Shazam Audio Search

In this homework you will implement a Shazam audio search engine.

I have provided a skeleton of the code below, and you will fill in any sections marked with `### START CODE BLOCK ###` and `### END CODE BLOCK ###`. Please do not change the code outside of these markers. In particular, do not define any new functions, change the code decomposition, or modify function inputs, outputs, or default values. Note that this code is written for ease of understanding and ease of grading, not for efficiency.

I have provided sample outputs for you to verify when your function is implemented correctly. Once you have completed this assignment, please run your notebook from beginning to end, make sure all outputs and plots are showing in the notebook, save your notebook, and then submit on Gradescope. Also, make sure to include both partners' names at the top of your notebook.

Partner 1 Name: Diego Weiss

Number of hours spent (Partner 1): 5

Partner 2 Name: Jasper Cox

Number of hours spent (Partner 2): 5

In [1]:
```
%matplotlib inline
```

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import librosa as lb
from scipy.signal import stft
import glob
import subprocess
import os.path
import pickle
from collections import defaultdict
#import hw3_solns
```

Step 1: Extract fingerprints (40 points)

The first step is to create a function that can extract fingerprints from an audio file.

In [3]:
```
def extractFingerprints(mp3file, sr = 22050, winsz = 2048, hop = 512, reg
    """
    Extract fingerprints from an audio file.

    Arguments:

    mp3file -- string specifying the full path of the mp3 file
    sr -- desired sampling rate
```

```
    winsz -- size of the STFT analysis window in samples
    hop -- hop size of STFT in samples
    regionH -- height of the rectangular region used to select maxima
    regionW -- width of the rectangular region used to select maxima
    maxDiffFrms -- maximum allowable time difference (in frames) between
    timeDiffQuant -- quantization factor applied to time differences betw

    Returns:

    result -- a list of (m,fp) tuples where fp is the fingerprint value (
    specifying when the fingerprint occurs in time.  The list of tuples i
    """
    # You don't need to change anything here, you'll modify the below fun
    peaks = getSpectralPeaks(mp3file, sr, winsz, hop, regionH, regionW)
    pairs = getPeakPairs(peaks, maxDiffFrms)
    result = calculateFingerprints(peaks, pairs, timeDiffQuant)
    return result
```

In the getSpectralPeaks function, you should

- calculate the one-sided STFT
- divide the STFT into rectangular neighborhood regions (you may ignore fragments at the edges)
- determine the location of the spectral maximum in each neighborhood region
- filter out any neighborhood maxima that are not local maxima (i.e. the maximum of the neighborhood region centered around the peak)
- sort the peaks by time in non-decreasing order

This function uses `getMaxIndex` and `isLocalMax`.

Hint: It can be confusing to think about which indices correspond to what in `S`. Remember that rows are the y coordinates and columns are the x coordinates.

In [4]:
```
def getSpectralPeaks(mp3file, sr = 22050, winsz = 2048, hop = 512, region
    """
    Determine the locations in time and frequency of spectral peaks in th

    Arguments:
    mp3file -- string specifying the full path of the mp3 file
    sr -- the desired sampling rate
    winsz -- size of the STFT analysis window in samples
    hop -- hop size of STFT in samples
    regionH -- height of the rectangular region used to select maxima
    regionW -- width of the rectangular region used to select maxima

    Returns:
    peaks_sorted -- a list of (m, k) tuples where m specifies the frame o
       k specifies the frequency bin of the peak.  The tuples are sorted
    """

    Y, sr  = lb.core.load(mp3file, sr = sr)
    f, t, S = stft(Y, fs = sr, nperseg = winsz, noverlap = winsz - hop)
    Smag = np.abs(S)
    height, width = S.shape
    peaks = []

    ### START CODE BLOCK ###
```

```
    for x in range(0, width-regionW+1, regionW):
        for y in range(0, height-regionH+1, regionH):
            region = Smag[y:y+regionH,x:x+regionW]
            (m, k) = getMaxIndex(region, y, x)
            if isLocalMax(k, m, Smag, regionH, regionW):
                peaks.append((m, k))
                        # hint: you can do it in one line!
    peaks_sorted = sorted(peaks, key=lambda p: p[0]) # sort by m

    return peaks_sorted
```

In the getMaxIndex function, you should

- Find the index of the maximum in `region`
- Translate this index to the absolute indexing using `x` and `y`

Be careful here about how you use the indices of `region` : in our indexing scheme, the point (i,j) is found at `region[j][i]` .

This function is called by `getSpectralPeaks` .

Hint: If you find the appropriate numpy functions, you can write this in only a few lines. The function `numpy.unravel_index()` is useful here.

In [5]:
```
def getMaxIndex(region, row, col):
    """
    Determine the index of the maximum of a region and translate it back

    Arguments:
    region -- a matrix to find the maximum of
    row -- the absolute row number of the bottom left corner of 'region'
    col -- the absolute col number of the bottom left corner of 'region'

    Returns:
    m -- the absolute column of the max
    k -- the absolute row of the max
    """

    ### START CODE BLOCK ###

    if region.size == 0:
        raise ValueError("getMaxIndex: empty region has no maximum")

    # Flatten argmax and convert back to (row_in_region, col_in_region)
    flat_index = np.argmax(region)
    r, c = np.unravel_index(flat_index, region.shape)

    m = col + c  # absolute column (time frame)
    k = row + r  # absolute row (frequency bin)
    return m, k
```

In the isLocalMax function, you should:

- determine the rectangular neighborhood region centered around the specified (row,col) element of matrix S (note that at the edges of S, this region may not be fully centered around the element)

- return True if the specified element is the maximum of this centered neighborhood region

This function is called by getSpectralPeaks()

Hint: You need to "clamp" the region around (row, col) if it exceeds the boundaries of `S`. Python built-in functions will be helpful here. Consider using `max`, for example.

In [6]:
```
def isLocalMax(row, col, S, regionH, regionW):
    '''
    Return true if the specified element in S is the maximum of a rectang
    This is a helper function used to filter out neighborhood peaks that

