19CSE353 - MASSIVE MINING OF DATASETS

Music Recommendation System

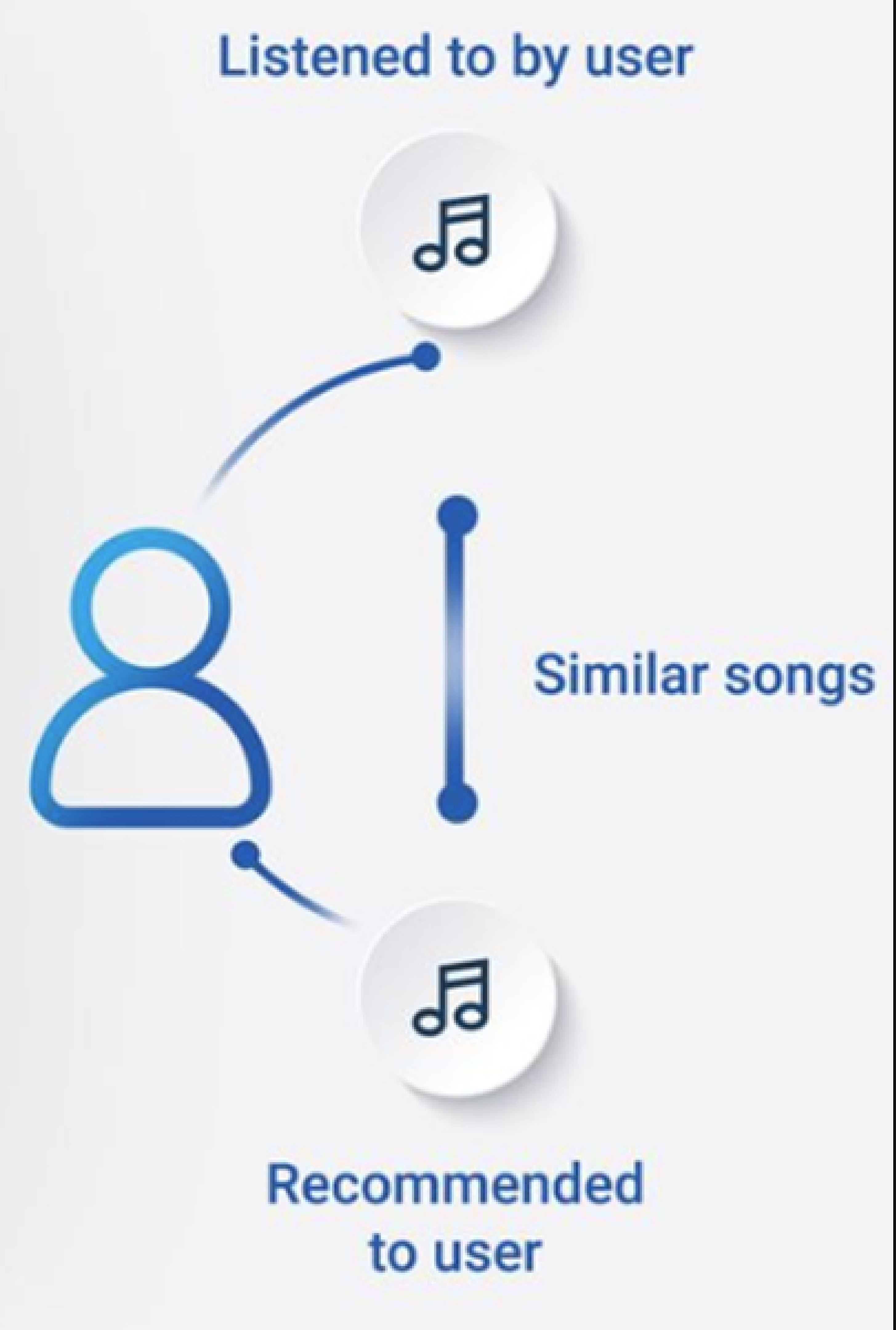
7th November, 2022

Team

S Sai Chadra Kaushik	CB.EN.U4CSE19651
D Rhuthvik	CB.EN.U4CSE19614
G Leepaakshi	CB.EN.U4CSE19630
M A Krithika	CB.EN.U4CSE19629

Problem Statement

Building a Music Recommendation System using Similarity Finding Algorithms



Motivation

Have you ever heard a song and wanted to listen similar songs?

