Query by Humming via Locality Sensitive Hashing

Naveen Sharma
CSCI 1952Q
Brown University
Providence, RI 02912
naveen_sharma@brown.edu

Abstract

In this paper, I present a query by humming (QBH) algorithm using locality-sensitive hashing for queries on popular music. Oftentimes, when someone wishes to identify a song, but only remembers the tune and not the lyrics, it is difficult to determine the song they are thinking of. With QBH, we can query large song databases given a hummed query input. This algorithm uses locality-sensitive hashing (LSH) to dramatically speed up performance when querying large databases without sacrificing accuracy. To process queries with songs, a pitch vector construction and dynamic time warping algorithm is applied to construct normalized melodies. Melodies are compared using a Euclidean LSH algorithm specifically designed for melody vectors, which I call Melodic LSH (MLSH). Much of the QBH models in the literature are trained to query a database of MIDI melodies. While they are able to achieve high accuracy on these datasets, the models often fail to perform well on full songs, specifically popular music of today. This QBH algorithm is fine-tuned for performance for popular music. On a dataset of 109 popular songs gathered from YouTube, the algorithm achieves a Top-3 hit rate of 75%. On the MIR-QBSH dataset, which consists of 48 MIDI folk-songs, the algorithm achieves a Top-3 hit rate of 51%. The algorithm demonstrates a fine-tuned approach to the query-by-humming problem that performs very well on popular music. See this link for example results with audio.

1 Introduction

Query by humming is an algorithm designed to search for a song given a hummed or sung melody by a user. The goal is that regardless of the key and tempo the user's query is in, the algorithm will be able to accurately determine a ranking of possible songs. This is especially useful in instances where someone forgets the lyrics to a song, but wants to know what it is called. For practical use, we would want a query by humming to be able to search through as many songs as possible. However, as the number of songs increases, it becomes far too computationally expensive to brute force search through the database. We require a method to efficiently search through a database of melodies, while maintaining the accuracy of the search results. To solve this problem, we make use of two key ideas. The first is called pitch vector extraction. Pitch vector extraction transforms an audio input into a set of fixed-length d vectors of MIDI note values. The note values map frequencies to the keys on a keyboard, and are denoted with the integers 0 through 88. The second key idea is locality-sensitive hashing (LSH). Now that we have a representation of audio input as vectors in d-dimensional space, we can employ LSH to drastically improve our search efficiency. In this paper, I utilize a fine-tuned LSH algorithm which I call MLSH, which is able to hash pitch vectors and perform very efficient queries. The complete framework of the algorithm is described in Figure 1. First, audio recordings from the database are processed via pitch vector extraction and stored in a database. This represents the set of pitch vectors we will be querying from. When a new query comes in, we extract pitch vectors and transform the query using dynamic time warping and recursive

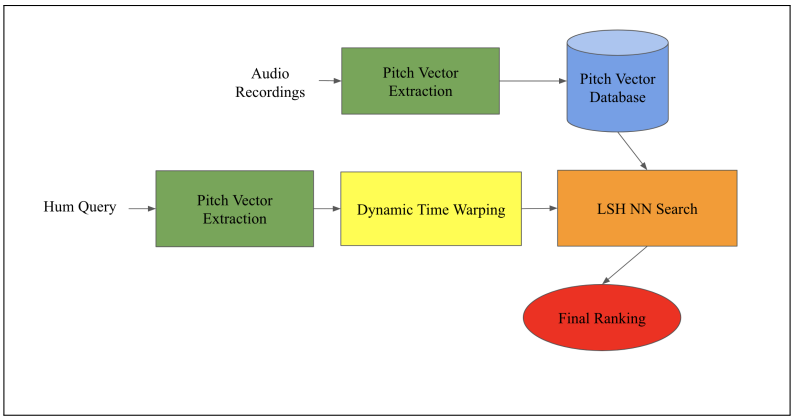

Figure 1: Framework of the Query by Humming algorithm, broken up into components

alignment before performing LSH nearest neighbor search. The LSH search compares transformed query pitch vectors with pitch vectors from our database and returns a final ranking of songs.