    Arguments:

    row -- the row of the specified element
    col -- the column of the specified element
    S -- the matrix
    regionH -- the height of the rectangular region in rows
    regionW -- the width of the rectangular region in columns

    Returns:

    A boolean indicating if the specified element is the local maximum of
    around it.  The rectangular region extends from from rows [row - regi
    columns [col - regionW/2, col + regionW/2).
    '''

    ### START CODE BLOCK ###
    # check if we need to clamp the region
    rows = S.shape[0]
    cols = S.shape[1]
    halfH_left  = regionH // 2
    halfH_right = regionH - halfH_left
    halfW_left  = regionW // 2
    halfW_right = regionW - halfW_left

    top    = max(0, row - halfH_left)
    bottom = min(rows, row + halfH_right)
    left   = max(0, col - halfW_left)
    right  = min(cols, col + halfW_right)

    region = S[top:bottom, left:right]
    if region.size == 0:
        return False


    # First check value-based maximality.
    center_val = S[row, col]
    region_max = np.max(region)
    if np.isnan(center_val) or center_val != region_max:
        return False


    m, k = getMaxIndex(region, top, left)
    isMax = (m, k) == (col, row)
    ### END CODE BLOCK ###
```

```
    return isMax
```

In the getPeakPairs function, you should:

- iterate through the sorted list of peaks
- find all pairs of peaks that have a time difference <= maxDiff
- construct a list of tuples (i,j) containing the peak pair indices where i < j

In [7]:
```
def getPeakPairs(peaks, maxDiff):
    """
    Return a list of all peak pairs that satisfy the maximum time differe

    Arguments:
    peaks -- a list of (m,k) tuples specifying the frame offset and frequ
       are sorted in m in non-decreasing order.
    maxDiff -- the maximum allowable time difference between peaks in a p

    Returns:
    pairs -- a list of (i,j) tuples specifying the indices of peak pairs,
       in the given list of peaks.  This list of tuples should be non-dec
    """
    pairs = []

    ### START CODE BLOCK ###
    i = 0
    j = 1
    n = len(peaks)
    for i in range(0, n-1):
        for j in range(i+1, n):
            if peaks[j][0] - peaks[i][0] <= maxDiff:
                pairs.append((i,j))

    ### END CODE BLOCK ###

    return pairs
```

In the calculateFingerprints function, you should:

- iterate through the peak pairs
- compute the fingerprint integer value for each peak pair
- store the fingerprint integer value and time offset in a list

Note that this function should call the packIntoInt() function.

In [8]:
```
def calculateFingerprints(peaks, pairs, timeDiffQuant):
    """
    Calculate a list of fingerprints on a set of peak pairs.

    Arguments:
    peaks -- a list of (m,k) tuples specifying the frame offset and frequ
       are sorted in m in non-decreasing order.
    pairs -- a list of (i,j) tuples specifying the indices of peak pairs.
       non-decreasing in i.
    timeDiffQuant -- quantization factor applied to time differences betw

    Returns:
    result -- a list of (m, fp) tuples where m specifies the frame offset
```

```
    anchor peak) and fp specifies the fingerprint value (integer)
    """
    result = []

    ### START CODE BLOCK ###
    # get m out of peaks
    for pair in pairs:
        peak1 = peaks[pair[0]]
        peak2 = peaks[pair[1]]
        peak1_freq = peak1[1]
        peak2_freq = peak2[1]
        time_diff = peak2[0]-peak1[0]
        fp = packIntoInt(peak1_freq, peak2_freq, time_diff, timeDiffQuant
        result.append((peak1[0],fp))

    ### END CODE BLOCK ###

    return result
```

In the packIntoInt function, you should:

- store f1 in the first 10 bits
- store f2 in the next 10 bits
- store the quantized time difference in the remaining bits

Note that you can shift an integer to the left by n bits by multiplying it by 2^n.

In [9]:
```
def packIntoInt(f1, f2, timediff, quant_factor):
    """
    Calculate the fingerprint value of a peak pair and package into a sin

    Arguments:
    f1 -- frequency bin of anchor peak
    f2 -- frequency bin of non-anchor peak
    timediff -- time difference between peaks (in frames)
    quant_factor -- quantization factor applied to the time difference

    Returns:
    fp -- a single integer that specifies the fingerprint value, where th
       f1, the next 10 bits specify f2, and the remaining bits specify th
    """

    ### START CODE BLOCK ###
    fp = int(f1) | (int(f2) << 10) | (int(timediff/quant_factor) << 20)
    ### END CODE BLOCK ###