- With commercial music streaming service which can be accessed from mobile devices, the availability of digital music currently is abundant compared to previous era. Sorting out all this digital music is a very time-consuming and causes information fatigue.Therefore, it is very useful to develop a music recommender system that can search in the music libraries automatically and suggest suitable songs to users.
- By using music recommender system, the music provider can predict and then offer the appropriate songs to their users based on the characteristics of the music that has been heard previously.
- Our research would like to develop a music recommender system that can give recommendations based on similarity of features on audio signal.

MP3S-32k Dataset

https://www.kaggle.com/datasets/dhrumil140396/mp3s32k

MP3 AUDIO FILES

1413 SONGS

Project Pipeline

STEP 1

Converting the songs from mp3 to wav with Librosa and extraction of the peaks.

STEP 2

Extracting shingles from songs spectrogram

STEP 3

MinHashing with permutations on the shingles matrix.

STEP 4

Locality sensitive hashing to divide the songs in buckets.

STEP 5

Jaccard similarity calculation

STEP 6

Song recommendation

MinHashing

- Minhashing involves compressing the large sets of unique shingles into a much smaller representation called “signatures”.
- We then use these signatures to measure the similarity
- Although it is impossible for these signatures to give the exact similarity measure, the estimates are pretty close.
- The larger the number of signatures chosen, the more accurate the estimate is.

$$h(x) = (ax + b) \bmod c$$

- x is the row numbers of your original characteristic matrix.
- a and b are any random numbers smaller or equivalent to the maximum number of x, and they both must be unique in each signature.
- b- coefficient in signature
- c is a prime number slightly larger than the total number of shingle sets.

LSH

The concept for locality-sensitive hashing (LSH) is that given the signature matrix of size n (row count), we will partition it into b bands, resulting in each band with r rows. This is equivalent to the simple math formula — $n = br$, thus when we are doing the partition, we have to be sure that the b we choose is divisible by n.

Jaccard Similarity

- After creating shingle sets and characteristic matrix, we now need to measure the similarity between documents.
- We will make use of Jaccard Similarity for this purpose.
- For example, with two shingle sets as set1 and set2, the Jaccard Similarity will be :

