Music Retrieval by Rhythmic Similarity with Locality Sensitive Hashing

Yenchih Andrew Tang

Philip Cori

March 17, 2020

Acknowledgements

We would like to thank Dr. Wang for teaching us the important concepts of Data Mining necessary to understand and solve the problem described in this paper. Without our class resources, designing our algorithm would have been impossible. We would also like to thank the researchers that helped us get started with this project who provided a baseline for our implementation, namely Jonathan Foote, Matthew Cooper, and Unjung Nam for publishing the paper *Audio Retrieval by Rhythmic Similarity*. Additional thanks to Kay Zhu for giving us a running start on LSH implementation

1. Table of Contents

2. Introduction

2.1 What is the problem

Audio retrieval is one of the most complex information retrieval problems due to the numerous features and representations of audio. Even more complex is the problem of retrieving similar audio. What makes one song more similar to one than the other? Even a human cannot answer this question easily, much less a computer. There are many features of a song one could identify and use to find similar songs, such as the key, beats per minute, lyrics, energy, or chord progressions. Therefore, we've decided to narrow the scope of our project to querying music based on *rhythm*. Rhythm refers to the count of, or time relationship between, the main notes and melodies of the song. Another problem that exists is the inefficiency of current algorithms. Finding the top K most similar songs traditionally requires querying the entire database and sorting the entries based on similarity. Clearly this is not scalable and requires a solution to solve this.

2.2 Why this project is related to this class

Any type of information retrieval problem is within the scope of data mining. Thus, audio-based information retrieval is clearly still within the scope of data mining. This problem deals with efficiently querying items in a database using LSH, a main area within the field of data mining.

2.3 Area or Scope of Investigation

As mentioned, we restrict our research to querying audio based on the query rhythm. Rhythm can be extracted from audio using its spectrogram representation, a similarity matrix, and ultimately a beat spectrum, which can allow for the application of finding songs with a similar rhythm to a certain query song (Foote, Cooper, and Nam, 2002).

3. Theoretical Bases

3.1 Definition of the problem

The problem with music information retrieval is the complexity of audio. Audio can be represented in a multitude of ways that makes it unclear what features to be prioritized during querying. For the sake of this project, as mentioned, we narrow the scope to the *rhythm* of the song and use this as the only query feature. Research has been done on rhythm-based music querying. However, the most recent methods are inefficient, in that they query the entire data structure for song rhythms. We introduce LSH to solve this problem.

3.2 Related Research to Solve the Problem

We chose to compare our original algorithm with the traditional algorithm presented in *Audio Retrieval by Rhythmic Similarity* (Foote, Cooper, and Nam, 2002). Details about this algorithm are described in section 5.

3.2 Solution to Solve This Problem

Our solution to the problem of inefficiency is to take advantage of using Locality Sensitive Hashing (LSH). Such an algorithm can be helpful for finding songs that are similar based on a certain feature, in our case rhythm, as well as other features, in order to help people discover new yet similar music. LSH allows for extremely efficient querying speeds, having the entries stored in a hash table based on the inter-song rhythm similarity. Currently, LSH is not being used in mainstream algorithms for music retrieval and can significantly improve efficiency while maintaining accuracy in song retrieval. Rhythm-based audio querying has been researched

in the past and we will build upon previous algorithms to develop our own custom LSH algorithm. Previous implementations extract audio and puts them into a high dimensional vector form to be hashed. However, by focusing on just the rhythm of the song, we believe using the beat spectrum rhythm representation, along with LSH, could allow for improved performance.

3.3 Where Our Solution Differs

Our solution is the first of its kind; other solutions have neglected considering the efficiency of their algorithms by querying the entire rhythm vector (the beat spectrum) of the song. Rather than store and query every vector for each song, we use LSH to conduct highly efficient querying to find songs with similar to identical rhythm. We are the first to apply LSH to rhythm-based music retrieval, however LSH has been used in other applications for improving query performance.

3.4 Why is it Better?

Assuming our solution can maintain a reasonably high accuracy in retrieving songs with similar rhythms, our solution is better in its efficiency. LSH has the advantage of comparing low-dimensional hash keys, which is far faster than comparing the actual rhythm data structures such as the high-dimensional beat spectra. Additionally, LSH allows us to store our database of songs in buckets stored in a hash table. This allows for retrieval that runs in O(1) time, rather than O(nlogn), due to sorting the list, as in the original algorithm.

4. Hypothesis

Our positive hypothesis is that using LSH for song retrieval will drastically improve efficiency when finding similar songs. The inclusion of LSH also would improve efficiency because it acts as a dimensionality reduction. However, since we're only focusing on one aspect of the music, a negative hypothesis is that the results are going to be more accurate. That is, we do not expect LSH to improve the algorithm's ability to accurately retrieve rhythmically similar songs over the traditional method that compares the full beat spectra. This is a known trade-off of using LSH that we anticipate to see in our results.

5. Methodology

5.1 How to generate/collect input data

We use the Python package Librosa to extract audio for WAV audio files. The music dataset we will use will be short extracts from various publicly available music. We formed our music dataset from samples from www.bensound.com. Our dataset consisted of 15 song samples taken from this source.

