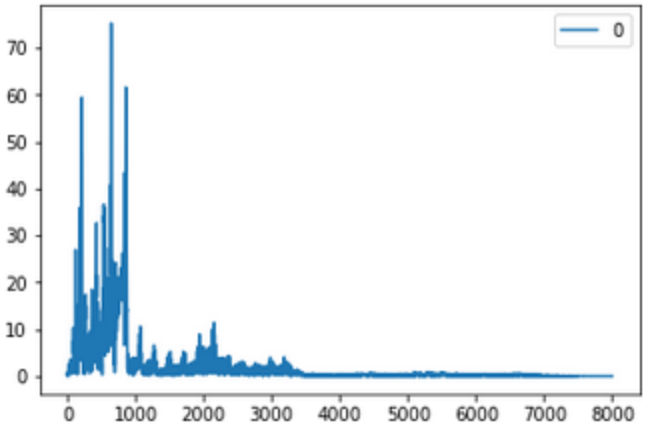


Fig.3: Plot of the fast Fourier transformation for the audio file from 'bed'

```
# Plotting the data after reducing the dimensions
fft_data_sample_new100 = ((fft_data_sample + fft_data_sample.shift(-1)) / 2)[::100]
fft_data_sample_new100[[0]].plot()
plt.show()
fft_data_sample_new500 = ((fft_data_sample + fft_data_sample.shift(-1)) / 2)[::500]
fft_data_sample_new500[[0]].plot()
plt.show()
```

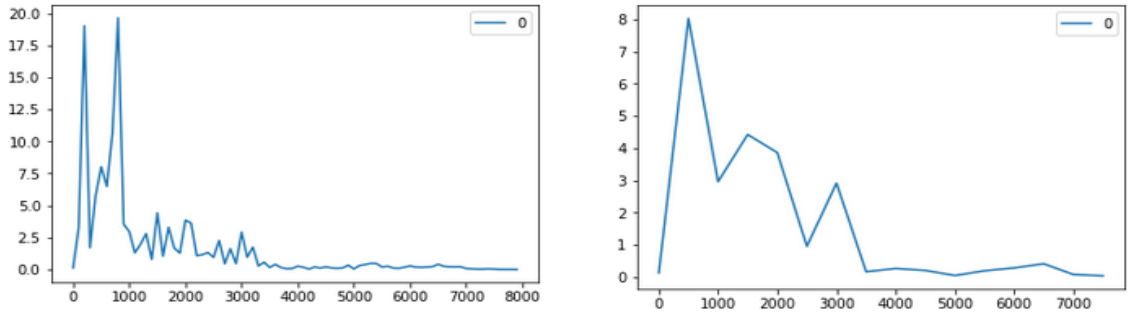


Fig.3: Plot of the fast Fourier transformation for the audio file from 'bed' after reducing the dimensions by 'mean substitution'

From the above plots, it was clear that reducing the dimension resulted in a not very good representation of the actual audio file and therefore, was not used for further studies.

2. SAMPLING AND SHORT-TIME FOURIER TRANSFORM

Spectrograms are visual representations of the spectrum of frequencies of sound as a variation at different time intervals. These are generally obtained by using short-time Fourier transformation of the digitally sampled signals. In the current dataset, each audio file was sampled again using 'wavfile.read' and was short-time Fourier transformed using 'spectrogram' function from the 'scipy' package which generates the 'Spectrogram'. This preprocessing generates a 99 x 160 matrix for each audio file where '99' corresponds to the time intervals and '160' corresponds to different frequencies from 0 to 8000 in the intervals of 50.

```
# Defining a function to calculate the short-time Fourier transform i.e. Spectrogram

def log_specgram(audio, sample_rate, window_size=20,
            step_size=10, eps=1e-10):
   nperseg = int(round(window_size * sample_rate / 1e3))
   noverlap = int(round(step_size * sample_rate / 1e3))
   freqs, times, spec = signal.spectrogram(audio,
                           fs=sample_rate,
                           window='hann',
                           nperseg=nperseg,
                           noverlap=noverlap,
                           detrend=False)
   return freqs, times, np.log(spec.T.astype(np.float32) + eps)
```

```
# Performing sampling and short-time Fourier transformation on one of the audio files from 'bed'

rate, data =
scipy.io.wavfile.read(r'C:\Users\srajalak\Documents\GA\Project\speechData1\bed\00f0204f_nohash_0.w
av')
freqs, times, spectrogram =log_specgram(data, rate)
spec_data_sample= pd.DataFrame(spectrogram, times, columns = [freqs])
spec_data_sample.head(10)
```

