
# Big Data Assignment 1

## Topic: Locality Sensitive Hashing (LSH) on audio data

## Section: BS (DS) M

**Group Members:**

**Dawood Tanvir**

**Umm e Hani**

**Laiba Batool**

Locality Sensitive Hashing is a widely used technique for locating similar items in massive datasets.

In this assignment we implement and learn Locality Sensitive Hashing on audio data files and to detect duplicate audio files.

PART I:

First, we do preprocess on given audio files and extract MFFCs from audio files using the librosa library for MFCC's and OS library for finding path of files. Where MFCCs is commonly used for feature extraction in audio processing. After finding MFFCs we store MFCCs in a pickle file so we can easily access it because due to the large dataset it is difficult to recompute the MFCCs every time we run the pipeline and It can be time-consuming.

Here is the **following code for extracting MFCCs**:

```
import warnings
warnings.simplefilter("ignore", UserWarning)
import os
import soundfile as sf
import numpy as np
from os import walk
import librosa
import pickle
import pandas as pd
MFCC_DICTIONARY={}

for root, directories, files in os.walk(r"C:\Users\Downloads\AudioFiles\AudioFiles"):
        for filename in files:
            filepath = os.path.join(root, filename)
            X, sample_rate =  librosa.load(filepath)
            #print(len(X))
            #print(sample_rate)
            MFCC_Features = librosa.feature.mfcc(y=X, sr=sample_rate)
            print (len(MFCC_Features))
            MFCC_DICTIONARY[filename]=MFCC_Features


data_frame=pd.DataFrame.from_dict(MFCC_DICTIONARY,orient='index')
print(data_frame)
with open('AudioFiles_mfcc.pkl', 'wb') as file:
    pickle.dump(data_frame, file)

read_pickle = pd.read_pickle(r"C:\Users\Laiba Arshad\AudioFiles_mfcc.pkl")
read_pickle
```

## HASH FUNCTIONS:

In the assignment we use 10 hash functions and the size of the hash table is 23.

Here is **code for hash functions**:

```
hashfunc1=[]
temp=[]
for i in range(0,len(data)):
    for j in range(0,len(data[i])):
        b=np.mean(data[i][j])
        a=(x*b+y)%23
        temp.append(a)
    hashfunc1.append(temp)
    temp=[]

hashfunc1
```

Here we give different values of x and y for 10 different hash functions.

The **output of first hash function** look like following:

```
Out[18]: [[19.190452575683594,
          11.697776794433594,
          22.833585739135742,
          19.7772216796875,
          10.784759998321533,
          11.890010833740234,
          19.53473961353302,
          5.916703224182129,
          19.45431661605835,
          11.971983909606934,
          1.5257941037416458,
          14.823960304260254,
          16.565908670425415,
          9.753606796264648,
          4.739619255065918,
          3.9752559661865234,
          18.143887639045715,
          19.631829738616943,
          13.293524980545044,
```

## Conversions to 0 and 1:

Then convert hash functions values to zero and one. First, we multiply hash values by 100 and then set threshold of 100, if value less than threshold then set it to 1 else 0.

**Code:**

```
def chng(hashfun):
    newlist=[]
    temp=[]
    for i in range(0,len(hashfun)):
        for j in range(0, len(hashfun[i])):
            if (hashfun[i][j]*10)<100:
                temp.append(1)
            else:
                temp.append(0)
        newlist.append(temp)
        temp=[]
    return newlist
```

**Output of conversion of first hash function**:

```
h1
[[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0],
 [0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0],
 [0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1],
 [1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1],
 [1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
 [1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0],
 [1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
 [0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1],
 [0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0],
 [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
 [1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0],
 [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1],
 [0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0],
 [0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1],
 [0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0],
```

# Random Permutation:

Now we generate random permutations of integers from 0 to 19 for all hash functions. Each permutation is a random ordering of the set of possible hash

values, and we use multiple permutations to create multiple hash functions that map the input data to different buckets in the hash table. We use “np.random.permutation” for random permutation. It is a convenient function because it creates an array of unique values that are randomly ordered. Different random permutations are used to increase the probability that similar data points will be mapped to the same or nearby buckets.