    return fp
```

Below I have included some sample outputs for you to verify that your implementation is correct.

In [10]:
```
a = np.zeros((4,5))
a[2,4] = 1
getMaxIndex(a, 10, 20)
```

Out[10]:
```
(np.int64(24), np.int64(12))
```

(24, 12)

In [11]:
```
peaks = getSpectralPeaks('refs/005936.mp3')
peaks[0:5], len(peaks)
```

Out[11]:
```
([(np.int64(3), np.int64(574)),
  (np.int64(3), np.int64(944)),
  (np.int64(11), np.int64(850)),
  (np.int64(64), np.int64(24)),
  (np.int64(86), np.int64(507))],
 97)
```

([(3, 574), (3, 944), (11, 850), (64, 24), (86, 507)], 97)

In [12]:
```
pairs = getPeakPairs(peaks, 120)
pairs[0:10], len(pairs)
```

Out[12]:
```
([(0, 1),
  (0, 2),
  (0, 3),
  (0, 4),
  (0, 5),
  (0, 6),
  (1, 2),
  (1, 3),
  (1, 4),
  (1, 5)],
 825)
```

([(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (1, 2), (1, 3), (1, 4), (1, 5)], 825)

In [13]:
```
packIntoInt(34, 240, 73, 2)
```

Out[13]:
```
37994530
```

37994530

In [14]:
```
fps = calculateFingerprints(peaks, pairs, 2)
fps[0:8], len(fps)
```

Out[14]:
```
([(np.int64(3), 967230),
  (np.int64(3), 5065278),
  (np.int64(3), 31482430),
  (np.int64(3), 43511358),
  (np.int64(3), 43790910),
  (np.int64(3), 50347582),
  (np.int64(3), 5065648),
  (np.int64(3), 31482800)],
 825)
```

([(3, 967230), (3, 5065278), (3, 31482430), (3, 43511358), (3, 43790910), (3, 50347582), (3, 5065648), (3, 31482800)], 825)

In [15]:
```
fps = extractFingerprints('refs/005936.mp3')
fps[0:8], len(fps)
```

Out[15]:
```
([(np.int64(3), 967230),
  (np.int64(3), 5065278),
  (np.int64(3), 31482430),
  (np.int64(3), 43511358),
  (np.int64(3), 43790910),
  (np.int64(3), 50347582),
  (np.int64(3), 5065648),
  (np.int64(3), 31482800)],
 825)
```

([(3, 967230), (3, 5065278), (3, 31482430), (3, 43511358), (3, 43790910), (3, 50347582), (3, 5065648), (3, 31482800)], 825)

Construct database (10 points)

Now we will construct our database of fingerprints.

In the constructDatabase function, you will:

- iterate through all mp3 files in the directory (hint: use `glob`)
- extract a list of fingerprints from each file
- store the fingerprint information in the dictionary d as an inverted index

Note that this function should call the extractFingerprints() function.

In [16]:
```
def constructDatabase(indir):
    """
    Construct a database of fingerprints for all mp3 files in the specifi

    Arguments:
    indir -- directory containing mp3 files

    Returns:
    d -- database of fingerprints in the form a python dictionary, where
        and the value is a list of (song, m) tuples specifying the name o
        the time offset in frames where the matching fingerprint occurs.
    """
    # a defaultdict is like a python dictionary, but if you have some 'ke
    # 'd[key]' is initialized to a default value (in this case the empty
    # always safe to do 'd[key].append(value)'.
    d = defaultdict(list)

    ### START CODE BLOCK ###
    mp3_files = glob.glob('*.mp3',root_dir=indir)
    for rel_name in mp3_files:
        full_path = os.path.join(indir, rel_name)
        song = os.path.basename(rel_name)

        # Assumes extractFingerprints returns iterable of (m, fp)
        for (m, fp) in extractFingerprints(full_path):
            d[int(fp)].append((song, int(m)))
    ### END CODE BLOCK ###

    return d
```

In [17]:
```
ref_dir = 'refs/'
```

```
db = constructDatabase(ref_dir) # note that this may take a minute or so
```

Let's save the database to file for convenient access later.

In [18]:
```
with open('db.pkl','wb') as f:
    pickle.dump(db, f)
```

In [19]:
```
with open('db.pkl','rb') as f:
    db = pickle.load(f)
```

Search (30 points)

Now we will process the audio queries in the queries/ folder. These are random 10 second segments extracted from the reference tracks.

In the getMatchScore function, you will:

- get a list of scatterpoints and offsets for matching fingerprints by calling the `getOffsets` function
- convert the list of scatterpoints and offsets to numpy arrays
- compute a histogram of offsets with bin width 1. The function `numpy.histogram` may be useful.
- calculate the match score as the maximum histogram count

Hint: Make sure your function handles the case when there are no matching fingerprints at all!

In [21]:
```
def getMatchScore(query_fps, db, songid, visualize = False):
    """
    Determine the match score between a set of query fingerprints and a s

    Arguments:
    query_fps -- a list of tuples (m, fp) specifying the time offset (in
       fingerprints extracted from an audio query
    db -- database of fingerprints
    songid -- the song of interest, specified as a filename (e.g. '005936
    visualize -- if True, show a plot containing (a) a scatterplot of mat
       given song, and (b) a histogram of offsets for matching fingerprin

    Returns:
    matchScore -- a scalar indicating how well the query matches this son
    """

    ### START CODE BLOCK ###
    points, offsets = getOffsets(query_fps, db, songid)
    offset_count = len(offsets)
    if offset_count == 0:
        return 0
    histogram, bin_edges = np.histogram(offsets, bins=np.arange(min(offse
    matchScore = max(histogram)


    ### END CODE BLOCK ###

    if visualize:
```

```
        if len(points) == 0:
            print('No matching fingerprints')
        else:
            plt.subplot(121)
            xs, ys = zip(*points)
            plt.plot(xs, ys, 'o')
            plt.title('Matching Fingerprints')
            plt.xlabel('Ref Time (frames)')
            plt.ylabel('Query Time (frames)')
            plt.subplot(122)
            plt.hist(offsets, bins=np.arange(min(offsets), max(offsets)+2
            plt.show()

    return matchScore
```

In the `getOffsets` function, you will:

- iterate through the query fingerprints
- retrieve all matching fingerprints from the inverted index
- filter out any matching fingerprints from songs other than the one specified in songid
- create a list of (ref_offset, query_offset) tuples for matching fingerprints in songid
- return that list, as well as a list containing the (ref_offset - query_offset) differences

In [22]:
```
def getOffsets(query_fps, db, songid):
    """
    Find the offsets of matching fingerprints between a query and a datab

    Arguments:
    query_fps -- a list of tuples (m, fp) specifying the time offset (in
       fingerprints extracted from an audio query
    db -- database of fingerprints
    songid -- the song of interest, specified as a filename (e.g. '005936