	0.0	50.0	100.0	150.0	200.0	250.0	...	7750.0	7800.0	7850.0	7900.0	7950.0	8000.0
0.01	0.223507	-0.093135	1.790785	2.133407	-0.468045	-2.745347	...	-12.222241	-10.010121	-11.170351	-13.303512	-11.002518	-12.126349
0.02	-0.866040	-1.524011	2.449461	2.301860	-2.025680	-2.411648	...	-10.538596	-12.126993	-12.323122	-13.373431	-10.848579	-10.987273
0.03	0.088954	-2.494596	2.050632	2.214396	-0.499036	-1.449178	...	-10.705381	-11.313753	-11.441858	-12.760565	-11.025206	-11.192350
0.04	-1.089610	-0.617435	1.802585	1.578745	-3.544367	-0.647245	...	-11.482220	-11.767186	-10.931141	-11.433665	-15.718157	-12.914542
0.05	-3.395147	-0.404876	1.471095	1.181125	-0.960714	-2.048784	...	-11.373401	-13.656361	-11.758223	-12.069757	-10.741830	-11.636673
0.06	-2.692643	-2.479739	-0.003002	-0.871403	-3.117157	-1.302667	...	-10.073637	-10.118586	-14.003018	-11.356189	-11.044828	-10.720243
0.07	-3.343531	-0.826792	1.281096	0.816372	-2.822185	-3.752811	...	-10.056070	-11.357976	-10.439246	-10.628709	-12.575134	-11.656739
0.08	-1.935644	-0.978663	2.195481	1.962869	-1.633644	-0.936466	...	-12.393388	-13.709919	-14.103498	-11.321643	-10.414289	-11.490307
0.09	-4.140809	-0.331942	2.231247	2.101403	-1.214366	-2.472553	...	-10.993739	-10.253239	-9.822909	-11.756866	-11.490239	-11.693313
0.10	-3.943367	-0.122460	1.845573	1.607924	-1.650186	-0.962599	...	-9.242753	-10.603787	-13.268634	-11.443624	-11.760928	-13.992090

Fig.5: Preprocessed Data obtained after sampling and short-time Fourier transformation for the word 'bed'

```
#Plotting the data
fig = plt.figure(figsize=(8, 8))
ax1 = fig.add_subplot(212)
ax1.imshow(spectrogram.T, aspect='auto', origin='lower',
        extent=[times.min(), times.max(), freqs.min(), freqs.max()])
ax1.set_ylabel('Freqs in Hz')
ax1.set_xlabel('Seconds')
plt.show()
```

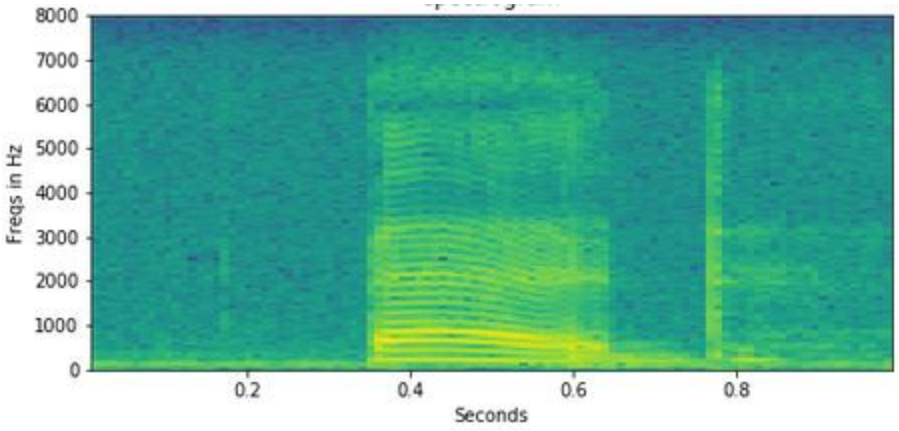


Fig.6: Spectrogram for the audio file from 'bed'

3. MFCC TRANSFORMATION

Mel-frequency cepstral coefficients (MFCC) mimics the logarithmic perception of loudness and pitch of human auditory system and tries to eliminate speaker dependent characteristics by excluding the fundamental frequency and their harmonics. MFCCs are commonly derived by taking the Fourier transform of (a windowed excerpt of) a signal which is then, mapped to the powers of the spectrum onto the mel scale (a perceptual scale of pitches judged by listeners to be equal in distance from one another). This is followed by taking the logs of the powers at each of the mel frequencies and then, the discrete cosine transform of the list of mel log powers were taken, as if it were a signal. The amplitudes of the resulting spectrum represent the MFC coefficients.