**Code for permutation:**

```
ran1=np.random.permutation(20)
ran2=np.random.permutation(20)
ran3=np.random.permutation(20)
ran4=np.random.permutation(20)
ran5=np.random.permutation(20)
ran6=np.random.permutation(20)
ran7=np.random.permutation(20)
ran8=np.random.permutation(20)
ran9=np.random.permutation(20)
ran10=np.random.permutation(20)
```

**Output for random permutation 1:**

```
ran1
```

```
array([ 5,  7, 13,  1,  9,  3, 12,  6,  4, 15,  2,  0, 19, 18, 11, 14, 16,
       10, 17,  8])
```

# BUCKETS:

Now we  make different buckets. Buckets are used in locality-sensitive hashing to store the hash values of the input data in a way that allows for efficient nearest neighbor search. LSH can speed up and improve nearest neighbor search by

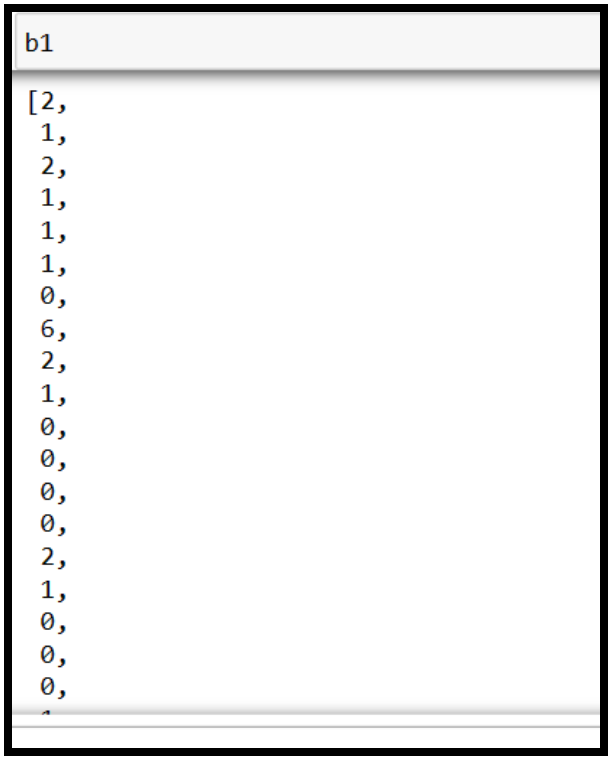
condensing the search space for related data points into the same or nearby hash buckets.

**Code:**

```
def bucket(hashfun, per):
    buc=[]
    minbuc=[]
    for i in range(0,len(hashfun)):
        for j in range(0,len(hashfun[i])):
            if hashfun[i][j]==1:
                buc.append(per[j])
        a=min(buc)
        minbuc.append(a)
        buc=[]
    return minbuc
```

```
b1=bucket(h1,ran1)
b2=bucket(h2,ran2)
b3=bucket(h3,ran3)
b4=bucket(h4,ran4)
b5=bucket(h5,ran5)
b6=bucket(h6,ran6)
b7=bucket(h7,ran7)
b8=bucket(h8,ran8)
b9=bucket(h9,ran9)
b10=bucket(h10,ran10)
```

**Output for bucket 1:**




## Partial Sums:

We compute partial sums for locality sensitive hashing. Partial sums use to accelerate the calculation of hash values for the input data. This can be more efficient than computing the hash values directly for each column, because we can pre compute the sums for each row of the input data matrix, which can be done more efficiently than computing the hash values directly for each column. However, it can introduce sum error because we are approximating the hash values based on the partial sums,rather than computing directly.