$$\frac{|\ set1 \cap set2\ |}{|\ set1 \cup set2\ |}$$

Preprocessing

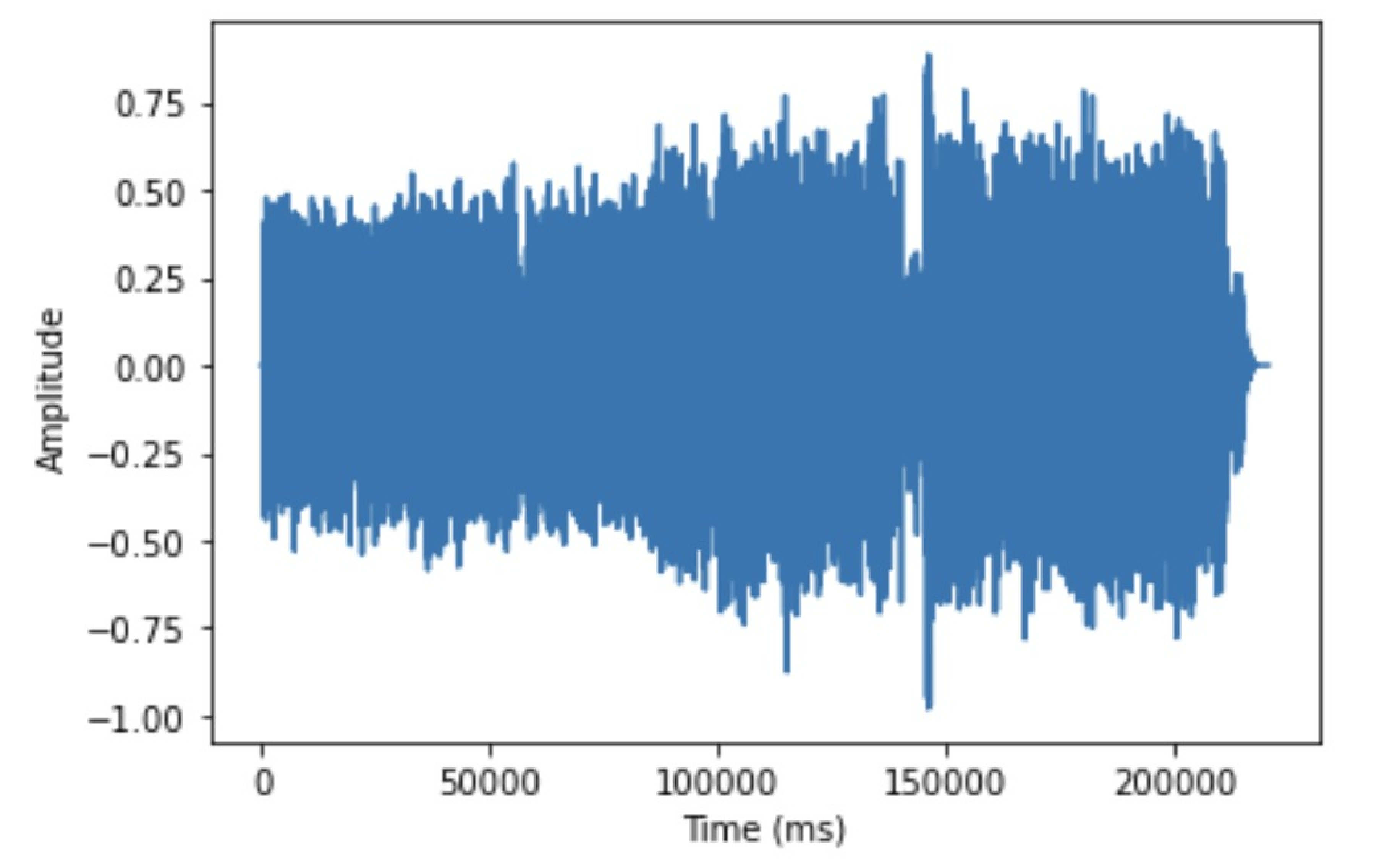
```python
def unique_shingles(song_peaks):

    tot_shingles = list(song_peaks.values())

    shingles = []
    for i in tqdm(tot_shingles):
        shingles.append(i)

    shingles = np.hstack(shingles)
    shingles = np.array(list(dict.fromkeys(shingles))) # all unique peaks

    return shingles
```

