Fundamentals of Big Data Analytics Assignment # 1 Report



Submitted to: Sir Kifayat Alizai Submitted by:

Mirza Nehan

Eman Ijaz

Hammad Javaid

Contents

Introduction………………………………………………………………………………………………………………………………………2

Part (Q1)………………………………….………………………………………………………………………………………………………..2

PRE-PROCESSING …….…………………………………………………………………………………………………………..2

LSH HASHING………………………………………………………………………………………………………………………..4

GENERATING BUCKETS………………………………………………………………………………………………………………………5

IMPLEMENTATION…………..………………………………………………………………………………………………………………6

Part 1 (Q2) ....………………………………………………………….……………………………………………………………………….7

TASK 2 .………………………………………………………………………………………………………………………………………..8

Contributions..………………………………………………………….……………………………………………………………………….9

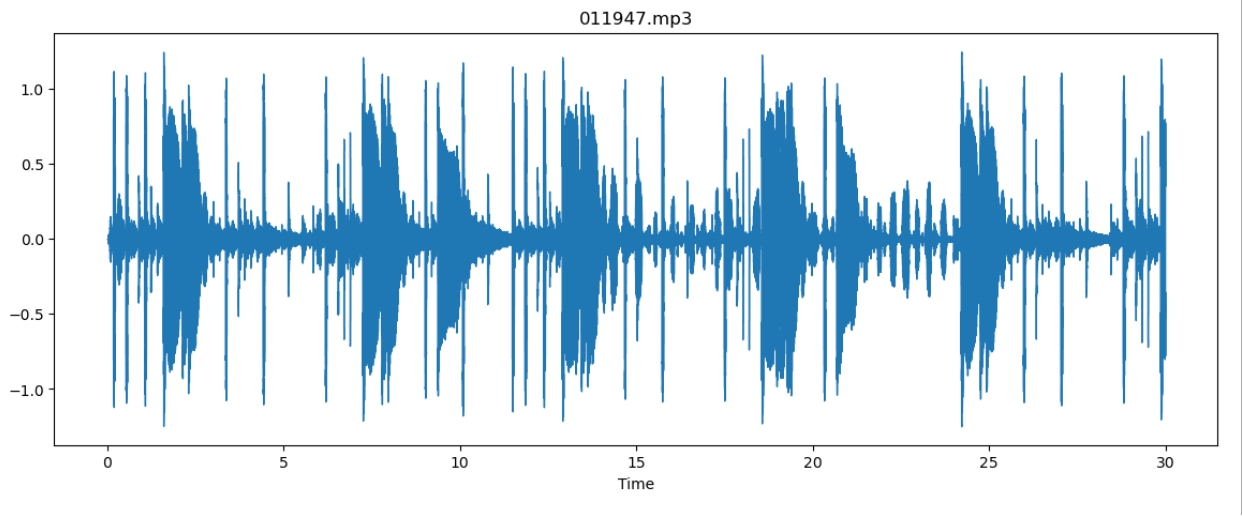
Introduction:

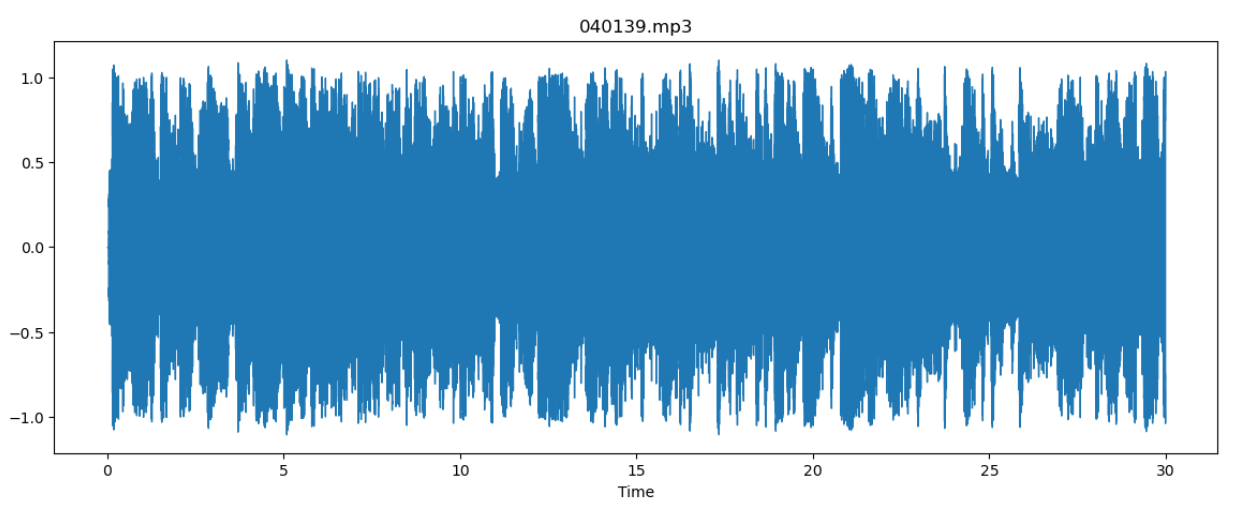
LSH (Locality-sensitive-hashing) is the process of assigning items into different buckets. Items (data points) that are similar to each other are placed into the same bucket. To group similar items into the same bucket, it is necessary to define a similarity metric to generate a score for how similar two items are to one another. A common similarity metric is the Jaccard Index - which is intersection over a union.