    Returns:
    points -- a list of (ref_offset, query_offset) tuples
    offsets -- a list of (ref_offset - query_offset) values
    """

    points = []
    #db structure
    # key: fingerprint
    # value: (song_id, frame offset)
    ### START CODE BLOCK ###
    offsets = []
    for query_offset, fp in query_fps:
        # (frame offset, fingerprint)
        matches = db[fp]
        for match_song_id, ref_offset in matches:
            # check if song id matches
            if match_song_id == songid:
                points.append((ref_offset, query_offset))
                offsets.append(ref_offset - query_offset)
    ### END CODE BLOCK ###

    return points, offsets
```

In the findBestMatch function, you will:

- extract fingerprints from the specified query file
- iterate through all of the songs in the database and calculate a match score on each
- sort the songs in decreasing order of match score
- if verbose = True, print out the top 5 match scores and song names
- determine the song with highest match score

Note that this function should call the extractFingerprints() and getMatchScore() functions.

In [23]:
```
def findBestMatch(query_file, db, ref_dir, verbose = False):
    """
    Return the name of the mp3 file that has the highest match score.

    Arguments:
    query_file -- string specifying the query audio file
    db -- database of fingerprints
    ref_dir -- directory containing all of the reference files in the dat
    verbose -- whether or not to print out the top 5 matches (for debuggi

    Returns:
    bestmatch -- a string containing the name of the mp3 file with highes
    """

    ### START CODE BLOCK ###
    bestMatch = None

    # call extractFingerprints
    query_fps = extractFingerprints(query_file)
    # extract all song names from reference directory
    mp3_files = glob.glob('*.mp3',root_dir=ref_dir)

    # pass those into getMatchScore along with query fingerprints
    match_scores = []
    for songid in mp3_files:
        # call getMatchScore
        match_score = getMatchScore(query_fps, db, songid)
        match_scores.append((songid, int(match_score)))

    match_scores_sorted = sorted(match_scores, key=lambda p: p[1], revers

    if verbose:
        print(match_scores_sorted[0:5])

    return match_scores_sorted[0][0]
    ### END CODE BLOCK ###

    return bestMatch
```

In the runBenchmark function, you will:

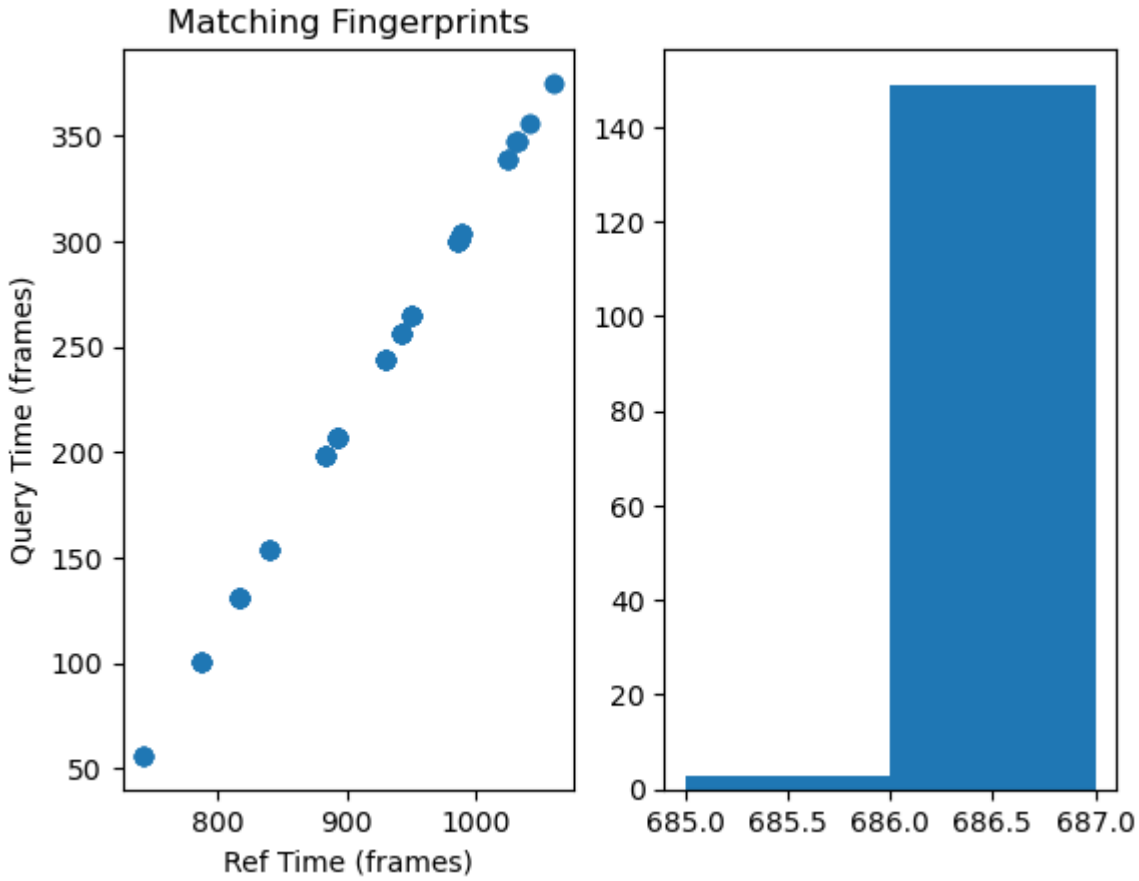
- iterate through all of the query files in the specified directory
- find the song with highest match score
- check if the predicted match is correct
- print out the accuracy across all queries

Note that this function should call the findBestMatch() function.

In [24]:
```
def runBenchmark(query_dir, ref_dir, db):
    """
    Run an entire benchmark and print out the accuracy of the system.