In the current dataset, each audio file was transformed into their respective MFC coefficients using the 'mfcc' from the 'librosa' package. This preprocessing generates MFC coefficients in an 20 x 11 matrix for each audio file. The matrices corresponding to the same word (for instance, 'bed') are appended together and stored in a numpy file labeled with the respective word (in this case, 'bed.npy').

```
# Defining a function to calculate MFCC (Output = 20 x 11 matrix)
def calc_mfcc(wave_file, max_pad_len=11):
    data, rate = librosa.load(wave_file, mono=True, rate=None)
    data = data[::3]
    mfcc = librosa.feature.mfcc(data, rate=16000)
    pad_width = max_pad_len - mfcc.shape[1]
    mfcc = np.pad(mfcc, pad_width=((0, 0), (0, pad_width)), mode='constant')
    return mfcc
```

```
# Performing MFCC transformation on one of the audio files from 'bed'

mfcc_data_sample=
pd.DataFrame(calc_mfcc(r'C:\Users\srajalak\Documents\GA\Project\speechData1\bed\00f0204f_nohash
_0.wav', max_pad_len = 11))

mfcc_data_sample
```

	0	1	2	3	4	5	6	7	8	9	10
0	-567.610739	-566.760655	-542.897907	-341.840983	-277.175192	-250.473469	-248.879873	-314.132993	-371.569909	-420.091146	-508.954545
1	80.288451	72.389148	68.361175	29.455025	3.206865	-9.165578	23.323532	52.528145	47.413732	48.150497	70.896217
2	23.906982	19.122831	-4.998897	-47.252404	-50.993598	-69.590654	-72.454044	-57.501834	-10.878171	-1.580431	15.790708
3	3.989030	3.819819	5.806693	-6.515886	-0.686350	-5.709155	-23.833099	-35.215653	-23.256402	-20.559667	-1.857126
4	1.096233	2.640310	14.501615	21.143898	17.257950	12.128402	14.772363	18.827896	-24.045490	-20.617113	9.489418
5	0.437399	0.859556	-10.701122	-36.017015	-41.985822	-48.210340	-45.236051	-29.507618	-10.212152	-9.983163	-18.439216
6	-1.001116	3.108822	-9.103020	-23.128434	-21.101869	-21.690407	-20.233770	-22.500295	-4.449128	-1.906624	-13.452097
7	-1.788863	-3.580208	-2.537597	-1.246591	1.677696	-1.137205	-2.625123	-5.584562	-6.287697	-7.866723	-8.232969
8	-6.402041	-6.771073	-2.909410	0.943160	-4.911938	-16.666377	-16.270129	-4.539594	4.164030	-1.564259	-12.636876
9	-18.443931	-19.128120	-13.247619	7.557161	11.933975	9.190003	1.966793	-1.146538	11.954632	14.913280	8.311287
10	-11.671178	-13.726727	-7.815776	4.900387	15.626864	21.324169	-0.795268	-12.385246	-2.628645	4.142022	-2.870926
11	-6.134264	-7.428605	-1.095490	4.431319	6.573571	22.368662	1.943474	-2.062086	2.052042	4.356875	-8.782899
12	-3.733292	-0.029216	-6.722374	-27.808852	-25.786974	-2.678465	-6.722053	-12.804454	-3.757381	-5.220060	-1.705821
13	4.076621	0.170590	-10.249841	-17.731900	-18.183042	-9.568578	6.641629	8.727845	6.039863	7.909135	11.401873
14	2.190991	0.881304	4.494207	-8.078573	-13.273857	-25.727245	-16.700492	-4.829244	-13.924952	-11.007816	-8.436059
15	5.261690	8.689530	9.341943	2.809810	2.165692	-6.976098	-14.581117	-5.854710	-2.622434	-4.221603	-5.490754
16	15.128092	11.587232	-0.427477	-7.948558	-6.196346	4.024162	-2.455359	-7.066185	-5.226491	-4.852498	4.866396
17	13.410044	11.651258	4.402123	-6.111028	-9.573207	-5.626544	0.069436	4.568971	4.773977	6.413750	7.319140
18	1.180168	2.400893	8.728827	8.786773	1.452824	-16.848961	-17.690401	-8.194760	0.784295	6.753733	6.729339
19	-4.035184	-0.262653	7.679917	26.168339	29.877308	13.180935	-1.054622	-3.184731	15.601466	20.158641	8.417351