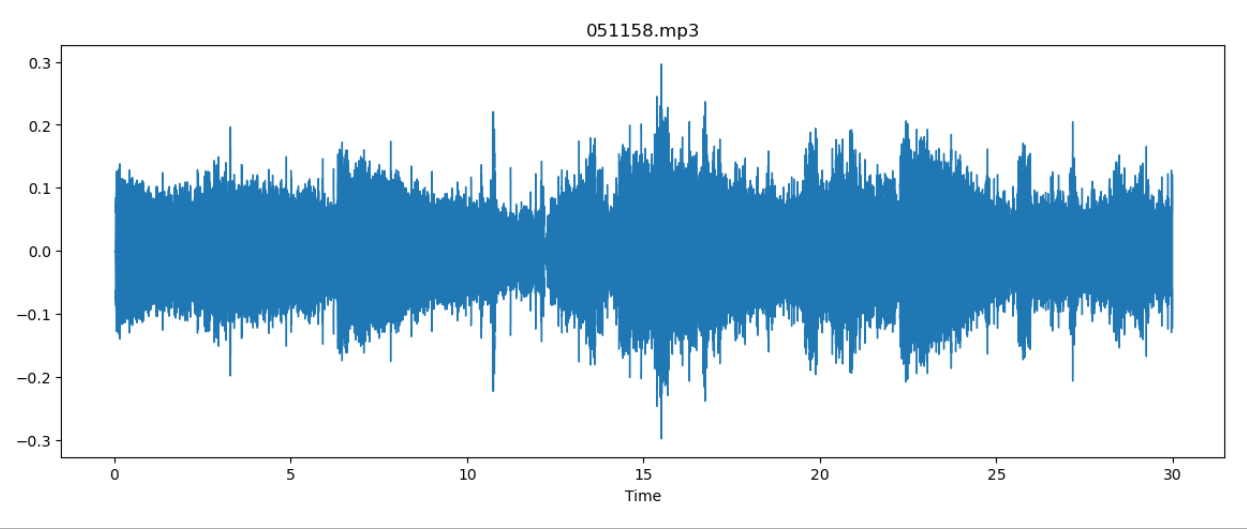
Part I: Audio Duplicate Detection using Locality Sensitive Hashing

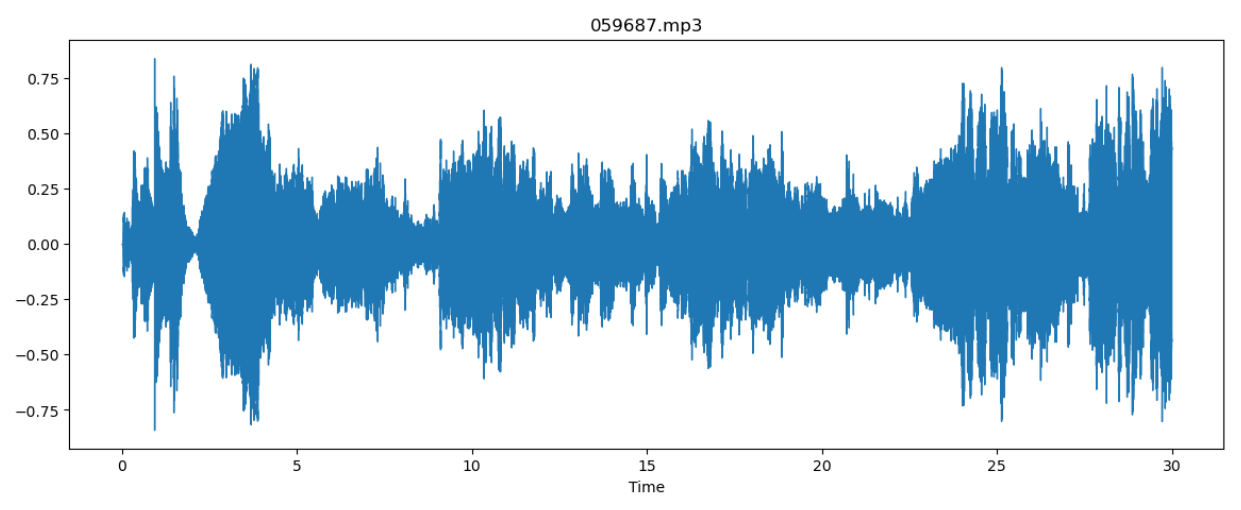
Pre-processing: We extracted the Mel-frequency cepstral coefficients (MFCCs) from the audio files. MFCCs are commonly used features for audio analysis and are often used for speech recognition and music information retrieval.