    Arguments:
    query_dir -- directory containing the query mp3 files
    ref_dir -- directory containing the reference mp3 files
    db -- database of fingerprints

    Returns:
    No return value.  Prints out the accuracy of the system.
    """

    ### START CODE BLOCK ###
    accuracy = 0

    query_files = glob.glob('*.mp3',root_dir=query_dir)
    #reference_files = glob.glob('*.mp3',root_dir=ref_dir)
    #print(query_files)
    for q in query_files:
        full_path = os.path.join(query_dir, q)
        song = os.path.basename(q)

        best_match = findBestMatch(full_path, db, ref_dir)
        if best_match == song:
            accuracy += 1
        else:
            print(f"Query name = {song}, match_name = {best_match}\n")
            getMatchScore(extractFingerprints(full_path),db,best_match,vi
    accuracy = accuracy / len(query_files)
    ### END CODE BLOCK ###




    print('Accuracy: %.2f %%' % (accuracy * 100.0))
```

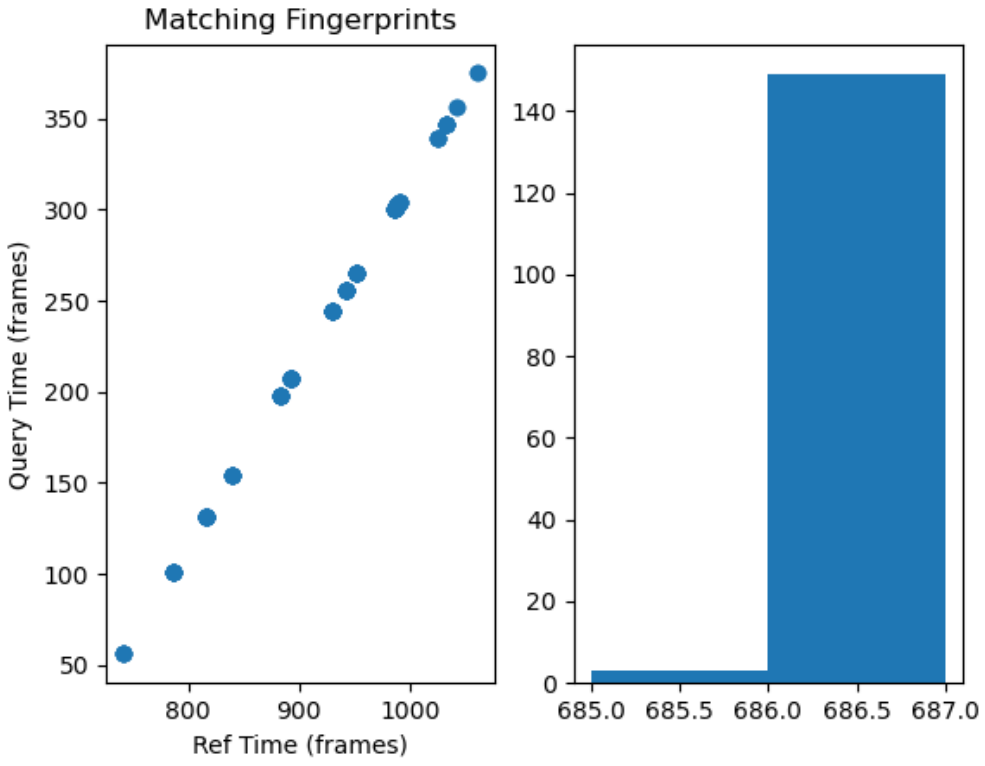
Here are some sample outputs to verify that your implementation is correct.

In [25]:
```
query_fps = extractFingerprints('queries/005936.mp3')
getMatchScore(query_fps, db, '005936.mp3', True)
```



Out[25]: np.int64(149)

149



In [26]:
```
getMatchScore(query_fps, db, '054156.mp3', True)
```

Out[26]: 0

No matching fingerprints

0

In [27]:
```
findBestMatch('queries/120208.mp3', db, ref_dir, True)
```

```
[('120208.mp3', 226), ('113934.mp3', 1), ('066076.mp3', 1), ('148610.mp3',
0), ('054156.mp3', 0)]
```

Out[27]: '120208.mp3'

Top match scores 120208.mp3 226, 066076.mp3 1, 113934.mp3 1, 114065.mp3 0, 071372.mp3 0

'120208.mp3'

In [28]:
```
runBenchmark('queries', ref_dir, db)
```

