# Abstract

This report presents a comprehensive study of a voice recognition system developed using Mel Frequency Cepstrum Coefficients (MFCC). The system’s design and methodology are explored in detail, with a focus on feature extraction and the use of MFCC. The system’s performance is evaluated under various conditions, providing insights into its robustness and accuracy. The findings indicate that the system performs exceptionally well on homogeneous datasets, achieving 100% accuracy. However, performance drops slightly on non-homogeneous datasets, suggesting areas for future enhancement. The system also demonstrates reasonable robustness to the loss of certain frequency bands. The report concludes with a discussion on the system’s performance and potential areas for improvement.

# Introduction

In the era of digital transformation, voice recognition has emerged as a pivotal technology, driving advancements in various sectors such as healthcare, automation, and security. This report presents a comprehensive study of a voice recognition system developed using Mel Frequency Cepstrum Coefficients (MFCCs) and the LBG algorithm, a well-established method in the field of speech and audio processing.

We provide first a detailed overview of the voice classification process used to develop our system. We then explain the structure of our code and our system for selecting the specific hyperparameters to be used by our system. We thoroughly evaluation of the system’s performance under various conditions, providing valuable insights into its robustness and accuracy, and finally summarize the strengths and weaknesses of our system and offer specific paths for improvement.

# Methodology

The central principle of voice recognition, and indeed of all classification, is feature extraction: determining or deciding which properties of the data are most discriminatory, i.e. contain the most information about the class of the data.

For voice recognition, an initial approach might be to use the spectrogram of the sound; however, the features of the spectrogram are highly dependent on the specific words being spoken and the speed of articulation, and information about the timbre of the voice, while present, is still buried.

A more successful method is to use MFCCs. The training data is divided into frames of a given length with a given amount of overlap. These frames are windowed to mitigate the spectral distortion which would arise from abrupt changes at the beginning and end of the frame.

The Fourier Transform of each frame is taken, and the strength of the signal in a number of frequency bins is found. The exact bins used is critical in the efficacy of the overall recognition scheme. A popular principle is to use the mel-frequency scale, which is designed to reflect how human aural acuity changes with frequency: it uses linear spacing below 1kHz and logarithmic spacing above.

Once the mel-spectrum values are obtained for each frame, their log is converted from frequency to time domain via the Discrete Cosine Transform (DCT). The logarithm is used to facilitate disentangling the vocal characteristics (a filter) from the essential sound sequence (considered as a pulse train).

After the MFCCs are found for each frame, they are best thought of as vectors in a n-dimensional space, where n is the number of mel-scaled bins used. A person’s voice will tend to produce vectors of a certain type, or types, corresponding to the qualities of their voice. Formally, this is implemented via a clustering algorithm, such as the LBG algorithm.

A centroid is defined at the mean of all the MFCC vectors. Then it is split into two centroids, still close to the mean on opposite sides. The MFCC vectors are grouped according to which of the two centroids are closest. The position of the two centroids are updated based on the mean of these two groups of vectors, new groups are formed from those newly positioned centroids, and the process repeats until the average distance drops below a certain threshold. Then the two centroids are split, and the entire process repeats until a certain number of centroids is attained. When the algorithm finishes, the final centroids are saved as a codebook for that audio file.

A codebook is generated for each audio file in the training set. Then, to classify any given audio file, it is split into frames and MFCCs are found for each frame. Whichever class’s codebook has centroids with the smallest overall distance to the test MFCCs for each frame is the predicted class.

# Design Choices

Our system uses an object-oriented approach. The basic class is the “codebook”, which has methods for feature extraction, centroid creation, and saving its centroids as a numpy array in a specified folder, and loading a set of centroids from a numpy array from a specified folder. This facilitates creation of new codebooks and using pretrained codebooks. Given a new set of data, it has a method for finding the distance between its set of centroids and the new data. It finally has a method for plotting the centroids on a given set of axes.