Viewing every 500th audio to get a general idea of the dataset.









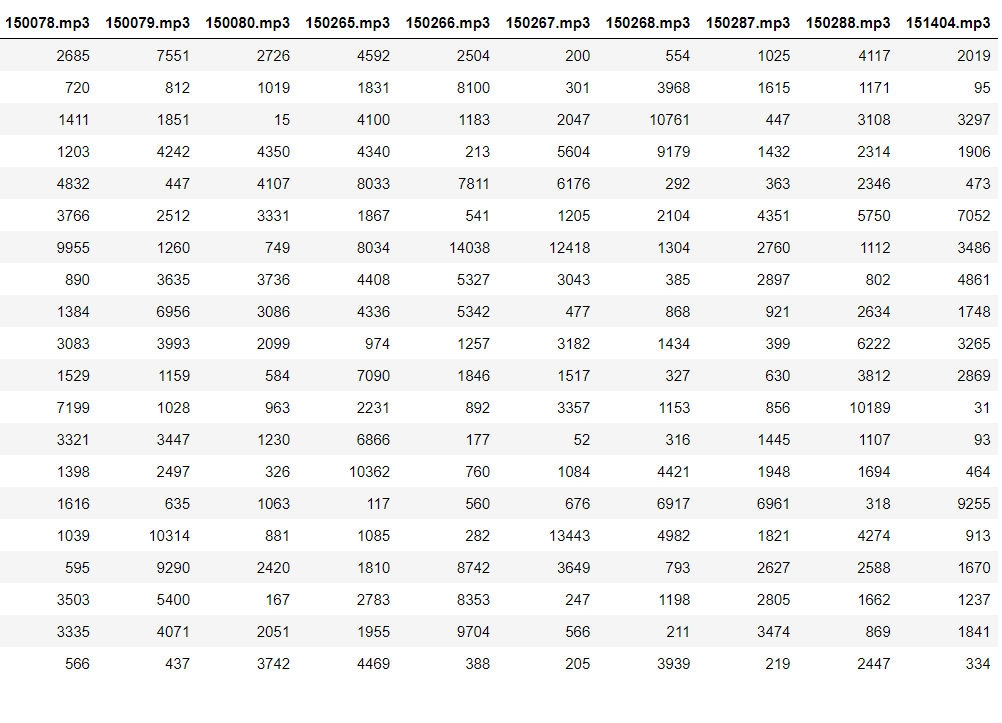
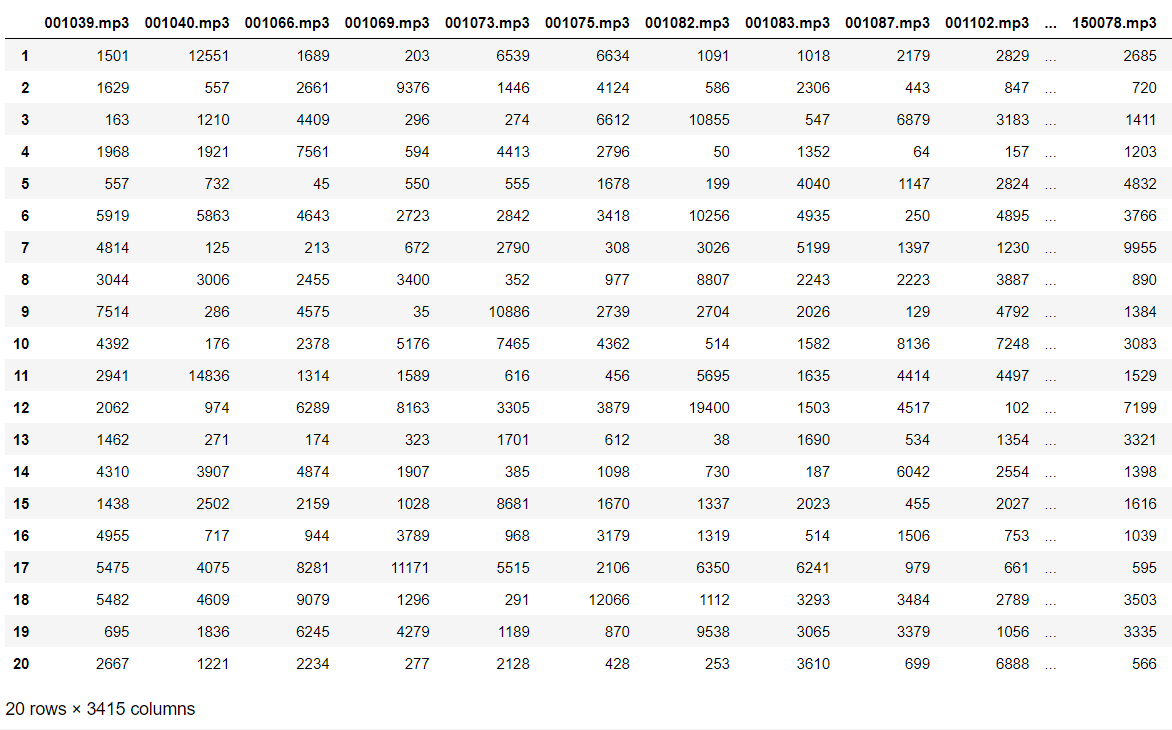
The amplitudes are generally high and there are no silences in the audios. So, we can directly extract MFCCs. We were provided with dataset of audio files (5 GB). The first step was to get the paths of all audio files which was done using Glob. All the audio files were read by librosa.read(), in which the re\_type=’kaiser\_fast’ allowed faster extraction. To get the feature vectors librosa.feature mfcc was used. The resulting mfccs were stored in a dictionary where the keys were audio paths and mfccs were the values.