5.2 How to solve the problem

5.2.1 Algorithm Design

We will use build on algorithms previously developed, namely that described in *Audio Retrieval by Rhythmic Similarity* (Foote, Cooper, and Nam, 2002). There are four distance measures in evaluating rhythmic similarity. Chen & Chen discuss using Hamming distance. Foote, Cooper, and Nam conduct a comparative analysis on using Euclidean and Cosine distance, as well as Fourier Beat Spectral Coefficients. We will experiment with cosine distance, as this was shown to have the best results in the experiments conducted by Foote, Cooper, and Nam. We will be evaluating the following metrics in our experiments: Query accuracy and time efficiency. Our experiment will be conducted as follows:

1) Implement the traditional algorithm described by Foote, Cooper, and Nam and compute its query accuracy and time efficiency on our dataset.
2) Implement our custom LSH algorithm and compare its query accuracy and time efficiency to the traditional algorithm on the same dataset.

The traditional algorithm goes as follows:

1) Divide each song into 1 + k 10 second excerpts. The first serves as the query and the last k serve as the targets of the query and are stored in a database with their rhythm representations. We assume excerpts from the same song are most rhythmically similar.
2) For each query, find the nearest k songs based on the cosine similarity of the beat spectra of the song excerpts. This is done by comparing with every beat spectrum in the database, sorting the songs, and taking the top k samples. Accuracy is equal to # of correct retrievals / total # of retrievals. A retrieval is correct if the retrieved sample is from the same song as the query sample.
3) To create the beat spectra of an audio sample:
    a) Compute the mel spectrogram using a Fast Fourier Transform

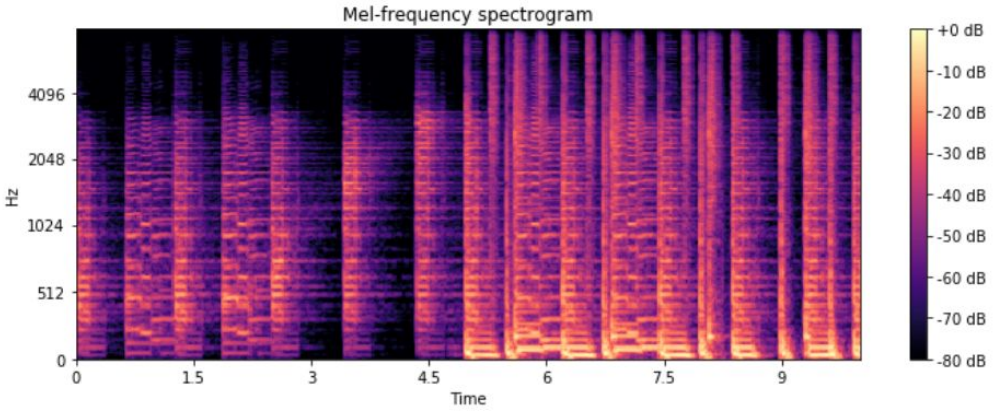

Figure 1: Mel Spectrogram

    b) Compute the similarity matrix S by finding frame-to-frame similarity of the spectrogram vectors. These vectors are F dimensional, where F is the frequency range and the values represent the amplitude at these frequencies. It can be seen

that the center diagonal has the highest similarity because the same time frame is going to be most similar to itself. Other points in the matrix, say (100, 50), represent the spectrogram similarity between the 100th time frame and the 50th time frame. Note the axes in Figure 2 represent time frames. Each time frame represents 10 / 256 = .04 seconds.

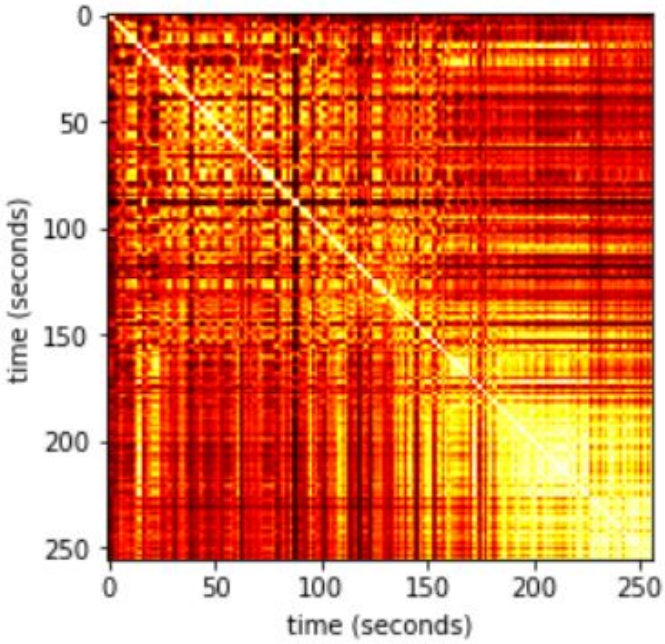

Figure 2: Song Similarity Matrix

c) Compute the beat spectrum by summing the diagonals of S.