The class “codelibrary” behaves as something as an organizer or mother of codebooks; it has methods for creating and saving a set of codebooks corresponding to a set of training data in a given folder. Its predict method finds the codebook which has the smallest distance from a piece of given data; its getAccuracy method runs the predict method for all the audio files in a specified testing directory, determines whether the prediction is correct, and returns the overall accuracy of the system.

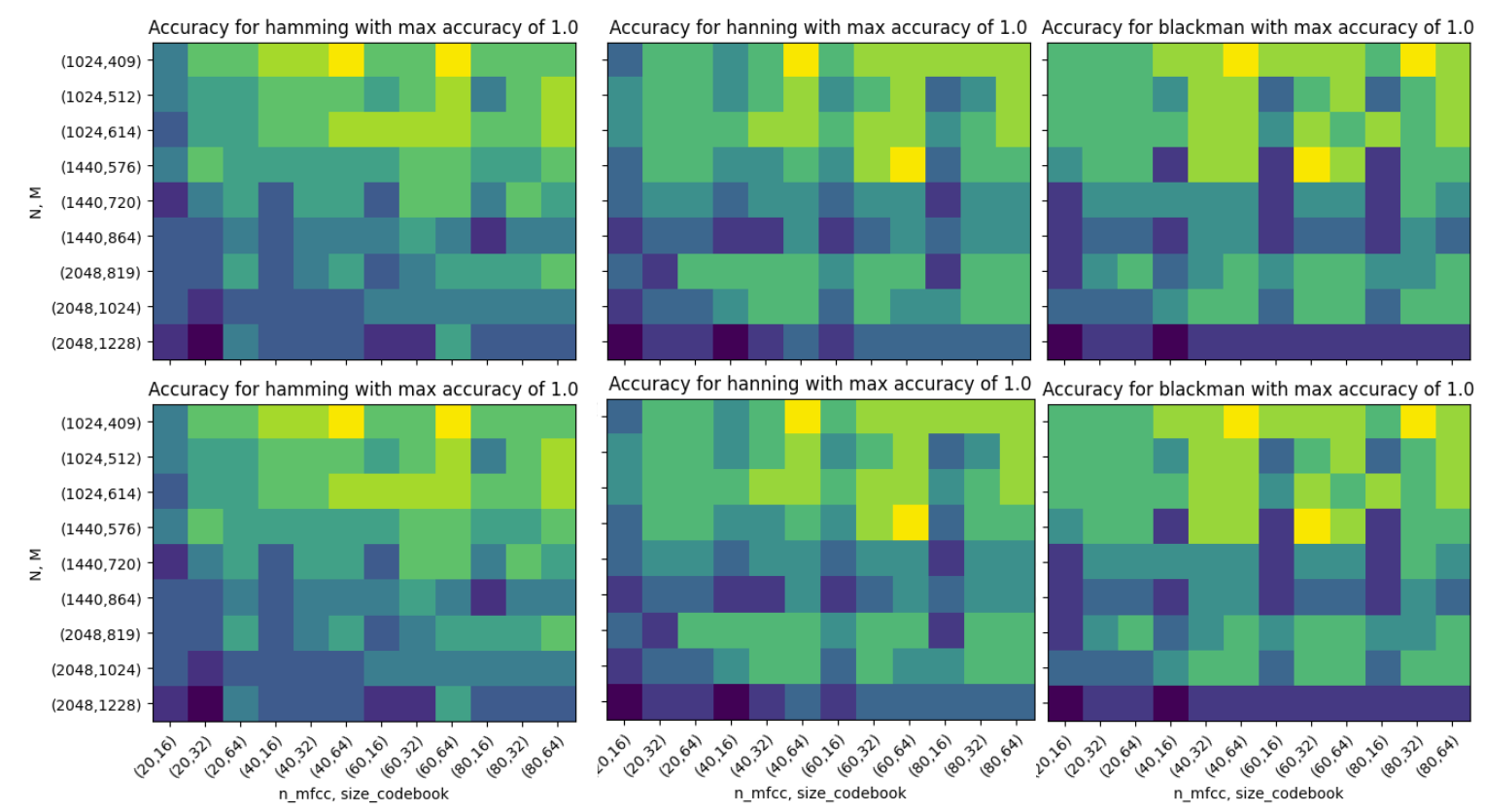
## Hyperparameter Sweep

The core system described in the Methodology section has a number of hyperparameters which can be tuned to improve performance on a specific instance of the speech recognition problem.