```
Accuracy: 100.00 %
```

Accuracy: 100.00 %

Test on class dataset (20 points)

Now we will run some experiments on real noisy data that our class will collect.

In this part, you will do the following:

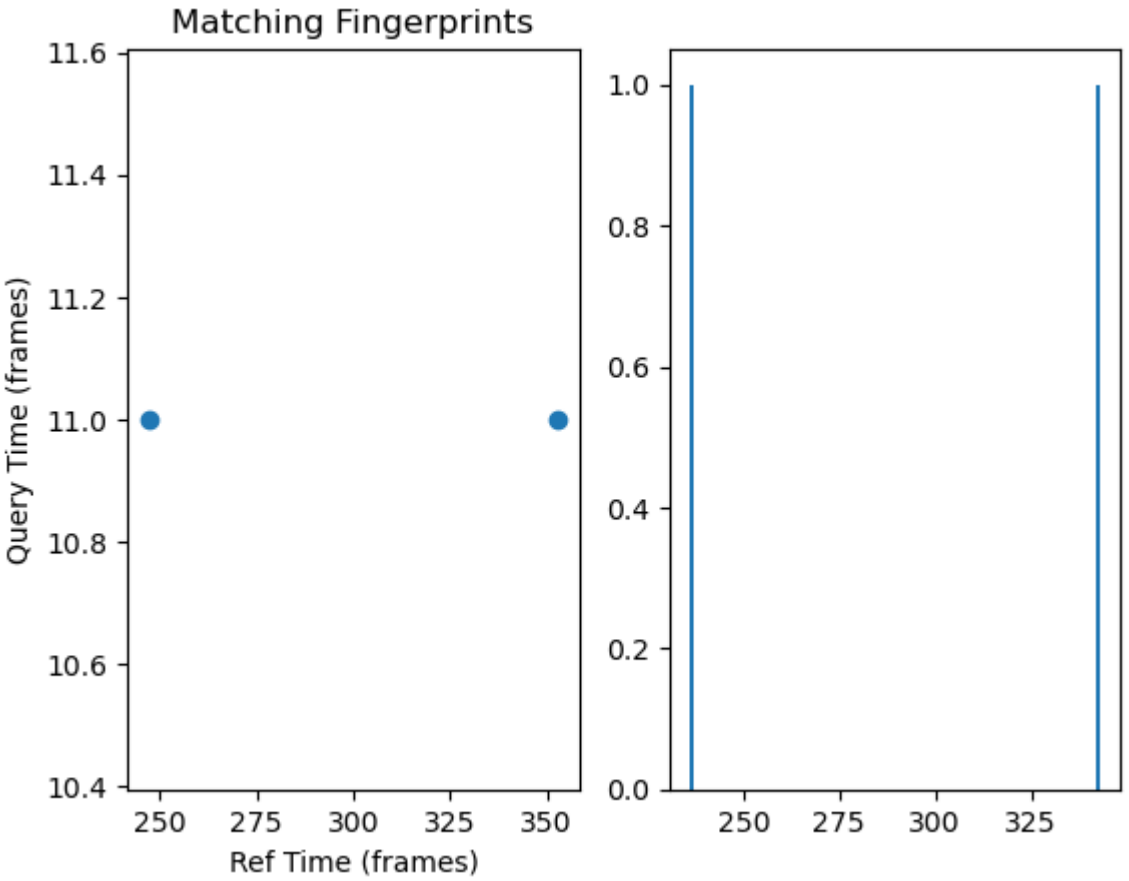
- Each team will collect 3-4 cell phone recordings. Please sign up for your 3-4 recordings in this spreadsheet. For each recording, play the original audio recording from your laptop (or nicer speaker if you have one) and record approximately 10 seconds of the song on your cell phone. Upload your recordings to this shared google drive folder, making sure to name them appropriately (they should have the same basename as the original audio recording). Your cell phone recordings should be uploaded to the folder by Saturday noon (-5 points if not done by then) to ensure everyone has enough time to run experiments before the homework deadline.
- Once our class has finished collecting the audio data, download the audio data from the shared google drive folder and save it into your homework directory.
- Run a benchmark on the real noisy queries. Describe how the system behaves with real noisy queries vs clean queries. What are the weaknesses of the system? Use plots, figures, and/or data to support your answer.
- Describe one way that you could improve the performance of the system on the real noisy queries. Explain the tradeoffs (if any) of your proposal, describe the intuition for why it would improve performance, and provide plots, figures, and/or data to support your answer.

Please make sure to include both your comments (in markdown cells) and your code and

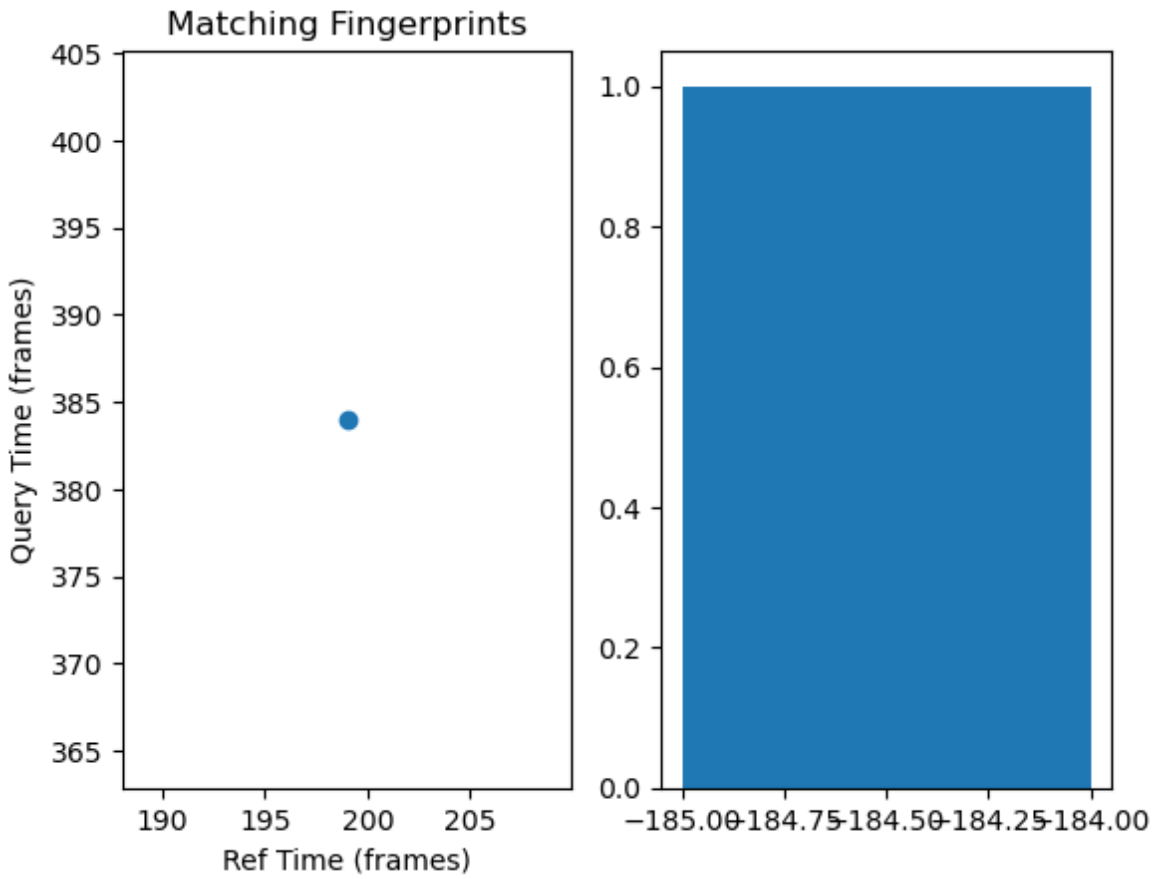
visualizations.

In [29]:
```
### INSERT CODE AND MARKDOWN CELLS BELOW ###
runBenchmark('class_queries', ref_dir, db)
```