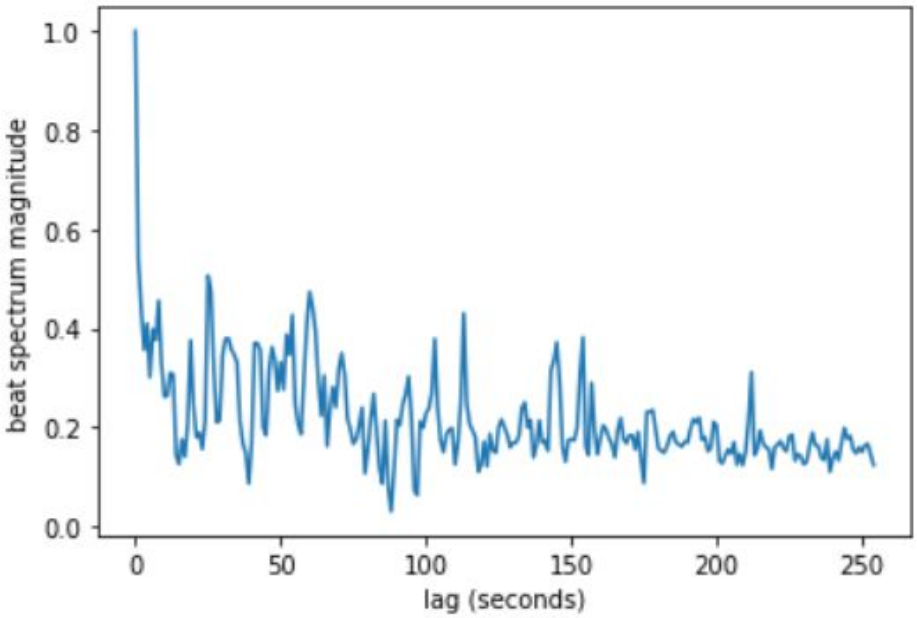

Figure 3: Song Beat Spectrum

Again, the x-axis represents the time frames of the 10 second song sample. Here, peaks represent repetitions with a lag of the x value of the peak. For example, if a saxophone C# note is played at time = 0s and again at time = 1.5s, a peak will appear at 1.5s. This same peak would appear in the beat spectrum of another sample if, for example, a piano F# note were played at 3.1s and 4.6s. It can be seen that the beat spectrum considers relative frame similarities rather than absolute frame similarities, taking into account only the time difference rather than the actual times themselves. Intuitively, it can be seen how this vector can represent the rhythm of a song. More details of this algorithm are described by Foote, Cooper, and Nam in their paper. Figure 4 shows the beat spectra of two separate 10 second samples from the song “All That”. It can be seen they are very similar and would yield a high cosine similarity rating.

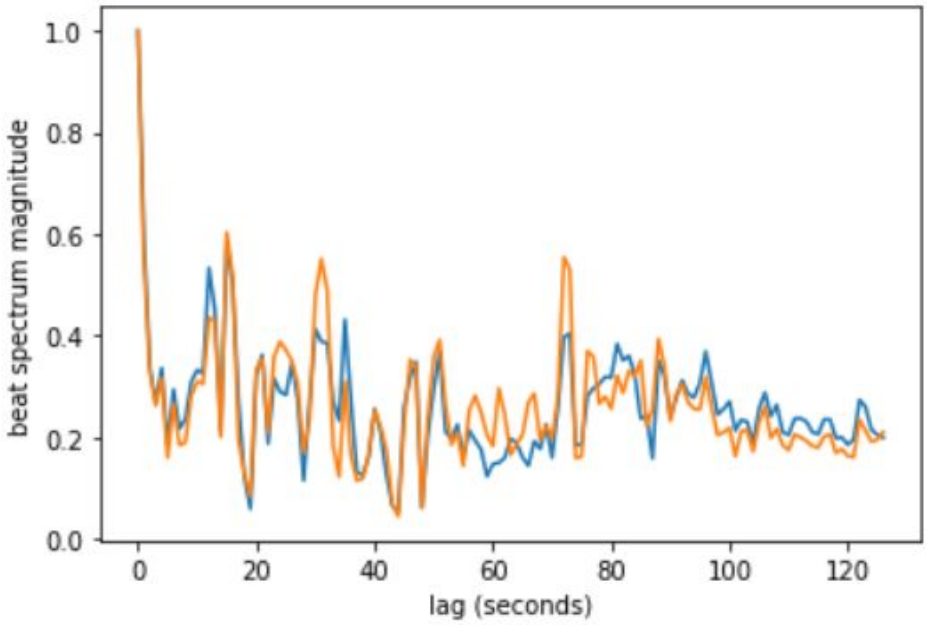

Figure 4: Beat Spectra of Two Samples From Same Song

Our LSH algorithm goes as follows:

1) For each corpus song excerpt:
    a) Compute the beat spectrum using the same method as the traditional algorithm mentioned before.
    b) Using the random hyperplanes method, compute a cosine-similarity sensitive hash of the beat spectrum vector. The beat spectrum is represented by a vector of 127 of 64-bit numpy float. Each hyperplane will therefore also be a vector of 127 floats, each randomly generated by a normal distribution with a mean of 0 and variance of 1. The inner product of these two vectors is then taken. If the result is > 0, the bit of the hash will be 1, and if the result is < 0, the bit of the hash will be

0. This process is repeated for the designated number of bits desired for the binary hash.

c) Insert the SongEntry into the bucket with key = computed hash. This hashtable will serve as our song database from which queries will be checked over.