**Code:**

```
temp=[]
parsumlist=[]
for i in range(0,len(b1)):
    a=b1[i]+b2[i]+b3[i]+b4[i]+b5[i]
    b=b6[i]+b7[i]+b8[i]+b9[i]+b10[i]
    temp.append(a)
    temp.append(b)
    parsumlist.append(temp)
    temp=[]
```

```
audname=[]
for i in newMfcc.index:
    audname.append(i)
```

```
audict1=defaultdict(list)
audict2=defaultdict(list)
```

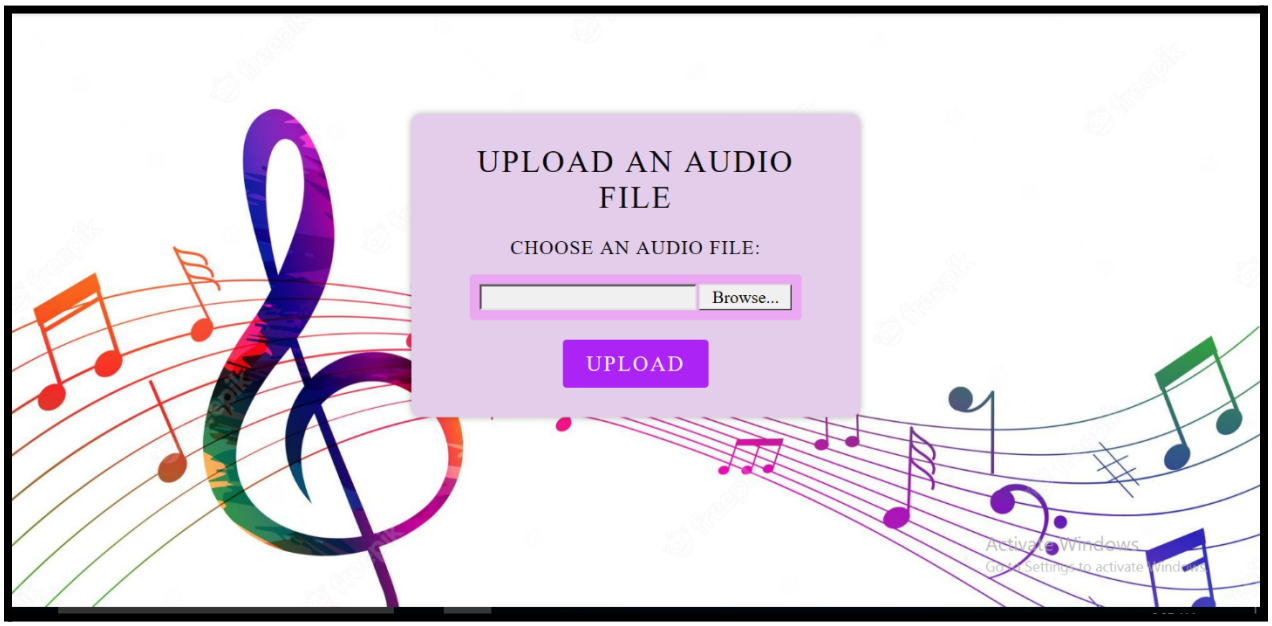
```
for i in range(0,len(parsumlist)):
    audict1[parsumlist[i][0]].append(audname[i])
    audict2[parsumlist[i][1]].append(audname[i])
```