LSH Hashing:

After feature extraction the next step was to implement minHash and LSH. We started by creating unique shingles. The mfccs were stacked horizontally to form a 1D NumPy array. Then using dict.fromkeys() we extracted the unique values from the NumPy array and converted them to a list—shingles.

To create the binary matrix, we compared the shingles with mfccs of each song one by one, using a helper function. If the shingle value was present in the current song a 1 was placed in matrix otherwise 0. The helper function returned a list which was vertically stacked to form a complete binary matrix.

The next step was to perform random permutations and build a hash matrix. The binary matrix was first transposed to make the shingles as rows and song names as columns. This was converted into a data frame for better computations. The dataframe was then sample randomly using df.sample(frac = 1, random state = i). Frac=1 means that we need the whole dataframe and the random state= i randomly arranges the dataframe ‘i’ times. As we did 20 permutations, ’i’ was changed in a loop from 1 to 20. The problem statement asked us to perform min hashing therefore after the dataframe was permuted the indexes were reset and the first index where a 1 occurred was chosen and stored. This was done for all 20 permutations. The resulting dataframe was a min hash matrix.



Generating Buckets Using Hash Matrix Division and Combination of Tuple IDs:

The last step performed on the given dataset was to generate buckets. We divided the hash matrix into 10 bands of 2 rows so that there will be more chances of a match. The indexes of the 2 rows were combined into a tuple which acted as the bucket id and the name of the songs were the values.

Sample:

A computer screen capture

Description automatically generated with low confidenceText

Description automatically generated

Implementation of Locality Sensitive Hashing (LSH) :

Now to handle the queries, all the above algorithm was performed once again but this time on the query. After the query was mapped into a bucket, we calculated the jaccard similarity with all the songs present in that bucket. The song which had the highest jaccard value was displayed on the html page.