These hyperparameters, and their variable representations in our code, are described in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Hyperparameter** | **Meaning** | **Variable** | **Hyperparameter Range** |
| Frame Length | Number of Samples in Frame | N | 1024, 1440, 2048 |
| Hop Size | Number of Samples between start points of subsequent frames | M | 0.4, 0.5, 0.6 (fraction of N) |
| Window | Type of window applied to frame | window | Hamming, Hanning, Blackman, Bartlett, Kaiser |
| Number of MFCCs | Corresponds to the number of frequency bins | n\_mfcc | 20, 40, 60, 80 |
| Codebook Size | Number of centroids | size\_codebook | 16, 32, 64 |

To improve the performance of our system on the in-class recordings, we performed a hyperparameter sweep over the values given in the Hyperparameter Range column of the table. Figure 1 contains an example of the results.



For each window except Kaiser, the hyperparameters N=1024, M=409, n\_mfcc = 40, and size\_codebook = 64 resulted in perfect accuracy, so those parameters were selected for the system (with modifications to N and M for different sampling rates).

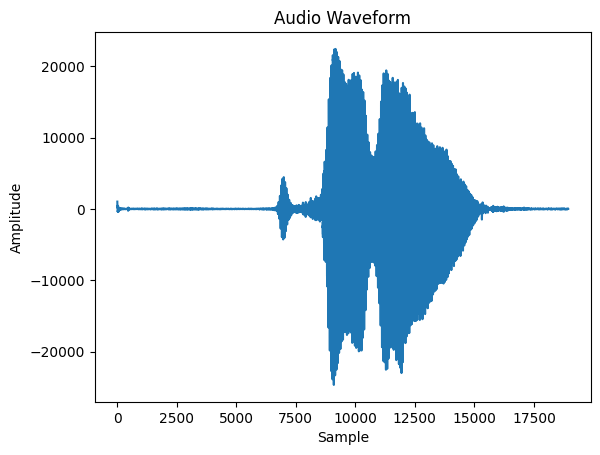
# Tests

## Test 1

Randall played the entire training dataset for Conor to familiarize himself with, and then played the test dataset. Conor correctly identified 3 of the 8 test audio files, giving us a baseline human accuracy of 0.375.

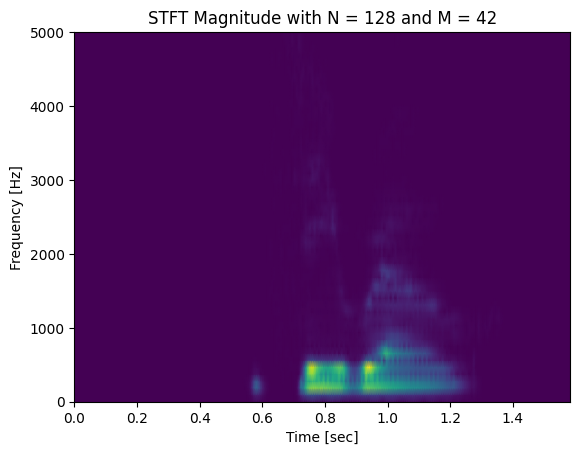
## Test 2

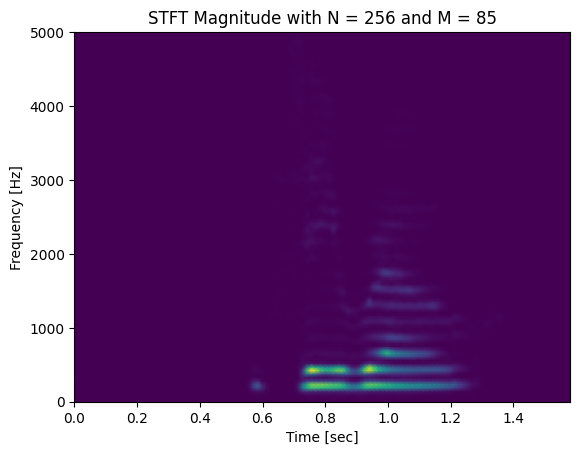
Our Test.ipynb file makes it easy to play and plot any of the audio files in the provided data. For example, here is a plot of the 2nd of the audio files in the test set.