2) For each query:

a) The structure of the query for this experiment will be similar to the traditional algorithm in which two 10 second intervals will be taken from each of the 15 songs to generate the hashtable. Then, a third 10 second interval will be used to query the hashtable with the intent to land in the same bucket with the other 10 second intervals. This allows for a more direct comparison of the performance between the LSH and

b) Compute the beat spectrum and its hash using the same random hyperplane as mentioned before.

c) Hashing the query into a bucket, the bucket is then searched to find the song in the bucket. The accuracy is determined by how many song intervals are in the resulting bucket. The total runtime of this process is also recorded.

5.2.2 Languages and Tools Used

This project is programmed in Python. We chose this language primarily for the breadth of resources publicly available, especially when it comes to data extraction from audio files. As previously mentioned, the Python package LibROSA is used for music analysis and feature extraction such as the spectrogram. Steps such as computing the similarity matrix and

beat spectrum were implemented manually through the descriptions of Foote, Cooper, and Nam. Other packages used include Numpy and Scipy for their mathematical and vector calculation capabilities.

6. Implementation

We deeply analyzed the algorithm described by Foote, Cooper, and Nam and were able to reproduce the algorithm in Python. Using our database of 15 songs, we implemented the algorithm described in section 5.2.1. We do this by using LibROSA to extract audio from the WAV files in 10 second durations. For each excerpt, we then compute the beat spectra and store the database of songs as a list of SongEntry objects. Each SongEntry contains the song ID (note we do not care about the actual excerpt ID), the audio time series, sampling rate, and beat spectrum. Once the database has been constructed, we iterate through the queries and retrieve the top k most rhythmically similar songs using a nearest k approach. We then evaluate the performance of the algorithm by evaluating the accuracy and time efficiency. This same pattern is used for both our and the traditional algorithm. Our algorithm was built off a template of fast implementation of LSH by Zhu. The template was modified to accommodate to take in songs to build the database and queries as well as work with the beat spectrum.

7. Data Analysis and Generation

We decided to use 15 songs for a few reasons. Firstly, Foote, Cooper, and Nam originally used 15 songs as well in their experiments, showing that this can be sufficient to get meaningful results. Secondly, due to limitations in RAM and computing speed, using a large database of songs is infeasible. Constructing the database of only 15 songs takes approximately 1 minute. Lastly, there is limited availability of free songs online, making it difficult to curate a large dataset in the first place.

Our implementation of the traditional algorithm was able to achieve an accuracy of 83% with k = 2, where k is the number of targets per query. The findings of Foote, Cooper, and Nam originally achieved 97%, however this could be due to handpicking their song samples and ensuring rhythmic similarity manually, doing extra pre-computing on the song samples, or having more time to fine-tune parameters. However, we can still use this as a benchmark to compare our LSH algorithm to. The average time elapsed per query is 0.0019s.

The LSH algorithm resulted in an average accuracy of 62.5% over the course of 10 runs. However, the accuracy for each trial was highly variable, going as high as 93% or as low as 41%. This is most likely the result of an oversimplified LSH algorithm. The randomly generated hyperplane logically creates variable results as binary hashes can change greatly in value and number depending on the generation. A more accurate method of LSH would be to use multiple hashtables and then searching through each of them. However, since the goal is to maximize performance, a simpler implementation was done instead.

Our hypothesis proved to be correct. Efficiency increased dramatically from an average of .0019s to .0004s, taking approximately only 1/5th of the original time. However, as mentioned, accuracy proved to decrease due to our simple LSH algorithm.

8. Conclusion and Future Works

8.1 Summary and Conclusions

The results of our experiment prove that LSH can be applied to information retrieval tasks. Assuming we had more time, our LSH algorithm could be fine-tuned to our application to provide accuracy similar to the traditional algorithm. Ultimately, the results are promising for applying LSH in the domain of audio retrieval based on rhythmic similarity.

8.2 Recommendations for Future Studies

This is very much an exploratory analysis of the topic. Future studies on the application of LSH for rhythmic similarity can look at various other more developed LSH algorithms. Through this research, many further routes of experiment are very clear. The LSH algorithm could utilize more hash tables, improving the accuracy of the random generation. The current implementation also currently outputs nothing if the query hashes to an unexisting bucket. Due to the nature of LSH, the closest bucket can be found. There are many further optimizations possible. The ideal goal would be to find a balance between accuracy and performance.

9. Bibliography

Chen, J., & Chen, A. (1998). Query by rhythm: an approach for song retrieval in music databases. *Proceedings Eighth International Workshop on Research Issues in Data Engineering. Continuous-Media Databases and Applications*. doi: 10.1109/ride.1998.658288

Foote, Jonathan & Cooper, Matthew & Nam, Unjung. (2002). Audio Retrieval by Rhythmic Similarity..