* 3415 documents stored as signature of length 20.
* Signature matrix: 20\*3415
* Brute force comparison of signatures will result in 20C2 comparisons.
* Let’s take b = 10 → r = 2
* We want 2 documents (D1 & D2) with 80% similarity to be hashed in the same bucket for at least one of the 20 bands.
* P (D1 & D2 identical in a particular band) = (0.8)^2 = 0.64
* P (D1 & D2 are not similar in all 20 bands) = (1–0.64)^10 = 0.000037
* This means in this scenario we have ~.037% chance of a false negative for 80% similar documents.

Problems Faced:

* The mfccs took a lot of time to load.
* Creating the hash matrix was heavy on the storage.

Part (I)//Q2

Findings by Mirza Nehan:

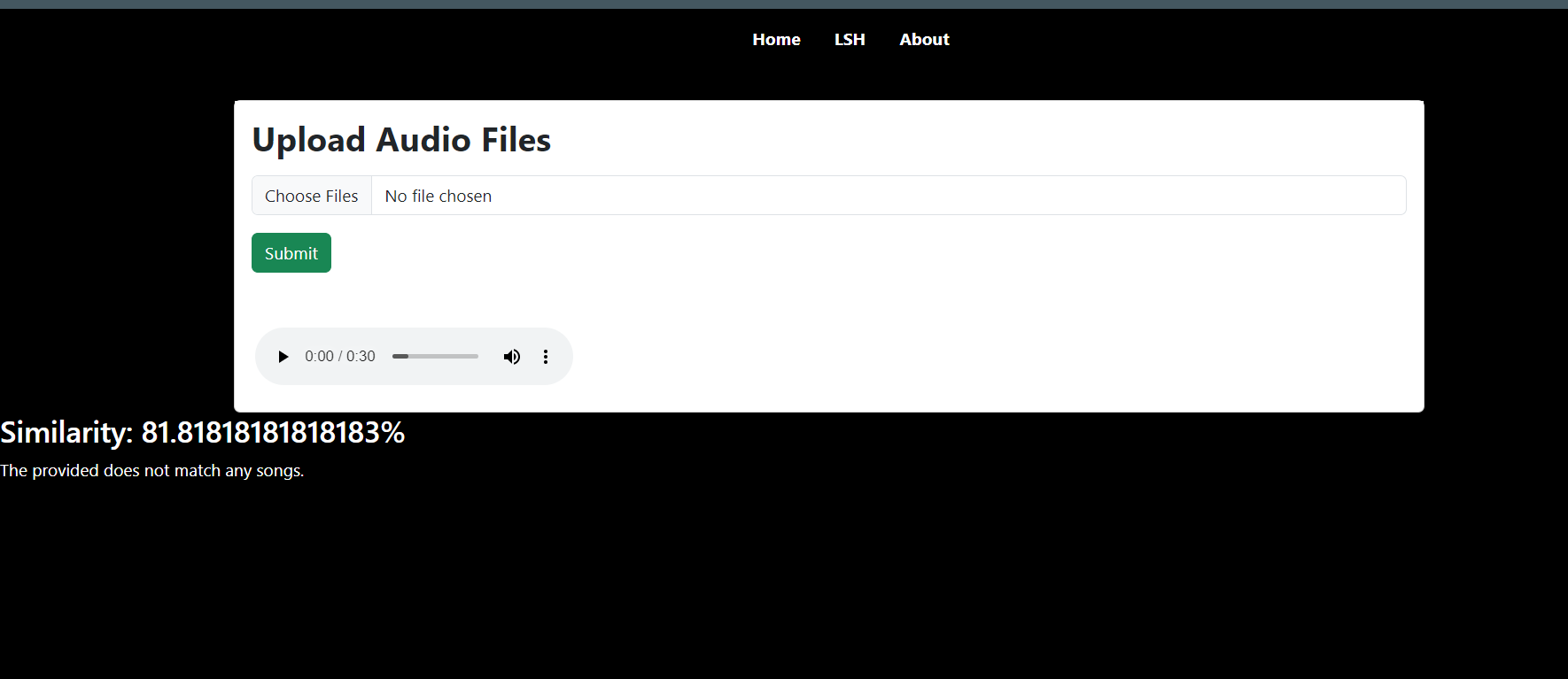
LSH has several potential applications in audio data beyond just detecting duplicates.