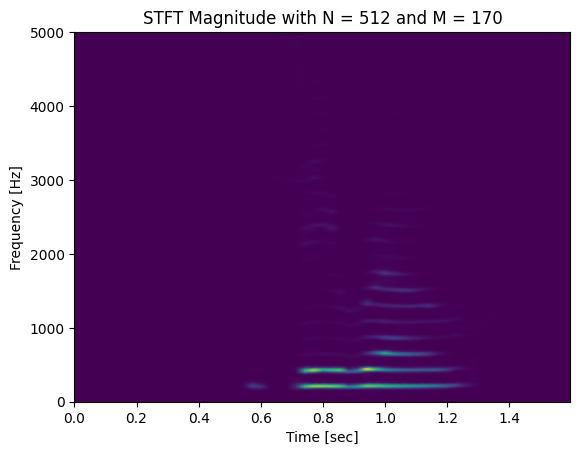


The sampling rate of the given data is 12500; at this sampling rate, 256 samples corresponds to 20.5 ms.

The last of the 8 test audio files could not be read originally; we exported and resaved it in a different program. This fixed the issue, but it did result in a sampling rate of 12000 for that file specifically.



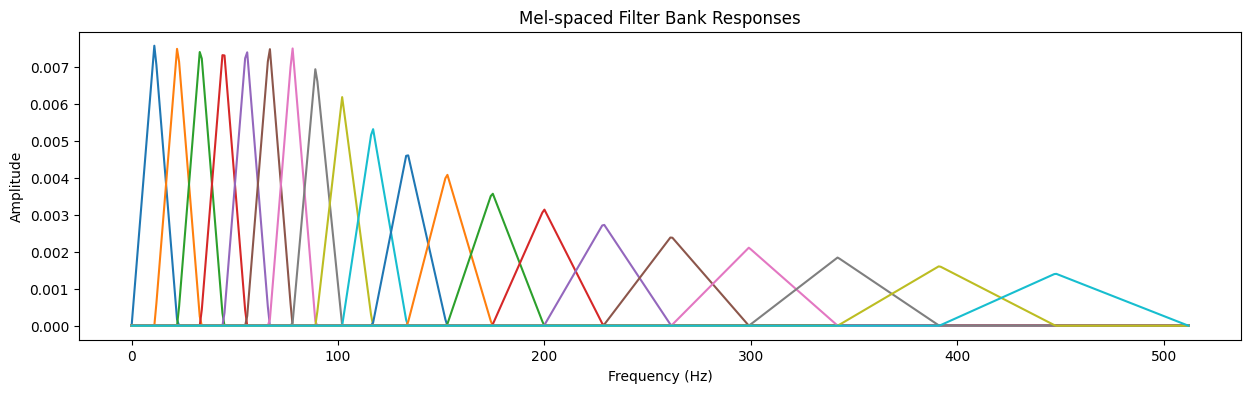




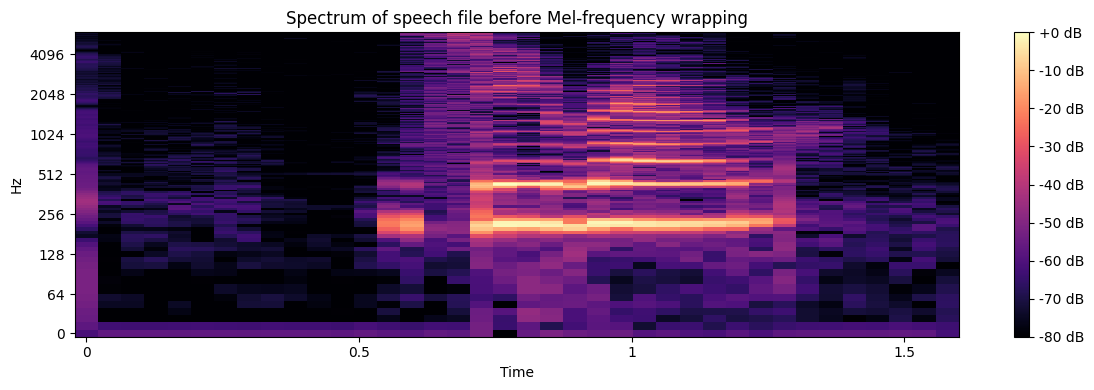
Spectrograms of this audio sample with different values of frame length (N) and overlap (M). The tradeoff between time and frequency resolution as N increases is clearly visible.

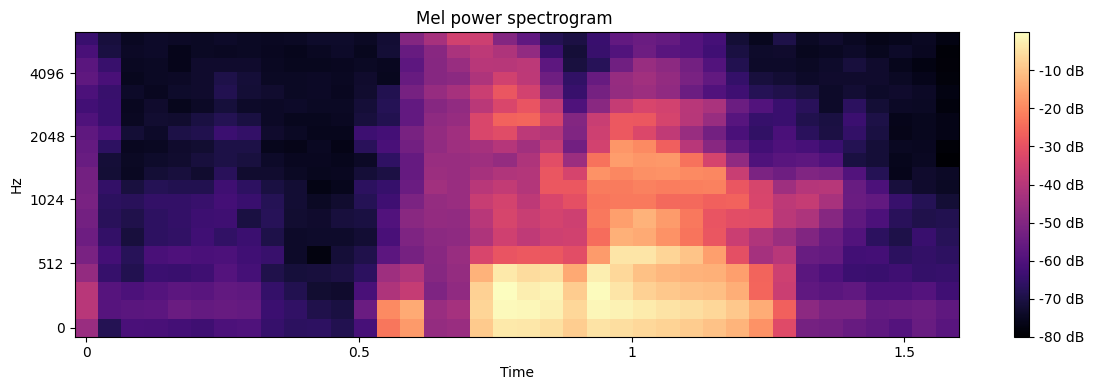
## Test 3

The mel-spaced filter bank responses (in this system, the switch from linear to logarithmic occurs at 100 Hz rather than 1 kHz). After the spacing becomes logarithmic, the height of the triangles decreases to keep total energy in each filter constant, despite the increasing bandwidth.