Previous research on QBH models use similar techniques and a variety of other approaches. [1] proposes a QBH algorithm using LSH and provides much of the framework and inspiration for this project. Unlike [1], however, this algorithm retrieves input directly from music as opposed to reading in MIDI files. This improvement was suggested in that paper as a future improvement. In addition, this project uses dynamic time warping and recursive alignment algorithms [2] which have been employed in a QBH context in [3], [4]. In industry, Google has created a application built into Search which allows users to search for songs by humming [5]. Much of the existing research on this task has been fine-tuned for performance on short MIDI melodies. The dataset used as a baseline in much of the literature (the MIR-QBSH dataset) consists mainly of one minute folksongs. In this paper, we are optimizing the algorithm for performance on popular songs of today. As potential users of this application are most likely searching for songs they will listen to, I feel as though it is more valuable to tune to algorithm this way. Despite this, testing will still be conducted on MIR-QBSH to provide a benchmark on the algorithm's performance compared to the state-of-the-art.

2 Pitch Vector Extraction

In order to hash and search through melodies, we must first come up with a way to represent them as vectors. Pitch vector extraction describes the process of transforming an audio input into a set of fixed-length vectors of MIDI notes.

2.1 Audio to Frequency Vector

When a person sings or hums a note, they are actually creating sound in a variety of frequencies (called overtones and undertones). It is this unique combination of frequencies, each having a different amplitude, that gives someone's voice its unique sound. So, to extract the main pitch from a human vocal audio file, we cannot simply extract the frequencies. We must take into consideration which frequencies are the loudest, and ensure that the melody we extract does not switch between principal and overtone melodies.

To do so, we apply a deep learning model called Convolutional Representation for Pitch Estimation (CREPE) [6]. CREPE is a convolutional neural network that operates directly on the time-domain waveform. It achieves an accuracy of approximately 0.999 on pitch extraction tasks (see [6]). Figure 2 demonstrates an example of CREPE's pitch extraction overlayed onto a waveform from the MIR-QBSH dataset. In this example, the melody is "Happy Birthday". The output of the CREPE model is a vector of time, frequencies, and confidence. Through some pruning of low confidence notes and sampling at predefined time intervals (50ms was used in the final version), we arrive at a frequency vector f.

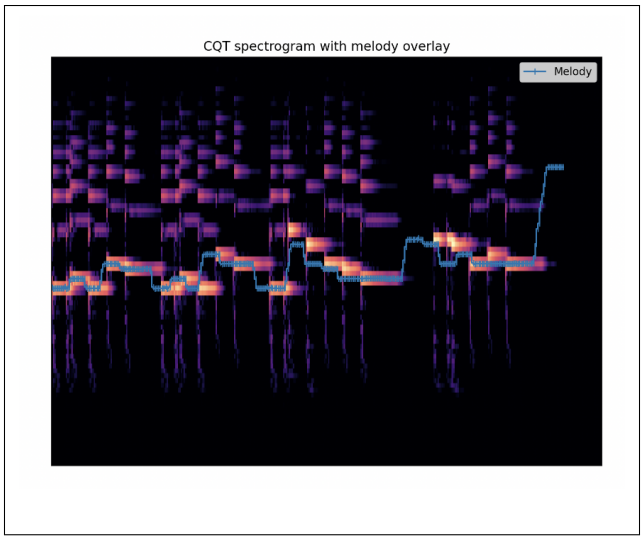

Figure 2: CREPE melody extraction onverlayed onto time-domain waveform

When dealing with real music, rather than MIDI input, we need to isolate one melody from the entire mix. Most popular songs today have several vocal layers, harmonies, and many instrumental tracks all playing at once. To handle this, I apply a vocal remover algorithm before frequency vector extraction. The vocal remover used is a recurrent neural network (RNN) trained to separate audio sources [7]. Since users are more likely to hum the vocal melody in a song, rather that the instrumental melodies, I apply this algorithm to extract vocals before using CREPE.