Fig.7: Preprocessed Data obtained after MFCC transformation (20 x 11) for the word 'bed'

```
# Plotting the data
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm
fig, ax = plt.subplots()
mfcc_data= np.swapaxes(mfcc_data_sample, 0 ,1)
cax = ax.imshow(mfcc_data_sample, interpolation='nearest', cmap=cm.coolwarm, origin='lower')
ax.set_title('MFCC')

plt.show()
```

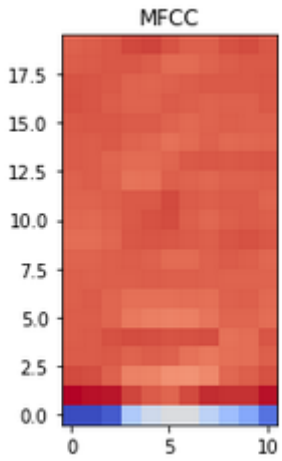


Fig.8: Plot of the MFCC transformation for the audio file from ‘bed’

This preprocessing generates MFC coefficients in an 13 x 11 matrix for each audio file. The matrices corresponding to the same word (for instance, ‘bed’) are appended together and stored in a numpy file labeled with the respective word (in this case, ‘bed.npy’).

```
# Defining a function to calculate MFCC (Output = 13 x 11 matrix)
def calc_mfcc13(wave_file, max_pad_len=11):
    data, rate = librosa.load(wave_file, mono=True, rate=None)
    data = data[::3]
    mfcc = librosa.feature.mfcc(data, rate=16000, n_mfcc=13)
    pad_width = max_pad_len - mfcc.shape[1]
    mfcc = np.pad(mfcc, pad_width=((0, 0), (0, pad_width)), mode='constant')
    return mfcc
```

```
# Performing MFCC transformation on one of the audio files from 'bed'

mfcc_data_sample=
pd.DataFrame(calc_mfcc13(r'C:\Users\srajalak\Documents\GA\Project\speechData1\bed\00f0204f_nohash_0.wav', max_pad_len = 11))

mfcc_data_sample
```

	0	1	2	3	4	5	6	7	8	9	10
0	-567.610739	-566.760655	-542.897907	-341.840983	-277.175192	-250.473469	-248.879873	-314.132993	-371.569909	-420.091146	-508.954545
1	80.288451	72.389148	68.361175	29.455025	3.206865	-9.165578	23.323532	52.528145	47.413732	48.150497	70.896217
2	23.906982	19.122831	-4.998897	-47.252404	-50.993598	-69.590654	-72.454044	-57.501834	-10.878171	-1.580431	15.790708
3	3.989030	3.819819	5.806693	-6.515886	-0.686350	-5.709155	-23.833099	-35.215653	-23.256402	-20.559667	-1.857126
4	1.096233	2.640310	14.501615	21.143898	17.257950	12.128402	14.772363	18.827896	-24.045490	-20.617113	9.489418
5	0.437399	0.859556	-10.701122	-36.017015	-41.985822	-48.210340	-45.236051	-29.507618	-10.212152	-9.983163	-18.439216
6	-1.001116	3.108822	-9.103020	-23.128434	-21.101869	-21.690407	-20.233770	-22.500295	-4.449128	-1.906624	-13.452097
7	-1.788863	-3.580208	-2.537597	-1.246591	1.677696	-1.137205	-2.625123	-5.584562	-6.287697	-7.866723	-8.232969
8	-6.402041	-6.771073	-2.909410	0.943160	-4.911938	-16.666377	-16.270129	-4.539594	4.164030	-1.564259	-12.636876
9	-18.443931	-19.128120	-13.247619	7.557161	11.933975	9.190003	1.966793	-1.146538	11.954632	14.913280	8.311287
10	-11.671178	-13.726727	-7.815776	4.900387	15.626864	21.324169	-0.795268	-12.385246	-2.628645	4.142022	-2.870926
11	-6.134264	-7.428605	-1.095490	4.431319	6.573571	22.368662	1.943474	-2.062086	2.052042	4.356875	-8.782899
12	-3.733292	-0.029216	-6.722374	-27.808852	-25.786974	-2.678465	-6.722053	-12.804454	-3.757381	-5.220060	-1.705821