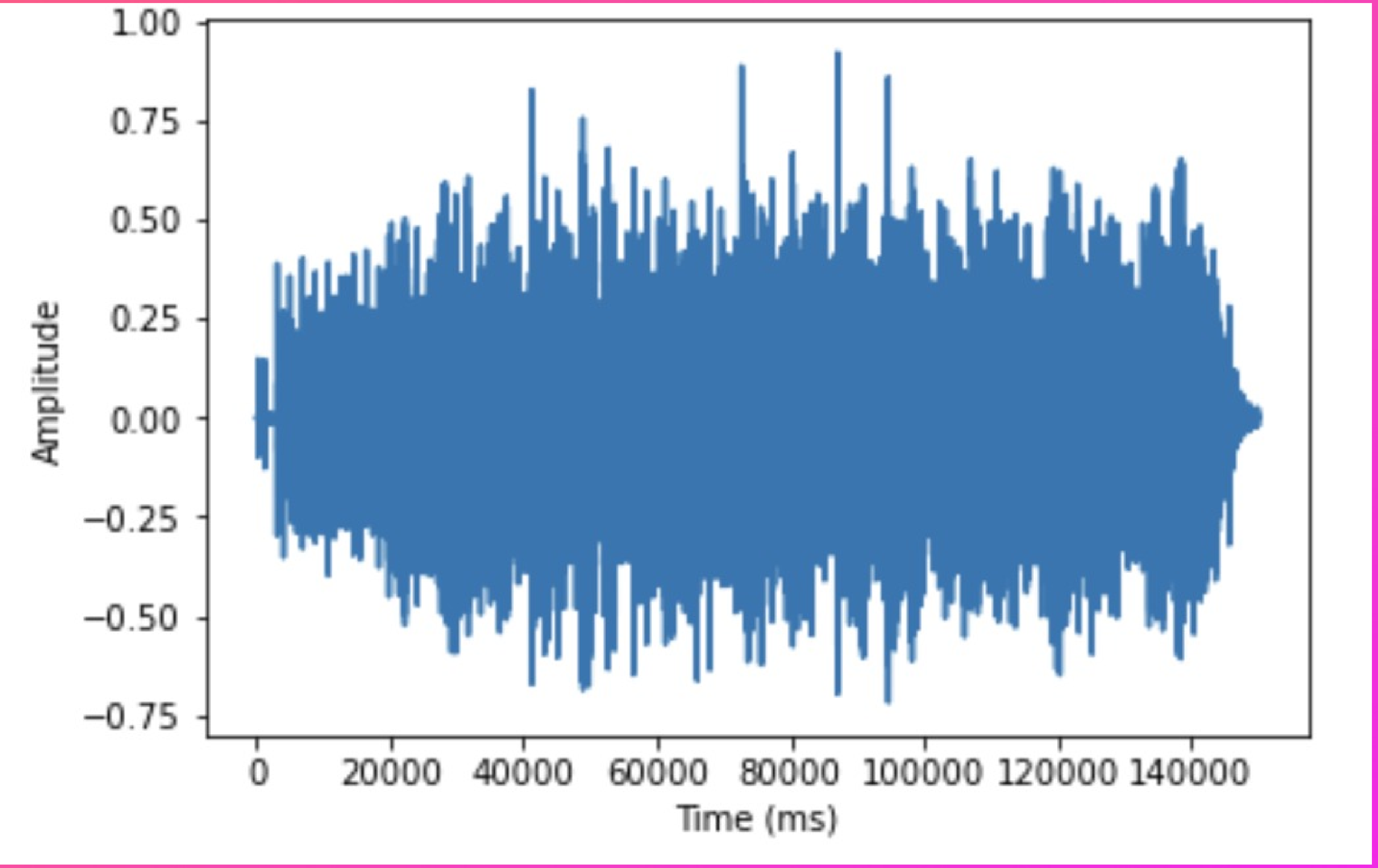
```python
def load_audio_peaks(audio, offset, duration, hop_size):
    d
    track, sr = librosa.load(audio, offset=offset, duration=duration)
    onset_env = librosa.onset.onset_strength(track, sr=sr, hop_length=hop_size)
    peaks = librosa.util.peak_pick(onset_env, 10, 10, 10, 10, 0.5, 0.5)

    return track, sr, onset_env, peaks
```

```python
def convert_mp3_to_wav(audio:str) -> str:
    if audio[-3:] == "mp3":
        wav_audio = audio[:-3] + "wav"
        if not Path(wav_audio).exists():
                subprocess.check_output(f"ffmpeg -i {audio} {wav_audio}", shell=True)
        return wav_audio

    return audio
```

```python
def onehot(peaks, shingles):
    return np.array([1 if x in peaks else 0 for x in shingles])


def shingles_matrix(shingles, song_peaks):

    matrix = np.zeros(len(shingles))

    for v in tqdm(list(song_peaks.values())):
        matrix = np.vstack([matrix, onehot(v, shingles)])

    matrix = np.delete(matrix, (0), axis=0)

    return matrix
```