2.2 Frequency Vector to Pitch Vectors

Using the `librosa` module, frequencies are converted to MIDI note values. This process maps the frequencies onto the keys of a keyboard, denoted by the integers 0 through 88. Next, using an approach similar to [1], we now convert MIDI note vectors into a set of fixed-length vectors called pitch vectors. This is done by sliding a window of size d across the frequency vector, normalizing the MIDI pitches to have a mean of 0. Each iteration, the window is moved forward by a step of s. This process is repeated until the entire vector is processed. The result is a set of pitch vectors of length d. The parameters d and s are chosen based on how long we consider a significant melody to be, and how different we want each melody to be from another. In the final implementation, d was chosen to be 120 (corresponding to around 6 seconds of audio) and s was chosen to be 20 (corresponding to around 1 second of audio).

3 Dynamic Time Warping

When comparing a hummed query to an actual song, three main problems arise:

1. What if the user hums in the wrong key?
2. What if the user hums a melody that does not line up directly with one in the song database, but at some offset?
3. What if the user hums the song too fast or too slow?

All of these issues are solved using a technique called dynamic time warping (DTW). DTW is described as a method used to compare temporal sequences which may vary in speed and is used to determine the optimal set of transformations we can apply to a query to best match a song vector. For the first problem outlined above, we can use a vertical shift to solve it. For the second, we can use a horizontal shift. Finally, for the third, we can use horizontal scaling. Note that we cannot vertically scale the query since that would inherently change the melody being hummed, as melodies directly rely on the intervals between notes in order to be identified. To determine the optimal shift and scales

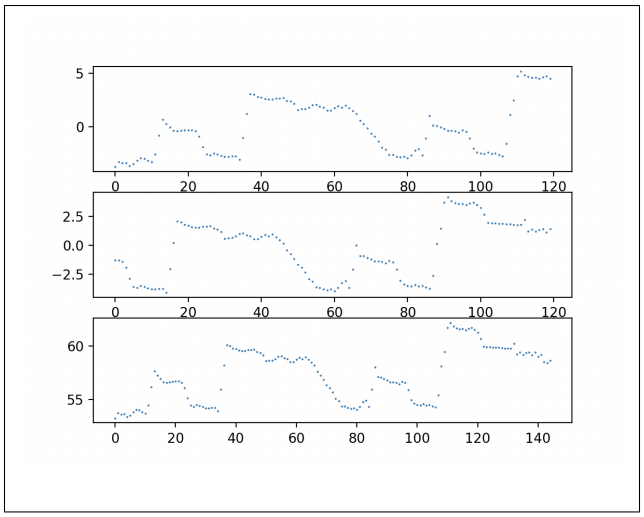

Figure 3: Top and Middle: examples of pitch vectors with $d = 120, s = 20$,
Bottom: plot of entire query

to apply to the query, DTW simply runs through combinations of horizontal shift, vertical shift, and scale parameters and determines which combination minimizes the Euclidean distance between the query and the song. So, rather than compare a query and a song using Euclidean distance alone, we compare using a dynamic time warping distance function to determine a more accurate comparison. In this model, horizontal shifts of up to 25 positions, vertical shifts of up to 5 notes, and scaling from 0.7 to 1.4 were permitted. Figure 4 demonstrates an example of dynamic time warping on one of the queries from the MIR-QBSH dataset. Here we can observe that prior to DTW, the blue graph does not line up at all with the song vector in orange - even though they are both versions of the same melody. After DTW, the green curve is generated. In this example, the pitch vector was scaled leftwards and horizontally scaled down to construct the green curve. And as we can see, the green curve now almost perfectly matches the song vector. Without DTW the algorithm deems this specific query-song pair as a rank 4 match, but after DTW, the algorithm correctly matches the query and song with rank 1.

The function to compute distance $D(p_q, p_s)$ between a scaled query pitch vector p_q and a song vector p_s is as follows:

$$D(p_q, p_s) = ||p_q - p_s|| = (\Sigma_{i=1}^{d}(p_{qi} - p_{si})^2)^{1/2} \tag{1}$$