Fig.9: Preprocessed Data obtained after MFCC transformation (20 x 11) for the word 'bed'

Comparison of the data preprocessing techniques:

	Features	**Sampling And Fast Fourier Transformation**	**Sampling And Short-Time Fourier Transform**	**MFCC Transformation**
1.	Size of preprocessed data for each audio file	1 x 16000 1 x 8000	99 x 160	11 x 20 11 x 13
2.	Time taken for processing 1000 files	~12 minutes	~4-5 minutes	<1 minute
3.	Extent of data preprocessing (as seen by neural network models)	Moderately preprocessed	Moderately preprocessed	Highly preprocessed

Based on the above comparison, MFCC transformation was used for all the further data modeling and analysis.

```
import numpy as np
dir = r'C:\Users\srajalak\Documents\GA\Project\speechData1'

# Getting the labels of the folder i.e. the actual 'word'
def get_labels(path=dir):
    labels = list()
    for subdir, dirs, files in os.walk(dir):
        if (subdir != dir):
            label = os.path.basename(subdir)
            labels.append(label)
            label_indices = np.arange(0, len(labels))

    return labels, label_indices, to_categorical(label_indices)
```

```
# Saving the data into a numpy (.npy) file (when MFCC output = 20 x 11 matrix)

def save_data_to_array(path=dir, max_pad_len=11):
    labels, _, _ = get_labels(path)

    for label in labels:
        mfcc_array = []

        for subdir, dirs, files in os.walk(os.path.join(dir, label)):
            for x in files:
                if x.endswith(".wav"):
                    actual_wavfile = os.path.join(os.path.join(dir, label), x)
                    mfcc = calc_mfcc(actual_wavfile, max_pad_len=max_pad_len)
                    mfcc_array.append(mfcc)

        np.save(label + '.npy', mfcc_array)
```

```
# Saving the data into a numpy (.npy) file (when MFCC output = 13 x 11 matrix)

def save_data_to_array(path=dir, max_pad_len=11):
    labels, _, _ = get_labels(path)

    for label in labels:
        mfcc_array = []

        for subdir, dirs, files in os.walk(os.path.join(dir, label)):
            for x in files:
                if x.endswith(".wav"):
                    actual_wavfile = os.path.join(os.path.join(dir, label), x)
                    mfcc = calc_mfcc13(actual_wavfile, max_pad_len=max_pad_len)
                    mfcc_array.append(mfcc)

        np.save(label + '.npy', mfcc_array)
```

DATA MODELING

Training Set and Test Set Split

Since each word has about 1700 to 2300 audio files associated with them, it might take a lot of time to develop a model with reasonably good accuracy with the limited infrastructure. Therefore, I decided to start my model on a list of 5 randomly selected words which include ‘bed’, ‘cat’, ‘happy’, ‘yes’ and ‘no’. The audio files associated with these words were MFCC-transformed and the respective .npy files were used. The MFCC coefficients were treated as the feature matrix, X while the words were converted into categories and treated as the response vector, y. Finally the train-test split was performed which resulted in 6991 records in the training set and 4662 records in the test set.

```
import numpy as np

from sklearn.model_selection import train_test_split

def get_train_test(split_ratio=0.6, random_state=42):
    # Getting the available labels
    labels, indices, _ = get_labels(dir)
    # Save the audio files to numpy array
    save_data_to_array(path=dir, max_pad_len=11)
    # Getting first arrays
    X = np.load(labels[0] + '.npy')
    y = np.zeros(X.shape[0])

    # Creating feature matrix, X and response vector, y by appending all of the dataset into one single array
    for i, label in enumerate(labels[0:]):
        x = np.load(label + '.npy')
        X = np.vstack((X, x))
        y = np.append(y, np.full(x.shape[0], fill_value= (i + 1)))

    assert X.shape[0] == len(y)

    return train_test_split(X, y, test_size= (1 - split_ratio), random_state=random_state, shuffle=True)
```

```
# Creating the train-test split (when MFCC output = 20 x 11 matrix)

X_train, X_test, y_train, y_test = get_train_test()

X_train = X_train.reshape(X_train.shape[0], 20, 11, 1)
X_test = X_test.reshape(X_test.shape[0], 20, 11, 1)
y_train_cat = to_categorical(y_train)
y_test_cat = to_categorical(y_test)
```