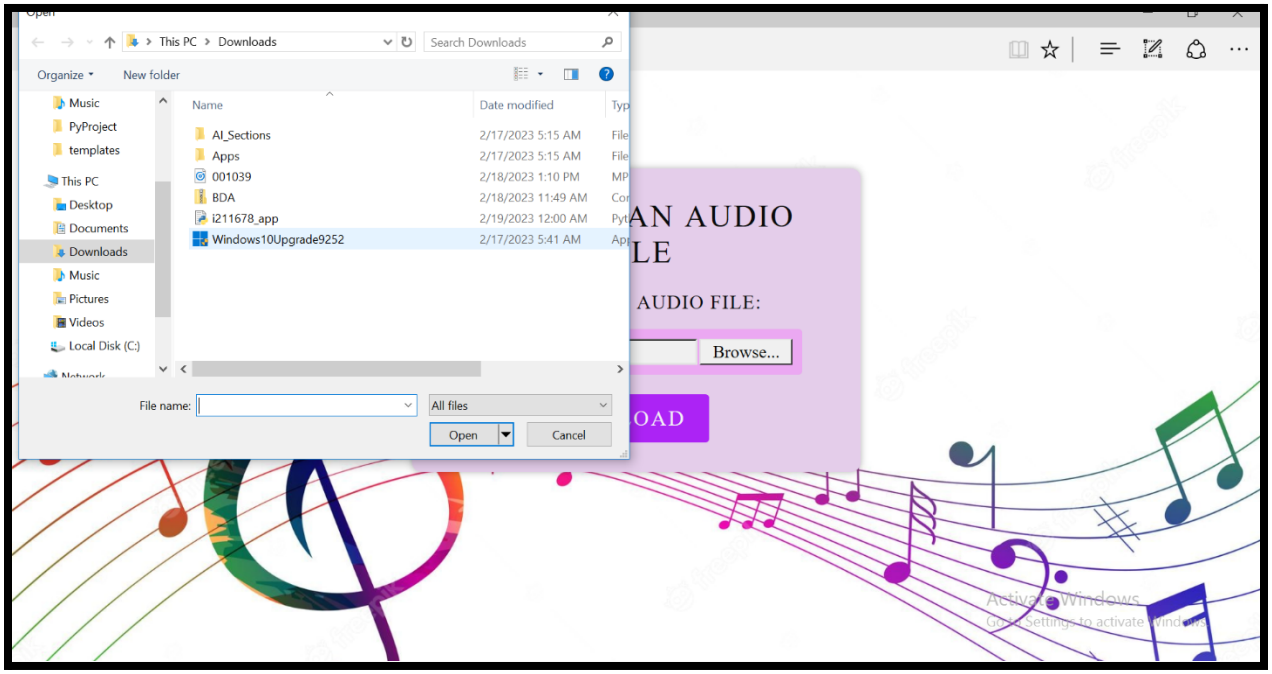
```
audmfc=defaultdict(list)
for i in range(0,len(parsumlist)):
    audmfc[audname[i]].append(parsumlist[i])
```

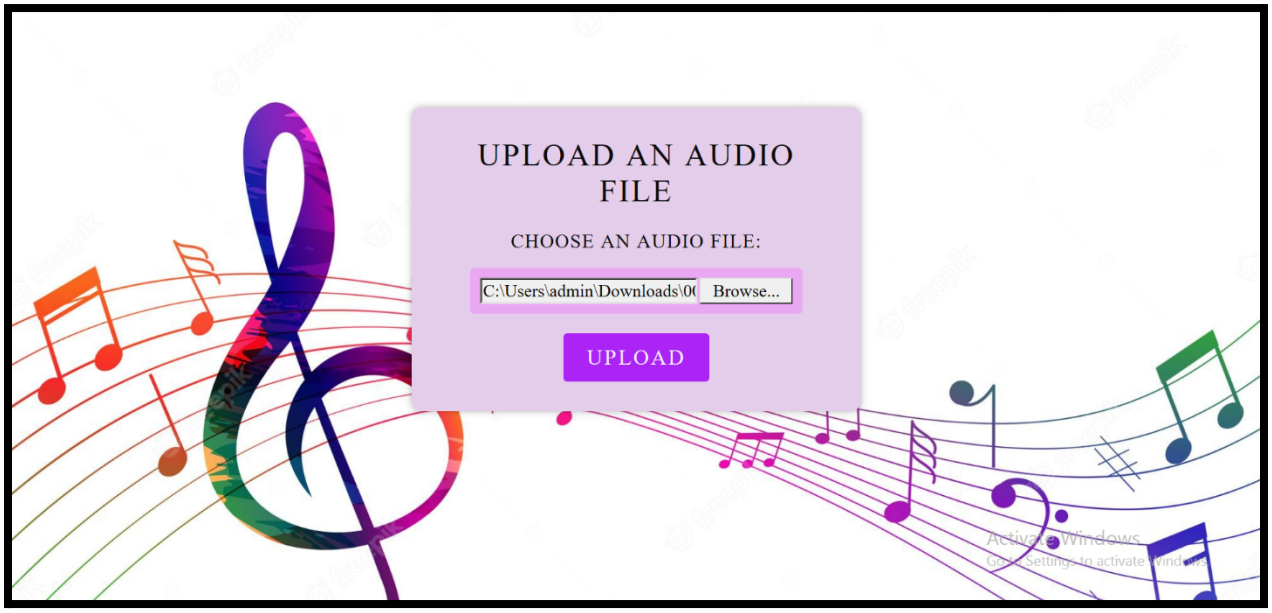
## PART II:

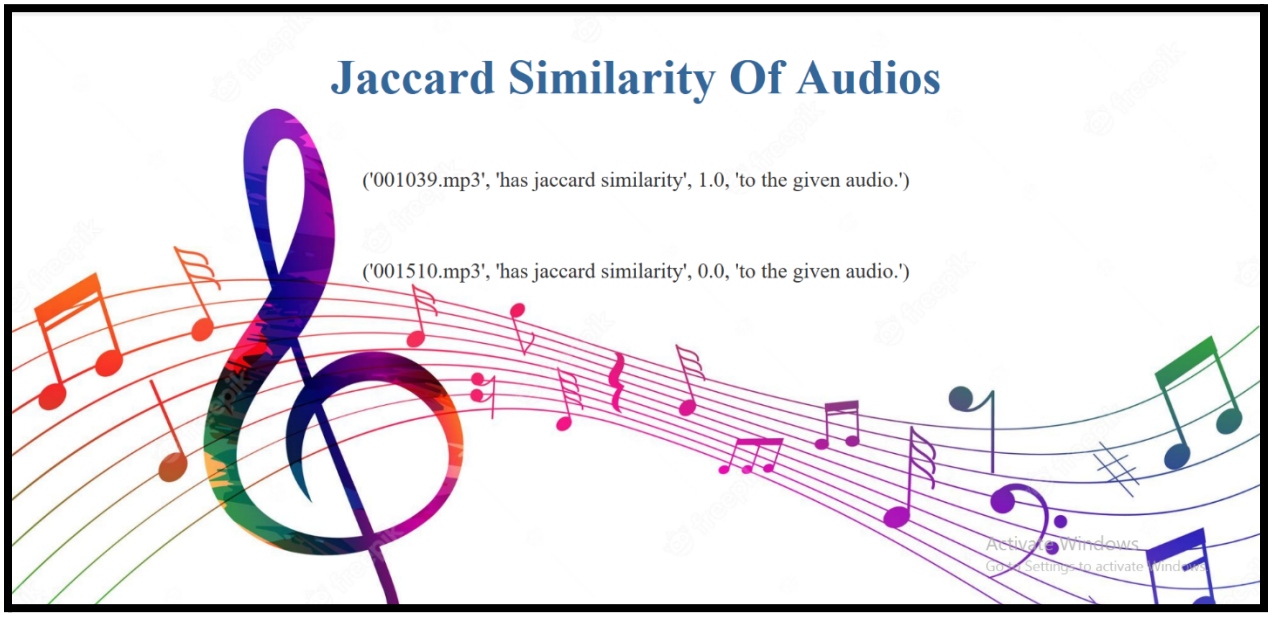
For part II we are required to create a flask application for the implemented code.First, we create flask and add necessary packages for audio processing, then we create an HTML page that allows the user to upload mp3 audio files. After uploading the file we use the same technique which we use for part I that is we extract MFCCs of audio and create hash functions and convert to 0 and 1 format. Then make buckets and partial sums of audio files. After this we check similarity on the basis of partial sums and bucket data.

# IMAGES TO UPLOAD AUDIO FILE:




UPLOAD AN AUDIO FILE
CHOOSE AN AUDIO FILE:
C:\Users\admin\Downloads\0(
Browse...
UPLOAD


Jaccard Similarity Of Audios
('001039.mp3', 'has jaccard similarity', 1.0, 'to the given audio.')
('001510.mp3', 'has jaccard similarity', 0.0, 'to the given audio.')