Spectrograms





Code

```python
def save_pickle(element, path):
    with open(f"{path}", 'wb') as f:
        pickle.dump(element, f, pickle.HIGHEST_PROTOCOL)
```

```python
def load_pickle(path):
    with open(f"{path}", 'rb',) as f:
        return pickle.load(f)
```

```python
def convert_mp3_to_wav(audio:str) -> str:
    if audio[-3:] == "mp3":
        wav_audio = audio[:-3] + "wav"
        if not Path(wav_audio).exists():
                subprocess.check_output(f"ffmpeg -i {audio} {wav_audio}", shell=True)
        return wav_audio

    return audio
```

Code

```
def load_audio_peaks(audio, offset, duration, hop_size):
    d
    track, sr = librosa.load(audio, offset=offset, duration=duration)
    onset_env = librosa.onset.onset_strength(track, sr=sr, hop_length=hop_size)
    peaks = librosa.util.peak_pick(onset_env, 10, 10, 10, 10, 0.5, 0.5)

    return track, sr, onset_env, peaks
```

```
def extract_peaks(song_path, rounded = False):

    song_peaks = {}
    if rounded == True:
        for song in tqdm(song_path):
            tmp1, tmp2, onset, peaks = load_audio_peaks(str(song), OFFSET, DURATION, HOP_SIZE)
            song_peaks[' '.join(str(song).split('/')[-1][3:-4].split('_')).lower() + ' - ' + ' '.join(str(song).split('/')[0].spl
    else:
        for song in tqdm(song_path):
            tmp1, tmp2, onset, peaks = load_audio_peaks(str(song), OFFSET, DURATION, HOP_SIZE)
            song_peaks[' '.join(str(song).split('/')[-1][3:-4].split('_')).lower() + ' - ' + ' '.join(str(song).split('/')[0].spl

    return song_peaks
```

Code

```python
def unique_shingles(song_peaks):

    tot_shingles = list(song_peaks.values())

    shingles = []
    for i in tqdm(tot_shingles):
        shingles.append(i)

    shingles = np.hstack(shingles)
    shingles = np.array(list(dict.fromkeys(shingles))) # all unique peaks

    return shingles
```

```python
def onehot(peaks, shingles):
    return np.array([1 if x in peaks else 0 for x in shingles])


def shingles_matrix(shingles, song_peaks):

    matrix = np.zeros(len(shingles))

    for v in tqdm(list(song_peaks.values())):
        matrix = np.vstack([matrix, onehot(v, shingles)])

    matrix = np.delete(matrix, (0), axis=0)

    return matrix
```

Code

```python
def hash_matrix(matrix, shingles, song_peaks):

    # we transpose the matrix in order to have the shingles on the rows and the songs on the columns
    df = pd.DataFrame(matrix.transpose(), index = range(len(shingles)), columns = list(song_peaks.keys()))

    hash_matrix = np.zeros(len(song_peaks), dtype = int)

    # we permutate the rows of the matrix and for each column we look at the first non-zero value and store in a list
    # the corresponing raw index of that value, then by stacking the list at each permutation we get back the hash matrix
    for i in tqdm(range(nperm)):
        hash_matrix = np.vstack([hash_matrix, list(df.sample(frac = 1, random_state = i).reset_index(drop=True).ne(0).idxmax())]
        # .sample shuffles all the rows of the matrix
        # .ne(x) looks for the values different from x
        # .idxmax finds the first index between all the indexes with non-zero values

    hash_matrix = np.delete(hash_matrix, (0), axis=0)
    hash_mat = pd.DataFrame(hash_matrix, index = range(1, nperm + 1), columns = list(song_peaks.keys()))

    return hash_mat
```