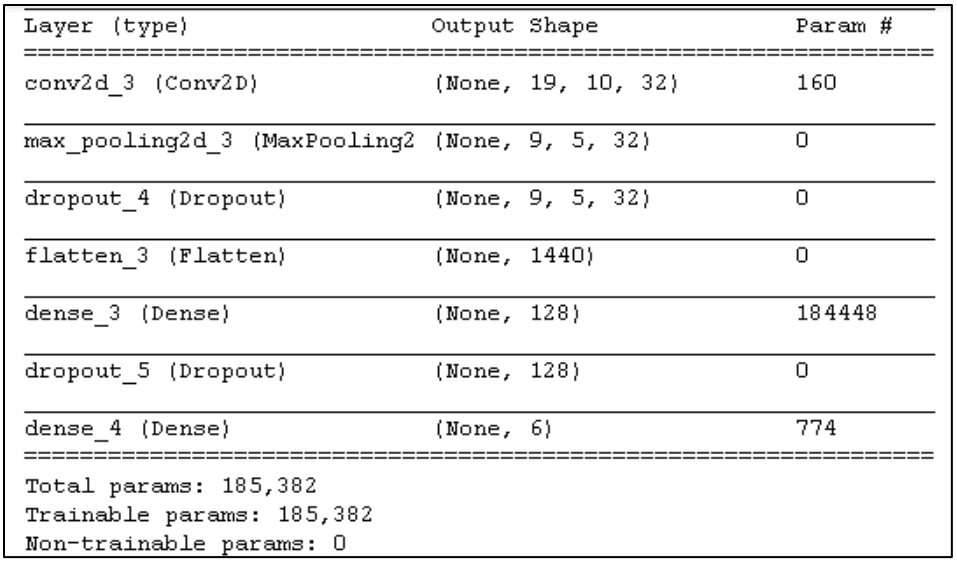
```
# Creating the train-test split (when MFCC output = 13 x 11 matrix)

X_train, X_test, y_train, y_test = get_train_test()

X_train = X_train.reshape(X_train.shape[0], 13, 11, 1)
X_test = X_test.reshape(X_test.shape[0], 13, 11, 1)
y_train_cat = to_categorical(y_train)
y_test_cat = to_categorical(y_test)
```

Convoluted 2D Neural Network Model

Since data preprocessing using MFCC transformation yielded an 11 x 20 matrix resulting in an image-like data, I decided to go with convolutional 2D Neural Network (Conv2D NN or CNN) model. CNNs use a variation of multilayer perceptrons designed to require minimal preprocessing and can be used for image recognition as well as natural language processing. I used Sequential CNN using the package 'keras' with 'Theano' backend. The model was parameterized with maxpooling size of (2, 2) and reshaped the input shape to (20, 11, 1) to represent the two-dimensional image in the 3D space. The ReLU layer represents the number of Rectified Linear Units which increases the nonlinear properties of the overall network (usually in the powers of 2) and the Softmax Dense is the generalization of the logistic function which represents the number of classes (here, the number of words). The model was compiled with function to minimize the categorical crossentropy losses and the accuracy was measured for both training set and testing set. The model was run in a batch size of 100 with epoch (number of iterations) value of 50.

```
# Defining and importing estimator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.utils import to_categorical
```

```
# Instantiating the estimator with a variable and other parameters for input shape = (20, 11, 1)
model = Sequential()
model.add(Conv2D(32, kernel_size=(2, 2), activation='relu', input_shape=(20, 11, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(6, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(),
metrics=['accuracy'])
model.summary()
```



Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 19, 10, 32)	160
max_pooling2d_3 (MaxPooling2	(None, 9, 5, 32)	0
dropout_4 (Dropout)	(None, 9, 5, 32)	0
flatten_3 (Flatten)	(None, 1440)	0
dense_3 (Dense)	(None, 128)	184448
dropout_5 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 6)	774