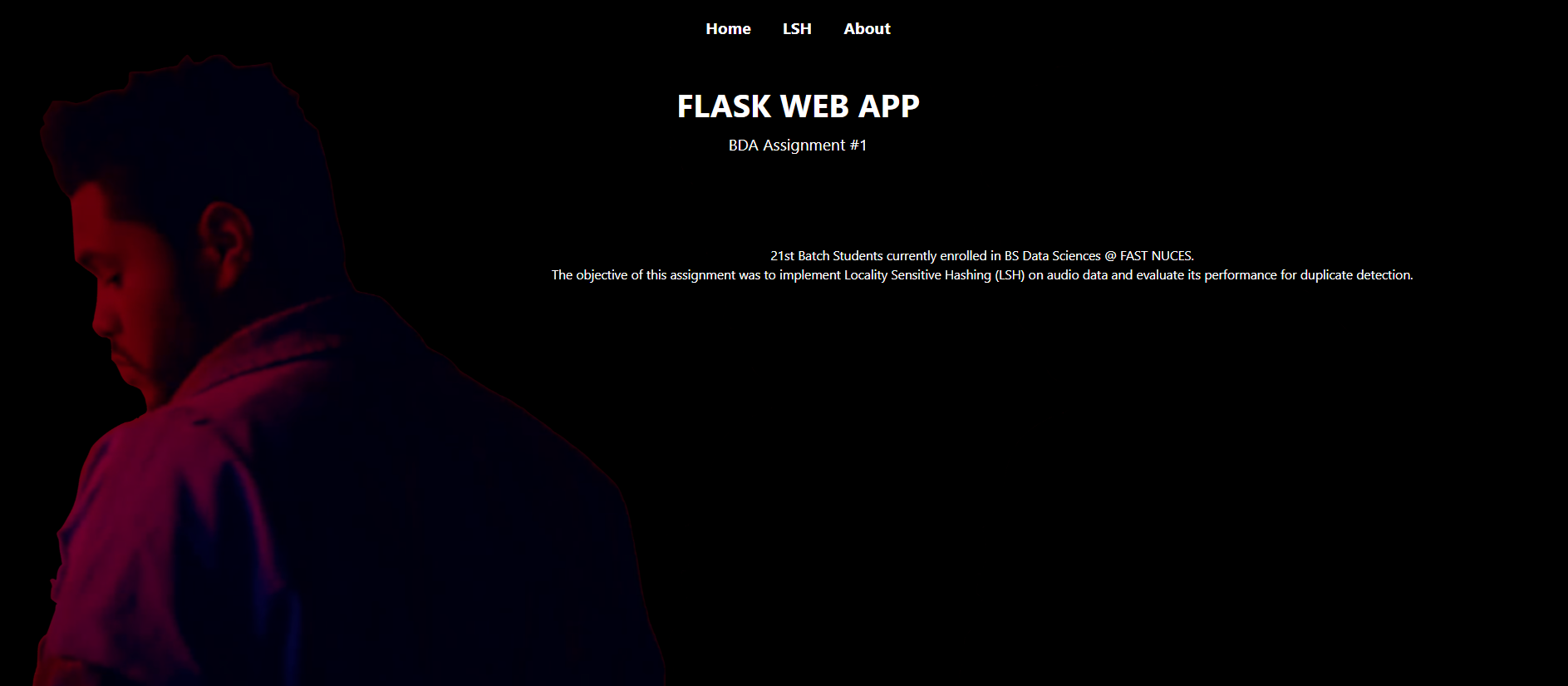
Speech Recognition: LSH can be used to accelerate speech recognition systems by quickly retrieving similar audio samples. For example, given an input audio sample of a person speaking a particular word, LSH can search a database of previously recorded audio samples to find the closest match, improving the accuracy and speed of the recognition process.

Audio Clustering: LSH can also be used to cluster audio data based on similarity. Given a large collection of audio files, LSH can be used to group together similar audio files which can be useful for tasks such as music genre classification or speech segmentation.

Anomaly Detection: LSH can be used for anomaly detection in audio data. For instance, given a set of audio samples from a particular source, LSH can be used to detect any samples that are significantly different from the rest (music genre classification) in a set of audio samples from a particular source, such as identifying fraudulent voice recordings or detecting anomalies in machine-generated audio.

TASK # 2 (FLASK & HTML):





Findings by Hammad Javaid:

I implemented a web application using the Flask framework for audio file upload and duplicate detection. Overall, this code provides a basic framework for building a web application for audio file upload and duplicate detection using LSH. However, the code lacks a bit of error handling and may not be suitable for use in production environments. The implementation showcases the power and versatility of LSH and Flask, and it provides a good starting point for building more advanced audio applications.

**Contributions:**

Task 1 –

Q1- Eman Ijaz

Q2- Mirza Nehan

Task 2 - Hammad Javaid