Code

```
def fingerprint(query, shingles, rounded=False):
    _, _, onset_q, peaks_q = load_audio_peaks(query, OFFSET, DURATION, HOP_SIZE)

    query_oh = onehot(np.array(onset_q[peaks_q]).round(1), shingles)

    query_df = pd.DataFrame(query_oh.transpose(), index = range(len(shingles)), columns = ['query'])

    hash_query = np.zeros(1, dtype = int)
    for i in range(nperm):
        hash_query = np.vstack([hash_query, list(query_df.sample(frac = 1, random_state = i).reset_index(drop=True).ne(0).idxmax(

    hash_query = np.delete(hash_query, (0), axis=0)
    hash_query = pd.DataFrame(hash_query, index = range(nperm), columns = ['query'])

    return hash_query
```

Code

```python
def db_buckets(hash_matrix, n_bands):

    # first we have to decide a number of bands that is a divisor of the signature length in order to have equal slices of the si
    # of course the less bands we use the more discriminant the LSH will be
    rows = int(nperm/n_bands)
    buckets = {}

    for song_name, song_hash in hash_matrix.iteritems():
        song_hash = list(song_hash) # convert the columns of the dataframe from pandas series into lists

        for i in range(0, len(song_hash), rows):
            bucket_hash = tuple(song_hash[i : i + rows]) # the hash of the bucket will be a tuple with number of elements = rows

            if bucket_hash in buckets:
                buckets[bucket_hash].add(song_name) # if we already encountered that band we only add the song name
            else:
                buckets[bucket_hash] = {song_name} # otherwise we create a new key:value

    return buckets
```

Code

```python
def query_buckets(fingerprint, n_bands):

    # same as before but in this case fingerprint is a list and not a dataframe

    rows = int(len(fingerprint)/n_bands)

    # splitting the signature in nbands subvectors
    q_buckets = {}
    for i in range(0, len(fingerprint), rows):
        q_buckets[tuple(fingerprint[i : i + rows])] = 'query'

    return q_buckets
```

Code

```python
def shazamLSH(query, database, shingles, buckets):

    print('Im listening to your music, please dont make noise ...')

    score = (0, '')
    db_keys = list(database.keys())
    buckets_keys = list(buckets.keys())

    query_fingerprint = list(fingerprint(query, shingles, rounded=True)['query'])
    query_bands = query_buckets(query_fingerprint, 5)
    query_keys = list(query_bands.keys())

    # we compute the intersection between the query buckets and the database buckets
    common_bands = set(query_bands).intersection(set(buckets_keys))

    # we compute the jaccard only with the songs in the buckets of the intersection
    for band in common_bands:
        for song in buckets[band]:
            jac = sklearn.metrics.jaccard_score(query_fingerprint, database[song], average = None)
            if score < (jac.any(), song):
                score = (jac.any(), song)     # store the maximum score

    print('Maybe you were looking for this song: ', score[1], '\n-----------------------\n')
```

Result

```
buckets = db_buckets(hash_matrix, n_bands=5)
```

```
print(shazamLSH(r"D:\MOMDS-cse353\MP3-dataset\mp3s-32k\aerosmith\Aerosmith\01-Make_It.wav" , hash_matrix, shingles, buckets))
```

```
Im listening to your music, please dont make noise ...
Maybe you were looking for this song:  momds-cse353\mp3-dataset\mp3s-32k\tori amos\to venus and back-orbiting\04-glory of the 8
0 s - D:\MOMDS-cse353\MP3-dataset\mp3s-32k\tori amos\To Venus and Back-Orbiting\04-Glory Of The 80 s.wav
-----------------------
```

Thank You!