Total params: 185,382
Trainable params: 185,382
Non-trainable params: 0

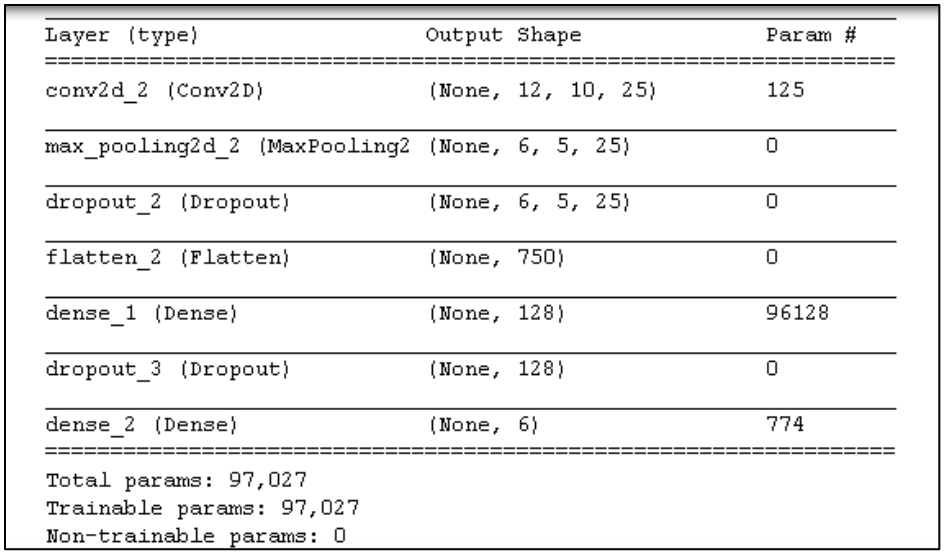
Fig.10: Model Summary for input shape = (20, 11, 1)

```
# Fitting the model and validating the accuracy
model.fit (X_train, y_train_cat, batch_size=100, epochs=50, verbose=1, validation data= (X_test,
y_test_cat))
```

The metrics for calculating the accuracy is built inside the compile function of the model. This model resulted in a testing set accuracy of with 50 iterations and the time take for each epoch varied between 3300-3800 seconds.

The data modeling process was repeated with input shape (13, 11, 1):

```
# Instantiating the estimator with a variable and other parameters for input shape = (13, 11, 1)
model = Sequential()
model.add(Conv2D(25 kernel_size=(2, 2), activation='relu', input_shape=(13, 11, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(6, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(),
metrics=['accuracy'])
model.summary()
```



Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 12, 10, 25)	125
max_pooling2d_2 (MaxPooling2	(None, 6, 5, 25)	0
dropout_2 (Dropout)	(None, 6, 5, 25)	0
flatten_2 (Flatten)	(None, 750)	0
dense_1 (Dense)	(None, 128)	96128
dropout_3 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 6)	774

Total params: 97,027
Trainable params: 97,027
Non-trainable params: 0

Fig.11: Model Summary for input shape = (13, 11, 1)

```
# Fitting the model and validating the accuracy
model.fit(X_train, y_train_cat, batch_size=100, epochs=50, verbose=1, validation_data=(X_test,
y_test_cat))
```

This model resulted in a testing set accuracy of ~76% with 50 iterations and the time take for each epoch varied between 1720-1750 seconds.

MODEL PREDICTION AND EVALUATION

The above model (both with input shape = (20, 11, 1) and input shape = (13, 11, 1) were used to predict the test set and the predicted labels (i.e. words) were compared against the actual labels (or words).

```
# Predicting the test set
y_pred_class = model.predict(X_test, batch_size=100, verbose=1)
```

```
# Printing out the predicted labels for comparison
for x in range(0,9):
   print('Actual result:' + get_labels()[0][np.argmax(y_pred_hot[x], axis=None, out=None)])
   print('Expected result:' + get_labels()[0][np.argmax(y_test_hot[x], axis=None, out=None)])
```