Zhu, Kay Fast Implementation of LSH. https://github.com/kayzhu/LSHash

10. Appendix

All code can also be found at https://github.com/philipcori/RhythmSearch

main.ipynb:

```
# In[1]:


import librosa
import librosa.display
import os
import matplotlib.pyplot as plt
import numpy as np
import time
from scipy.spatial.distance import cosine
from scipy.spatial.distance import euclidean


# In[2]:


def sim_matrix(spectrogram):
    num_frames = spectrogram.shape[0]
#     print("Number of frames: " + str(num_frames))
    S = np.ndarray(shape=(num_frames, num_frames))
    for i in range(num_frames):
        for j in range(num_frames):
            S[i, j] = 1 - cosine(spectrogram[i], spectrogram[j])
    return S


# In[3]:


def beat_spectrum(y, sr, S):
    # beat_spectrum = np.correlate(S[0], S[0], mode='full')
    # beat_spectrum = beat_spectrum[:len(beat_spectrum) // 2]
    duration = librosa.core.get_duration(y, sr)
    frames_per_sec = S.shape[0] / duration
    lag_range = duration
    num_frames = int(lag_range * frames_per_sec) - 1
#     print(num_frames)
    bs = []
    for l in range(num_frames):
        sum = np.sum([S[i, i + l] for i in range(len(S) - num_frames)])
#         sum /= (len(S) - num_frames)
        bs.append(sum)
#     mean = np.mean(bs)
#     bs -= mean
    return bs
```

```
# In[4]:


def robust_beat_spectrum(y, sr, S):
    duration = librosa.core.get_duration(y, sr)
    frames_per_sec = S.shape[0] / duration
    lag_range = duration
    num_frames = int(lag_range * frames_per_sec) - 1
    bs = []
    print(num_frames)
    for l in range(num_frames):
        sum = 0
        for i in range(num_frames):
            for j in range(num_frames - l):
                sum += S[i][j] * S[i][j + l]
        print(sum)
        bs.append(sum)
    return bs


# In[5]:


def plot_spectrogram(spectrogram):
    plt.figure(figsize=(10, 4))
    S_dB = librosa.power_to_db(spectrogram, ref=np.max)
    librosa.display.specshow(S_dB, x_axis='time', y_axis='mel', sr=sr, fmax=8000)
    plt.colorbar(format='%+2.0f dB')
    plt.title('Mel-frequency spectrogram')
    plt.tight_layout()
    plt.show()


# In[6]:


class SongEntry:
    def __init__(self, y, sr, id):
        self.y = y
        self.sr = sr
        self.id = id
        self.init_beat_spectrum()

    def init_beat_spectrum(self):
        spectrogram = librosa.feature.melspectrogram(y=self.y, sr=self.sr, n_mels=128)
        S = sim_matrix(spectrogram)
        bs = beat_spectrum(y=self.y, sr=self.sr, S=S)
        self.bs = bs

    def cos_sim(self, songEntry):
        return 1 - cosine(self.bs, songEntry.bs)


# In[7]:
```

```
def retrieveNearestK(query, db, k):
    sims = list()
    for entry in db:
        sim = query.cos_sim(entry)
        sims.append((entry.id, sim))
    sims.sort(key=lambda tup: tup[1], reverse=True)
    neighbors = list()
    for i in range(k):
        neighbors.append(sims[i][0])
    return neighbors


# In[8]:


def evaluate(db, queries, k): #lists of SongEntry's
    num_correct = 0
    time_sum = 0
    for query in queries:
        start = time.time()
        nearestK = retrieveNearestK(query=query, db=db, k=k)
        end = time.time()
        time_sum += (end - start)
        for id in nearestK:
            if (id == query.id):
                num_correct += 1
#             else:
#                 print('query id: ' + str(query.id) + ', db id: ' + str(id))
    total = len(queries) * k
    accuracy = num_correct / total
    avg_time_elapsed = time_sum / len(queries)
    return accuracy, avg_time_elapsed


# In[9]:


data_dir = os.path.join(os.getcwd(), '..\\data\\')
data_files = [os.path.join(data_dir, f) for f in os.listdir(data_dir)]
songDB = []
queries = []
songs_per_query = 2
initial_offset = 5
for j, data_file in enumerate(data_files):
    print('loading song ' + str(j) + ' of ' + str(len(data_files)))
    y, sr = librosa.load(data_file, duration=10, offset=initial_offset)
    a = SongEntry(y, sr, data_file)
    queries.append(a)
    for i in range(1, 1 + songs_per_query):
        y, sr = librosa.load(data_file, duration=10, offset=initial_offset+i*10)
        a = SongEntry(y, sr, data_file)
        songDB.append(a)
print(str(len(queries)) + ' query samples and ' + str(len(songDB)) + ' DB samples saved')


# In[27]:
```

```
accuracy, avg_time_elapsed = evaluate(db=songDB, queries=queries, k=songs_per_query)
print('Accuracy: ' + str(accuracy) + ', Average Time Elapsed: ' + str(avg_time_elapsed))

# In[11]:


from lshash import LSHash
import random


# In[12]:


def LSHSetup(db, songs_per_query, num_bit):
    songDB = []
    queries = []
    hash = LSHash(num_bit,127)

    for song in db:
        hash.index(song) #Adds to LSH hashtable

    print("All Keys in Hashtable")
    print(list(hash.hash_tables[0].storage.keys()))
    return hash


# In[13]:


def evaluateLSH(db, hash, queries, k):
    num_correct = 0
    time_sum = 0
    for query in queries:
        start = time.clock()
        result = hash.query(query)
        end = time.clock()
        time_sum += end - start
        for output in result:
            if (output[0] == query.id):
                num_correct += 1
            #else:
                #print('query id: ' + str(query.id) + ', db id: ' + str(output[0]))
    total = len(queries) * k
    accuracy = num_correct / total
    avg_time_elapsed = time_sum / len(queries)
    return accuracy, avg_time_elapsed


# In[22]:


N = 10 # Number of Iterations
total_accuracy = 0
total_time = 0

for i in range(N):
```

```
    print("SETTING UP HASHTABLE")
    hash = LSHSetup(songDB, songs_per_query=2, num_bit=4) # 4 bit binary hash
    print("EVALUATING")
    accuracy, avg_time_elapsed = evaluateLSH(db=songDB, hash=hash, queries=queries, k=2)
    total_accuracy += accuracy
    total_time += avg_time_elapsed
    print('Accuracy: ' + str(accuracy) + ', Average Time Elapsed: ' + str(avg_time_elapsed))

print("FINAL RESULTS")
print('Average Accuracy over ' + str(N) + ' iterations: ' + str(total_accuracy/N))
print('Average Time Elapsed over ' + str(N) + ' iterations: ' + str(total_time/N))
```