Difference in spectrogram before and after mel-frequency wrapping step:





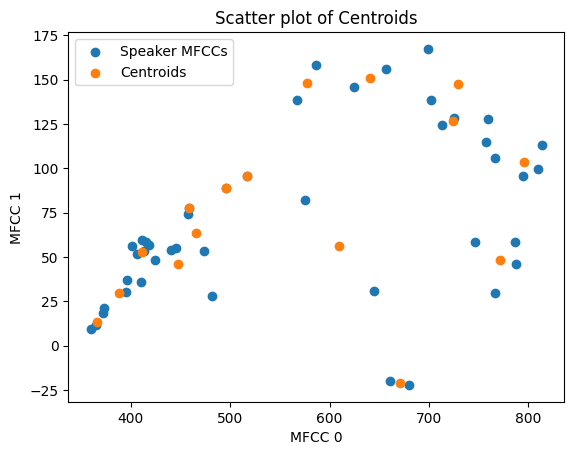
The mel-frequency wrapping step allows a similar amount of information (from a classification standpoint) to be placed in each bin, leading to more efficient classification.

## Test 4

The feature-extraction function in our processing.py file takes in a vector of audio data and selected hyperparameters, and returns a vector of MFCC coefficients.

## Tests 5 and 6

The Tests 5 and 6 section of our Tests.ipynb file allows you to easily choose any two dimensions of the MFCCs to use as axes and plot both the MFCCs of the current audiofile, and the resultant centroids. In this case, there are 16 centroids, plotted along dimensions 0 and 1 out of the 40 MFCCs.