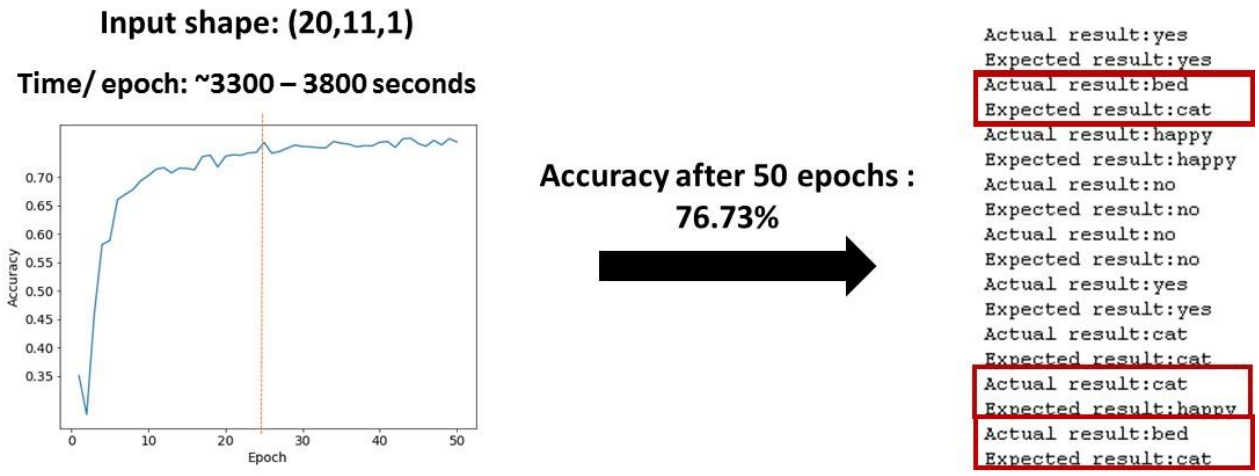


Fig.12: Model Prediction and Accuracy for input shape = (20, 11, 1)

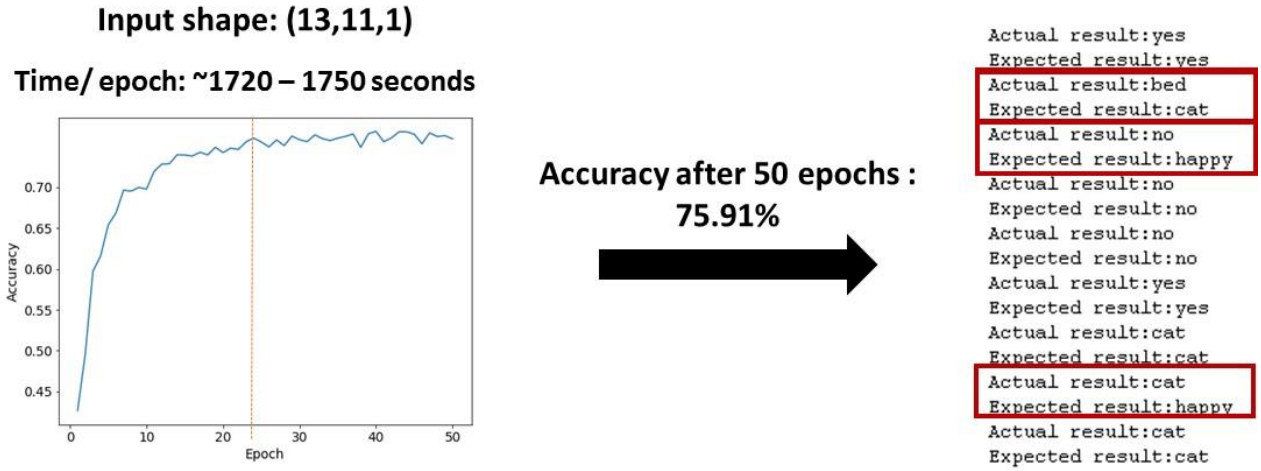


Fig.13: Model Prediction and Accuracy for input shape = (13, 11, 1)

CONCLUSION AND FUTURE DIRECTIONS

The major conclusions that can be drawn from the project are:

1. The accuracy score was found to be 76.73% for input shape = (20, 11, 1) and 75.91% for input shape = (13, 11, 1); therefore, reducing the MFCC matrix dimension did not impact the accuracy significantly.
2. From the accuracy score for both the input shapes, it looks like the accuracy improves with the number of iterations up to ~25 epochs, after which the graph reaches a plateau. Therefore, the

optimal value for epoch is ~25, irrespective of the input shape. However, this has to be further verified with other input shapes.

The future directions include:

1. Finding an optimal number of MFCC matrix dimension to tradeoff between accuracy score and time taken.
2. Using Spark or other multi-threading options and repeating the analysis on the entire dataset to check the robustness of the model.