lshash.py

```
# lshash/lshash.py
# Copyright 2012 Kay Zhu (a.k.a He Zhu) and contributors (see CONTRIBUTORS.txt)
#
# This module is part of lshash and is released under
# the MIT License: http://www.opensource.org/licenses/mit-license.php

import os
import json
import numpy as np

from storage import storage

try:
    from bitarray import bitarray
except ImportError:
    bitarray = None

class SongEntry:
    def __init__(self, y, sr, id):
        self.y = y
        self.sr = sr
        self.id = id
        self.init_beat_spectrum()

    def init_beat_spectrum(self):
        spectrogram = librosa.feature.melspectrogram(y=self.y, sr=self.sr, n_mels=128)
        S = sim_matrix(spectrogram)
        bs = beat_spectrum(y=self.y, sr=self.sr, S=S)
        self.bs = bs

    def cos_sim(self, songEntry):
        return 1 - cosine(self.bs, songEntry.bs)

class LSHash(object):
    """ LSHash implments locality sensitive hashing using random projection for
    input vectors of dimension `input_dim`.

    Attributes:

    :param hash_size:
        The length of the resulting binary hash in integer. E.g., 32 means the
        resulting binary hash will be 32-bit long.
```

```
:param input_dim:
    The dimension of the input vector. E.g., a grey-scale picture of 30x30
    pixels will have an input dimension of 900.
:param num_hashtables:
    (optional) The number of hash tables used for multiple lookups.
:param storage_config:
    (optional) A dictionary of the form `{backend_name: config}` where
    `backend_name` is the either `dict` or `redis`, and `config` is the
    configuration used by the backend. For `redis` it should be in the
    format of `{"redis": {"host": hostname, "port": port_num}}`, where
    `hostname` is normally `localhost` and `port` is normally 6379.
:param matrices_filename:
    (optional) Specify the path to the compressed numpy file ending with
    extension `.npz`, where the uniform random planes are stored, or to be
    stored if the file does not exist yet.
:param overwrite:
    (optional) Whether to overwrite the matrices file if it already exist
"""

def __init__(self, hash_size, input_dim, num_hashtables=1,
             storage_config=None, matrices_filename=None, overwrite=False):

    self.hash_size = hash_size
    self.input_dim = input_dim
    self.num_hashtables = num_hashtables

    if storage_config is None:
        storage_config = {'dict': None}
    self.storage_config = storage_config

    if matrices_filename and not matrices_filename.endswith('.npz'):
        raise ValueError("The specified file name must end with .npz")
    self.matrices_filename = matrices_filename
    self.overwrite = overwrite

    self._init_uniform_planes()
    self._init_hashtables()

def _init_uniform_planes(self):
    """ Initialize uniform planes used to calculate the hashes

    if file `self.matrices_filename` exist and `self.overwrite` is
    selected, save the uniform planes to the specified file.

    if file `self.matrices_filename` exist and `self.overwrite` is not
    selected, load the matrix with `np.load`.

    if file `self.matrices_filename` does not exist and regardless of
    `self.overwrite`, only set `self.uniform_planes`.
    """

    if "uniform_planes" in self.__dict__:
        return

    if self.matrices_filename:
        file_exist = os.path.isfile(self.matrices_filename)
        if file_exist and not self.overwrite:
```

```
            try:
                npzfiles = np.load(self.matrices_filename)
            except IOError:
                print("Cannot load specified file as a numpy array")
                raise
            else:
                npzfiles = sorted(npzfiles.items(), key=lambda x: x[0])
                self.uniform_planes = [t[1] for t in npzfiles]
        else:
            self.uniform_planes = [self._generate_uniform_planes()
                                   for _ in range(self.num_hashtables)]
            try:
                np.savez_compressed(self.matrices_filename,
                                    *self.uniform_planes)
            except IOError:
                print("IOError when saving matrices to specificed path")
                raise
    else:
        self.uniform_planes = [self._generate_uniform_planes()
                               for _ in range(self.num_hashtables)]

def _init_hashtables(self):
    """ Initialize the hash tables such that each record will be in the
    form of "[storage1, storage2, ...]" """

    self.hash_tables = [storage(self.storage_config, i)
                        for i in range(self.num_hashtables)]

def _generate_uniform_planes(self):
    """ Generate uniformly distributed hyperplanes and return it as a 2D
    numpy array.
    """

    return np.random.randn(self.hash_size, self.input_dim)

def _hash(self, planes, input_point):
    """ Generates the binary hash for `input_point` and returns it.

    :param planes:
        The planes are random uniform planes with a dimension of
        `hash_size` * `input_dim`.
    :param input_point:
        A Python tuple or list object that contains only numbers.
        The dimension needs to be 1 * `input_dim`.
    """

    try:
        input_point = np.array(input_point)  # for faster dot product
        projections = np.dot(planes, input_point)
    except TypeError as e:
        print("""The input point needs to be an array-like object with
              numbers only elements""")
        raise
    except ValueError as e:
        print("""The input point needs to be of the same dimension as
              `input_dim` when initializing this LSHash instance""", e)
        raise
```

```
        else:
            return "".join(['1' if i > 0 else '0' for i in projections])

    def _as_np_array(self, json_or_tuple):
        """ Takes either a JSON-serialized data structure or a tuple that has
        the original input points stored, and returns the original input point
        in numpy array format.

        if isinstance(json_or_tuple, basestring):
            # JSON-serialized in the case of Redis
            try:
                # Return the point stored as list, without the extra data
                tuples = json.loads(json_or_tuple)[0]
            except TypeError:
                print("The value stored is not JSON-serilizable")
                raise
        else:
            # If extra_data exists, `tuples` is the entire
            # (point:tuple, extra_data). Otherwise (i.e., extra_data=None),
            # return the point stored as a tuple
        """
        tuples = json_or_tuple

        if isinstance(tuples[0], tuple):
            # in this case extra data exists
            return np.asarray(tuples[0])

        elif isinstance(tuples, (tuple, list)):
            try:
                return np.asarray(tuples)
            except ValueError as e:
                print("The input needs to be an array-like object", e)
                raise
        else:
            raise TypeError("query data is not supported")

    def index(self, input, extra_data=None):
        """ Index a single input point by adding it to the selected storage.

        If `extra_data` is provided, it will become the value of the dictionary
        {input_point: extra_data}, which in turn will become the value of the
        hash table. `extra_data` needs to be JSON serializable if in-memory
        dict is not used as storage.

        param input_point:
            A list, or tuple, or numpy ndarray object that contains numbers
            only. The dimension needs to be 1 * `input_dim`.
            This object will be converted to Python tuple and stored in the
            selected storage.
        param extra_data:
            (optional) Needs to be a JSON-serializable object: list, dicts and
            basic types such as strings and integers.
        """
        #song = SongEntry(y, sr, id)
        input_point = input.bs

        if isinstance(input_point, np.ndarray):
```

```
            input_point = input_point.tolist()

        if extra_data:
            value = (tuple(input_point), extra_data)
        else:
            value = tuple(input_point)

        for i, table in enumerate(self.hash_tables):
            table.append_val(self._hash(self.uniform_planes[i], input_point),
             #                value)
                            input)

    def query(self, query, num_results=None, distance_func=None):
        """ Takes `query_point` which is either a tuple or a list of numbers,
        returns `num_results` of results as a list of tuples that are ranked
        based on the supplied metric function `distance_func`.

        param query_point:
            A list, or tuple, or numpy ndarray that only contains numbers.
            The dimension needs to be 1 * `input_dim`.
            Used by :meth:`._hash`.
        param num_results:
            (optional) Integer, specifies the max amount of results to be
            returned. If not specified all candidates will be returned as a
            list in ranked order.
        param distance_func:
            (optional) The distance function to be used. Currently it needs to
            be one of ("hamming", "euclidean", "true_euclidean",
            "centred_euclidean", "cosine", "l1norm"). By default "euclidean"
            will used.
        """

        candidates = set()
        binary_hash = ''
        query_point = query.bs


        if not distance_func:
            distance_func = "euclidean"

        if distance_func == "hamming":
            if not bitarray:
                raise ImportError(" Bitarray is required for hamming distance")

            for i, table in enumerate(self.hash_tables):
                binary_hash = self._hash(self.uniform_planes[i], query_point)
                for key in table.keys():
                    distance = LSHash.hamming_dist(key, binary_hash)
                    if distance < 2:
                        candidates.update(table.get_list(key))

            d_func = LSHash.euclidean_dist_square

        else:
            if distance_func == "euclidean":
                d_func = LSHash.euclidean_dist_square
            elif distance_func == "true_euclidean":
```

```
                d_func = LSHash.euclidean_dist
            elif distance_func == "centred_euclidean":
                d_func = LSHash.euclidean_dist_centred
            elif distance_func == "cosine":
                d_func = LSHash.cosine_dist
            elif distance_func == "l1norm":
                d_func = LSHash.l1norm_dist
            else:
                raise ValueError("The distance function name is invalid.")

            for i, table in enumerate(self.hash_tables):
                binary_hash = self._hash(self.uniform_planes[i], query_point)
                candidates.update(table.get_list(binary_hash))

        # worst case scenario
        if len(candidates) == 0:
            print("No Similar Candidates")
#             for table in self.hash_tables:
#                 distances = [self.hamming_binary(binary_hash, key) for key in table.keys()]
FUTURE WORK

        # rank candidates by distance function
        candidates = [(ix.id, d_func(query_point, self._as_np_array(ix.bs)))
                      for ix in candidates]

        candidates.sort(key=lambda x: x[1])

        return candidates[:num_results] if num_results else candidates

    @staticmethod
    def hamming_binary(binary1, binary2):
        assert len(s1) == len(s2)
        return sum(c1 != c2 for c1, c2 in zip(s1, s2))

    ### distance functions

    @staticmethod
    def hamming_dist(bitarray1, bitarray2):
        xor_result = bitarray(bitarray1) ^ bitarray(bitarray2)
        return xor_result.count()

    @staticmethod
    def euclidean_dist(x, y):
        """ This is a hot function, hence some optimizations are made. """
        diff = np.array(x) - y
        return np.sqrt(np.dot(diff, diff))

    @staticmethod
    def euclidean_dist_square(x, y):
        """ This is a hot function, hence some optimizations are made. """
        diff = np.array(x) - y
        return np.dot(diff, diff)

    @staticmethod
    def euclidean_dist_centred(x, y):
        """ This is a hot function, hence some optimizations are made. """
        diff = np.mean(x) - np.mean(y)
```

```
        return np.dot(diff, diff)

    @staticmethod
    def l1norm_dist(x, y):
        return sum(abs(x - y))

    @staticmethod
    def cosine_dist(x, y):
        return 1 - np.dot(x, y) / ((np.dot(x, x) * np.dot(y, y)) ** 0.5)
```