## Test 7

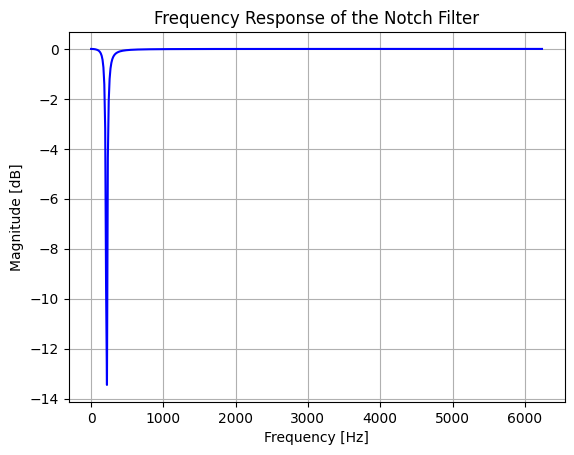
This was the baseline test of our system on the provided training and test data. It creates a codelibrary based on the training data, and then tests the 8 test data files. Our system had an accuracy of 1.0 with the following hyperparameters: N = 256, M = 100, n\_mfcc = 40, window = Hamming, size\_codebook = 64.

## Test 8

This test creates a notch filter at a given frequency and with a given quality factor, and then creates a new set of test files after applying the filter. The same library as Test 7 then predicts these new files.

We informally tested many different notch filters; our “official” tests consisted of notch filters as 215 Hz, 440 Hz, 1000 Hz, and 6000 Hz, each with a quality factor of 5.

This is the frequency response of the 215 Hz notch filter:



The following table summarizes our results:

|  |  |
| --- | --- |
| Notched Frequency (Hz) | Performance |
| 215 | 1.0 |
| 440 | 1.0 |
| 1000 | 0.75 |
| 6000 | 0.875 |

Overall, our system was reasonably robust to losing frequency components; the 1000 Hz range seems to have been the most key area for detection.

## Test 9

To perform this test, new training and test datasets were created, composed of the original given datasets and 10 of the “zero” in-class recordings, chosen at random. Due to the different sampling rates (12.5 kHz for the given data, 48kHz for the in-class recordings), the given data needed to be upsampled by a factor of 4, and hyperparameters N and M were updated to 1024 and 409, as specified by the parameter sweep. A new code library was trained on the new training dataset. The accuracy of this code library was found to be 0.83.

## Test 10

### Question 1

This test simply consisted of training and testing our system on the “twelve” in-class recordings. The accuracy of the system was found to be 1.0.

### Question 2

#### a)

To perform this test, we created training and test datasets composed of every audio file: the upsampled given “zero” recordings, the “zero” in-class recordings, and the “twelve” in class-recordings. The accuracy of our system on this aggregate data-set was 0.886.

#### b)

This tested the ability of our system to distinguish not different voices, but the words “zero” and “twelve”. For a given test file, the system was considered correct as long as the predicted class was in the correct group, “zero” or “twelve”. This required a custom accuracy test specifically designed for the structure of our aggregate datasets. According to this metric, the system has an accuracy of 1.0.

# Conclusion

Overall, our system was extremely reliable at predicting homogenous datasets. For datasets which all had the same sampling rates and background noise characteristics, the system and hyperparameters we selected achieved 100% accuracy. When the data became non-homogeneous, in tests 9 and 10, performance suffered somewhat, although the accuracy remained above 80% in all cases. To improve this, a more precise rate-changing scheme might prove effective, as well as applying a filter to remove background noise.

Our system was also fairly robust to losing certain frequency bands, although a notch filter at 1kHz was certainly detrimental.

Other improvements which could be made is clipping the ends of the audio files, removing occasional loud background noise, and normalizing each audio file, so that the loudness of a voice does not play a factor in classification. This is likely overkill for this specific situation, as most people had similar volume, but it would produce a more robust system in general.

# Contributions

Randall: Underlying class creation, parameter sweep, report outline, recorded presentation.

Conor: Class refinement, test implementation, report writing, recorded presentation.