3.1 Recursive Alignment

As described in [2], we can extend the idea of DTW to further improve the performance of our QBH model. Oftentimes, when people hum or sing songs, they modulate the tempo as they go. For instance, someone may start singing a song quickly but gradually slow down. Also, people tend to shorten, or omit, rests and pauses when humming songs. Using DTW with linear scaling and shifting alone, we would be unable to accurately match up a query to a song. To solve this issue, we utilize an algorithm known as recursive alignment, or RA. Recursive alignment recursively breaks up pitch vectors into segments, splitting into two halves each time. After each split, linear scaling is applied optimally to line up the pitch vector fragment to the song vector. This is done in the same manner as DTW. Then, the newly aligned segment is passed back into RA recursively until some maximum depth r is reached. The result is a pitch vector which can modulate its tempo several times. For instance, a pitch vector can now start with a scaling of 0.8, then finish with a scaling of 1.2, to account for the user changing tempo while humming the tune. Figure 5 demonstrates an example of RA with $r = 2$. The blue line represents scaled query using DTW alone while the red line represents the scaled query

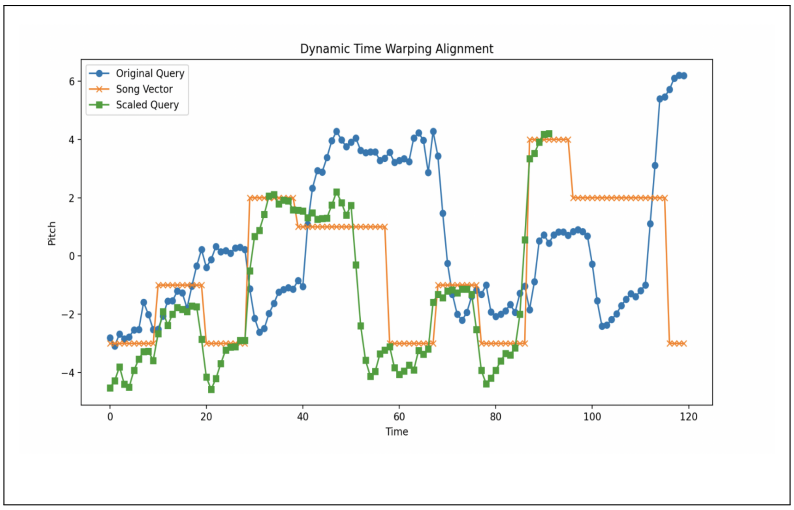

Figure 4: Example of Dynamic Time Warping

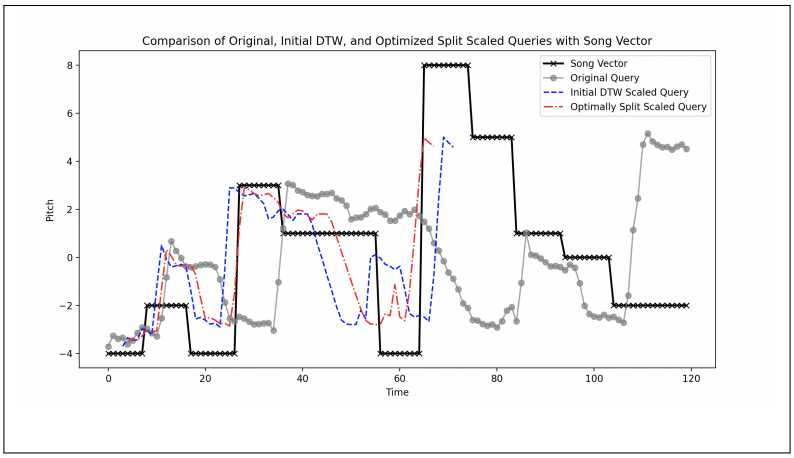

Figure 5: Example of Recursive Alignment with $r = 2$

using RA. While both curves do not match up perfectly to the song vector, the red curve is able to reach a better match by speeding up the query in the second half.

4 Locality Sensitive Hashing

Locality sensitive hashing (LSH) is a popular method for decreasing time complexity for nearest neighbor search with especially large datasets. What is so remarkable about LSH is that while significantly improving search efficiency, the accuracy of search results is barely compromised [8]. Another benefit of LSH is that it is extensible to a wide domain of inputs. For example, LSH can be used with documents, binary vectors, Euclidean vectors, and many more input types. Because of this, LSH has a broad range of applications such as image search [9] and collaborative filtering [10].