storage.py

```
# lshash/storage.py
# Copyright 2012 Kay Zhu (a.k.a He Zhu) and contributors (see CONTRIBUTORS.txt)
#
# This module is part of lshash and is released under
# the MIT License: http://www.opensource.org/licenses/mit-license.php

import json

try:
    import redis
except ImportError:
    redis = None

__all__ = ['storage']


def storage(storage_config, index):
    """ Given the configuration for storage and the index, return the
    configured storage instance.
    """
    if 'dict' in storage_config:
        return InMemoryStorage(storage_config['dict'])
    elif 'redis' in storage_config:
        storage_config['redis']['db'] = index
        return RedisStorage(storage_config['redis'])
    else:
        raise ValueError("Only in-memory dictionary and Redis are supported.")


class BaseStorage(object):
    def __init__(self, config):
        """ An abstract class used as an adapter for storages. """
        raise NotImplementedError

    def keys(self):
        """ Returns a list of binary hashes that are used as dict keys. """
        raise NotImplementedError

    def set_val(self, key, val):
        """ Set `val` at `key`, note that the `val` must be a string. """
        raise NotImplementedError

    def get_val(self, key):
        """ Return `val` at `key`, note that the `val` must be a string. """
```

```
            raise NotImplementedError

        def append_val(self, key, val):
            """ Append `val` to the list stored at `key`.

            If the key is not yet present in storage, create a list with `val` at
            `key`.
            """
            raise NotImplementedError

        def get_list(self, key):
            """ Returns a list stored in storage at `key`.

            This method should return a list of values stored at `key`. `[]` should
            be returned if the list is empty or if `key` is not present in storage.
            """
            raise NotImplementedError


    class InMemoryStorage(BaseStorage):
        def __init__(self, config):
            self.name = 'dict'
            self.storage = dict()

        def keys(self):
            return self.storage.keys()

        def set_val(self, key, val):
            self.storage[key] = val

        def get_val(self, key):
            return self.storage[key]

        def append_val(self, key, val):
            self.storage.setdefault(key, []).append(val)

        def get_list(self, key):
            return self.storage.get(key, [])


    class RedisStorage(BaseStorage):
        def __init__(self, config):
            if not redis:
                raise ImportError("redis-py is required to use Redis as storage.")
            self.name = 'redis'
            self.storage = redis.StrictRedis(**config)

        def keys(self, pattern="*"):
            return self.storage.keys(pattern)

        def set_val(self, key, val):
            self.storage.set(key, val)

        def get_val(self, key):
            return self.storage.get(key)

        def append_val(self, key, val):
```

```
    self.storage.rpush(key, json.dumps(val))

def get_list(self, key):
    return self.storage.lrange(key, 0, -1)
```