```
Query name = 139536.mp3, match_name = 054719.mp3
```




```
Query name = 066076.mp3, match_name = 115288.mp3
```




```
Accuracy: 93.33 %
```

Just looking at the benchmark, we can immediately determine that the performance of our system worsens slightly with noisy queries.

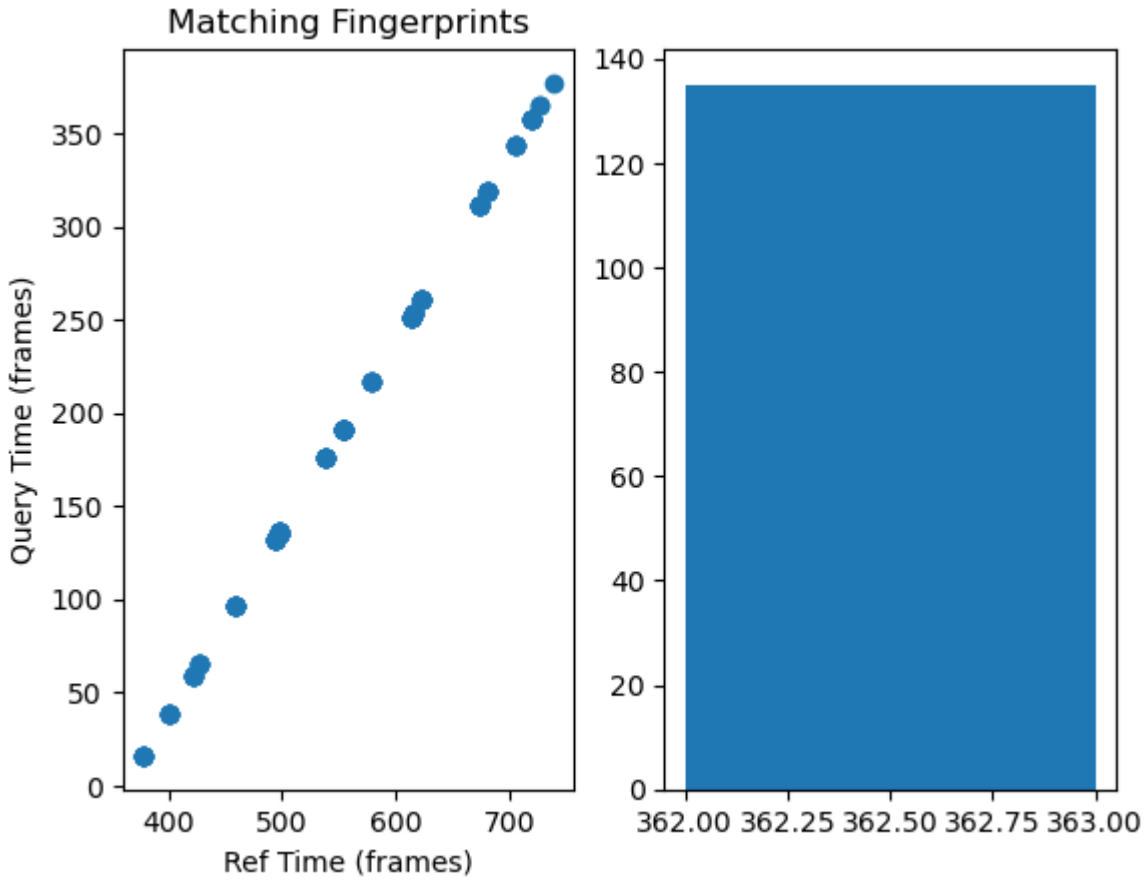
In [30]:
```
findBestMatch('class_queries/038820.mp3', db, ref_dir, True)
findBestMatch('queries/038820.mp3', db, ref_dir, True)
```

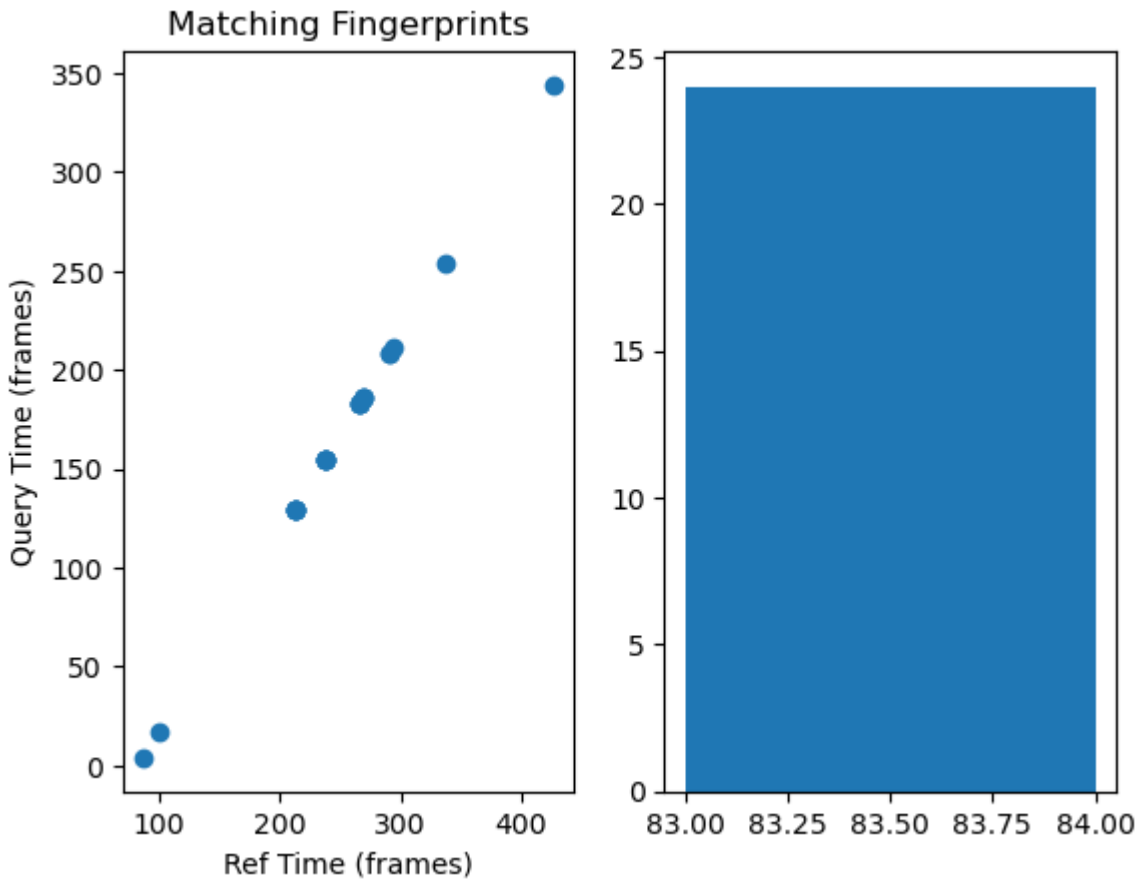
```
[('038820.mp3', 33), ('148610.mp3', 0), ('054156.mp3', 0), ('057164.mp3',
0), ('139536.mp3', 0)]
[('038820.mp3', 151), ('148610.mp3', 0), ('054156.mp3', 0), ('057164.mp3',
0), ('139536.mp3', 0)]
```

Out[30]:
```
'038820.mp3'