4.1 E2LSH

A commonly used variant of LSH for Euclidean vectors is E2LSH [11]. This algorithm is able to hash Euclidean vectors using an idea called random projection. As with traditional LSH, we define two parameters b and r, where b represents the number of hash functions in a band, and where r represents the number of bands. For each of the $b \cdot r$ hash functions, we define a random hyperplane in d-dimensional space (where d is the size of a pitch vector)

$$h_i(x) = \frac{(a \cdot x) + c}{b} \tag{2}$$

where $a \sim \mathcal{N}(0,1)^d$ and $c \sim Unif(0,b)$.

The hashed value of a vector x is determined by which side of each random hyperplane it lies. Therefore, identically hashed vectors should be relatively close together in d-dimensional space. In [11], we see that E2LSH is able to significantly speed up computation time and performs better than alternative algorithms such as SRS.

4.2 MLSH

Melodic LSH, or MLSH, is a slight variation of E2LSH which includes dynamic time warping and recursive alignment (see Section 3). It takes in pitch vectors of fixed-length d and returns the closest k matches from a song database. Prior to computing the hash functions, a song database is built by extracting pitch vectors (see Section 2) from a corpus of songs, and saving the set of resulting pitch vectors on disk. As each song produces dozens of pitch vectors, the size of this database is quite large. Next, each pitch vector is hashed according to the random projection idea from Section 4.1. When a query vector p_q is received, we first hash it according to E2LSH and determine which other pitch vectors from the song database have an identical hash value. From this set of candidate pitch vectors, we apply recursive alignment to determine the optimized distance between p_q and each candidate. Finally, the k -closest candidates according to this distance scheme are returned.

5 Ranking Matches

Given a query audio, we first perform pitch vector extraction (Section 2) to construct a set of pitch vectors $\mathbf{p_q} = (p_{q1}, p_{q2}, ..., p_{qn})$. For each pitch vector p_{qi} in $\mathbf{p_q}$, we apply MLSH to determine the k-closest song pitch vectors. For the final implementation, I found that a k value of 2 yielded the best results. After we have found the set of all candidate matching pitch vectors, we compile results for each song by determining the average RA distance of candidate pitch vectors for each song. We then apply the following scoring function to determine a score value S for each song:

$$S(s) = w_1 \cdot \frac{D_s}{D_{max}} + w_2 \cdot \frac{N_s}{N_{max}} \tag{3}$$

In this equation, D_s represents the compiled average RA distance for candidate pitch vectors for song s, and N_s represents the number of pitch vectors for song s that were returned as a match across MLSH runs for vectors in $\mathbf{p_q}$. D_{max} and N_{max} represent the highest D and N values across all songs and serve as constants for normalization. w_1 and w_2 are constants (whose sum is 1) which scale each portion of the score function. The reason we consider number of matches as well as distance in our score function is mainly to prevent outlier or fluke matches from skewing results. For instance, if we consider distance alone, one very close match from an incorrect song could outweigh tens of slightly worse matches from the correct song. It was observed that correct songs often had a high number of matches, and so the N term was included to account for this effect. In the final implementation I found that the values $w_1 = 0.8, w_2 = 0.2$ worked best.

Once score values $S(s)$ for all songs are calculated, the final ranking of songs is determined in ascending order of score.

6 Results

See this link for example results with audio.

6.1 Data

Two main datasets were used to test the performance of this QBH model. The first was a dataset of 109 popular songs gathered from a YouTube playlist. This dataset mainly consists of pop music hits from the last 10 years. Queries for this dataset were collected by myself. In total, 29 hummed queries were collected from friends and family. All queries were collected where users hummed melodies entirely from memory. The second dataset used was Roger Jang's corpus of queries and melodies called MIR-QBSH (available online). This dataset consists of 48 MIDI melodies of common folk-songs and traditional melodies. In addition, this dataset includes 2797 hummed queries collected over a span of six years. This dataset is commonly used a benchmark for performance in QBH algorithms in literature. (Note: for MIDI melodies, CREPE and `librosa` were not needed as we were no longer dealing with audio input).

6.2 Results and Tables

For the popular song data, the QBH algorithm reached a mean reciprocal rank (MRR) of 0.69 and a Top-3 hit rate of 75% (MRR and Top-3 hit rate are commonly used as primary QBH performance metrics as in [1]). Although these specific songs have not been tested by other QBH algorithms, this result exceeded my expectations for this project given its limited timeframe and resources.

While not a direct comparison, the model proposed in [1] by Ryannen et al. also tested their QBH model on popular songs. They converted 427 pop songs to MIDI format and collected queries on 32 of these songs. In their experiment, some queries were collected where users hummed melodies from memory, and some queries were collected where users hummed along to the song. Their model was able to achieve a 58% Top-3 Hit Rate using this data.

For the Jang corpus, however, results were not as promising. My QBH model achieved a 51% Top-3 hit rate on MIR-QBSH queries. Ryannen et al. achieved a Top-3 hit rate of 90% in [1] on the same data. As this model was primarily focused on performance on the popular songs dataset, perhaps this could explain the model's failure on the MIR-QBSH data. Next steps for this project would mainly focus on improving performance on this dataset through further experimentation with RA and DTW as well as fine-tuning parameters.

6.3 Improvements of LSH

The use of locality sensitive hashing in this project helped reduce computation time of the algorithm without decreasing accuracy significantly. On the popular song dataset, LSH sped up computation by approximately 34% while compromising at most 5% accuracy when compared to a brute force approach. Figure 6 demonstrates the execution time of LSH and brute force as a function of the size of the song dataset. (To simulate results for large dataset sizes, duplicates of the 109 popular songs were used). As we can see from the figure, the difference in time between the two approaches increases as dataset size increases. On a 981 song dataset, the brute force approach was approximately 2.24x slower than the LSH approach. For real-world applications of this algorithm, which would involve potentially thousands or even millions of songs, these results show that using locality sensitive hashing would be essential.

6.4 Future Improvements

In addition to the fine-tuning of parameters, an interesting area for potential improvement is the use of deep learning. Several deep learning methods have been used in QBH contexts in recent literature. [12] proposes a semi-supervised model training pipeline for the QBH task, achieving a Top-10 hit rate of 92% on the Jang corpus. [13] uses a Deep Belief Network (DBN), a type of generative model, paired with dynamic time warping. On a custom corpus of 100 popular Korean and Chinese songs, the DBN model achieves a Top-3 hit rate of 72%.

Learning from some of these approaches, it seems as though some improvements can be made to this QBH system through the introduction of deep learning. Future work with this project would likely involve experimenting with a generative model which encodes audio input into some latent representation. Comparison with other songs would be done using some distance metric in latent space.

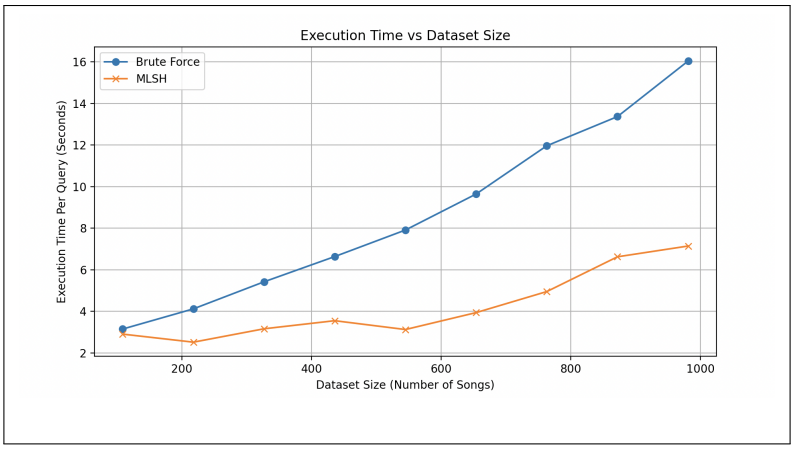

Figure 6: Execution time of brute force and LSH approaches compared to dataset size

Table 1: QBH results on each dataset, N= number of queries, D_{sz}= number of songs in dataset

Corpus	N	D_{sz}	MRR	Top-X hit rate (%)				
				1	3	5	10	20
MIR-QBSH	200	48	0.433	35	51	54	N/A	N/A
Popular Songs Dataset	29	109	0.690	62	75	75	86	89

In addition to deep learning, it would be interesting to experiment further with some temporal alignment techniques as described in [2] - for instance, more complex combinations of linear scaling with recursive alignment. Overall, these improvements might be able to improve the accuracy of this query by humming model.