```

Comparing the output of findbestMatch on a noisy query and the original query, we see that the match score is significantly (5x) higher for the original query than the noisy query.

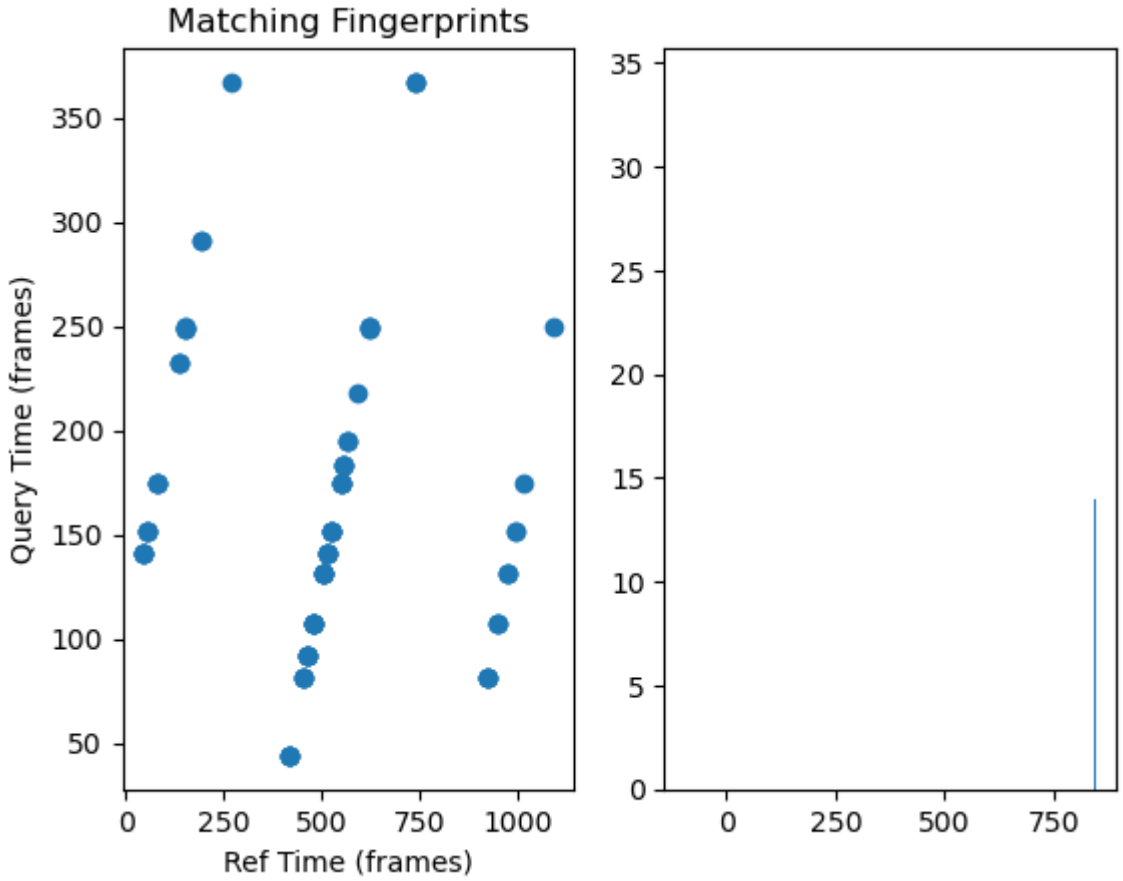
In [31]:
```
query_fps = extractFingerprints('queries/019187.mp3')
class_fps = extractFingerprints('class_queries/019187.mp3')
getMatchScore(query_fps, db, '019187.mp3', True)
getMatchScore(class_fps, db, '019187.mp3', True)
```





Out[31]: np.int64(24)

In [32]:
```
query_fps = extractFingerprints('queries/139536.mp3')
class_fps = extractFingerprints('class_queries/139536.mp3')
getMatchScore(query_fps, db, '139536.mp3', True)
getMatchScore(class_fps, db, '139536.mp3', True)
```



Out[32]: 0