7 Conclusion

This paper proposes a query by humming algorithm using recursive alignment and locality sensitive hashing that achieves promising results on recent popular songs. The model is able to take in audio as input for both queries and songs and output a ranking of songs. While the model is able to achieve a Top-3 Hit Rate of 75% on a dataset of 109 popular songs, it struggles to match state-of-the-art results on the MIR-QBSH dataset. To fix this, more fine tuning of parameters could be done. Additionally, experimentation with more techniques like RA such as those proposed in [2], or deep learning techniques could lead to better MIR-QBSH performance. Overall, the project was successful in its goal to create a query by humming algorithm which could perform well on popular music. The algorithm, through LSH, is able to handle a large number of songs and is well suited to scalability. This project serves as an example of how powerful locality sensitive hashing can be. Additionally, both dynamic time warping and recursive alignment were demonstrated to be successful temporal matching algorithms, each greatly increasing the performance of the algorithm.

Overall, this study successfully demonstrates a scalable and accurate query by humming system, performing best on popular music, which combines locality sensitive hashing with recursive alignment.

References

[1] Ryynanen, M & Klapuri, A. (2008) Query By Humming of MIDI and Audio Using Locality Sensitive Hashing., *International Conference on Acoustics, Speech, and Signal Processing*,

[2] Wu, X., Li, M., Yang, J., & Yan, Y. (2006). A top-down approach to melody match in pitch contour for query by humming. In Proceedings of the International Conference on Chinese Spoken Language Processing.

[3] Guo, Z., Wang, Q., Liu, G., & Guo, J. (2013). A query by humming system based on locality sensitive hashing indexes. Signal Processing, 93(8), 2229-2243.

[4] Kim, Y., & Park, C. H. (2013). Query by Humming by Using Scaled Dynamic Time Warping. In 2013 International Conference on Signal-Image Technology & Internet-Based Systems (pp. 1-5). Kyoto, Japan.

[5] Frank, C. (2020, November 12). The machine learning behind 'Hum to Search'. Google AI Blog. Retrieved from https://research.google/blog/the-machine-learning-behind-hum-to-search/

[6] Kim, J. W., Salamon, J., Li, P., & Bello, J. P. (2018). CREPE: A Convolutional Representation for Pitch Estimation. In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP).

[7] Jansson, A., Humphrey, E., & Montecchio, N. (2017). Singing voice separation with deep U-Net convolutional networks. Paper presented at the 18th International Society for Music Information Retrieval Conference, October 23-27, Suzhou, China.

[8] Andoni, A., & Indyk, P. (2005). LSH algorithm and implementation (E2LSH). Retrieved from https://www.mit.edu/ andoni/LSH/

[9] Lv, Q., Josephson, W., Wang, Z., Charikar, M., & Li, K. (2007, September). Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In Proceedings of the 33rd International Conference on Very Large Data Bases (pp. 950-961).

[10] Chow, R., Pathak, M. A., & Wang, C. (2012). A practical system for privacy-preserving collaborative filtering. In Proceedings of the 2012 IEEE 12th International Conference on Data Mining Workshops (pp. 547-554). Brussels, Belgium.

[11] Nakanishi, Y., Hiwada, K., Bando, Y., Suzuki, T., Kajihara, H., Sano, S., Endo, T., & Shiozawa, T. (2023). Implementing and evaluating E2LSH on storage. In Proceedings of the 26th International Conference on Extending Database Technology (EDBT) (pp. 437-449).

[12] Amatov, A., Lamanov, D., Titov, M., Vovk, I., Makarov, I., & Kudinov, M. (2023). A semi-supervised deep learning approach to dataset collection for query-by-humming task.

[13] Sun, J.-q., & Lee, S.-P. (2017). Query by Singing/Humming System Based on Deep Learning. International Journal of Applied Engineering Research, 12(13), 3752-3756.