Querying Large Collections of Music for Similarity

Matt Welsh, Nikita Borisov, Jason Hill, Robert von Behren, and Alec Woo
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720, USA {mdw,nikitab,jhill,jrvb,awoo}@cs.berkeley.edu

Abstract

We present a system capable of performing similarity queries against a large archive of digital music. Users are able to search for songs which "sound similar" to a given query song, thereby aiding the navigation and discovery of new music in such an archive. Our technique is based on reduction of the music data to a feature space of relatively small dimensionality (1248 feature dimensions per song); this is accomplished using a set of feature extractors which derive frequency, amplitude, and tempo data from the encoded music data. Queries are then performed using a k-nearest neighbor search in the feature space. Our system allows subsets of the feature space to be selected on a per-query basis.

We have integrated the music query engine into an online MP3 music archive consisting of over 7000 songs. We present an evaluation of our feature extraction and query results against this archive.

1 Introduction

Digital music archives are an increasingly popular Internet resource, supported by a surging market for personal digital music devices, an increase of bandwidth to the home, and the emergence of the popular MP3 digital audio format. Music archive sites such as `mp3.com` and search engines such as `mp3.lycos.com` are changing the face of music distribution by archiving and indexing a vast array of digital audio files on the Internet. Users of these sites need a way to navigate and discover music files based on a variety of factors. Many archive sites offer text-based searching of artist, song/album title, and genre. However, discovering music based on textual indexing alone can be difficult; for example, `mp3.com` categorizes its 208,000 songs into 215 separate genres. Discovering new music amounts to downloading (and listening to!) an arbitrary number of songs which match a particular text search, which could potentially number into the thousands.

In this paper, we present a system enabling *similarity queries* against a database of digital music. Our system allows a user to submit a query which, given a particular song in the database, returns a user-supplied number of other songs which "sound similar" to the query song. Our technique is based on a set of *feature extractors* which preprocess the music archive, distilling each song's content down to a small set of values (currently, 1248 floating-point values per song). Similarity queries can then be performed using a k-nearest-neighbor search in the feature space.

We have developed a number of domain-specific feature extractors, described in Section 3, which process audio data stored in MP3 format. These extract data from the music such as frequency histograms, volume, noise, tempo, and tonal transitions. The choice of feature extractors is essential for music similarity matching to work effectively. We have found that certain combinations of features (for example, low-frequency plus tempo data) are highly selective for certain types of music. Classical and soul music are particularly well-selected by our feature extractors, as detailed in Section 5.

We have incorporated our query engine into the Ninja Jukebox [6], a scalable, online repository of over 7000 songs totalling over 30 gigabytes of MP3 data. The Jukebox client application, described in Section 4, allows the user to select, listen to, and perform queries on the entire database. An additional feature provided by our system is the ability to select subsets of the feature space on which to perform queries, thus facilitating the investigation of individual feature extractors. The query engine performs well, with a response time of less than 1 second for a music database consisting of 10000 songs and 1248 feature dimensions per song.

This paper makes three major contributions.

This research was sponsored by the Advanced Research Projects Agency under grant DABT63-98-C-0038, and an equipment grant from Intel Corporation.

First, we present a full-featured search facility for digital music based on acoustic properties alone. Second, we present a number of novel feature extraction techniques for digital music, specifically tailored for performing similarity queries. Third, we evaluate our results against a reasonably large music archive containing a diverse assortment of genres.

2 Design Overview

The music query engine operates in two stages. The first stage consists of a relatively expensive preprocessing phase, wherein each song in is passed over by a number of independent feature extractors. Each feature extractor reduces the information content in the raw music data to a vector in a small number of dimensions (typically one or two, but one extractor produces as many as 306 dimensions per song). The music data is stored in MP3 [10] format, however, the feature extractors generally operate on raw audio samples produced by decoding the MP3 file. The features associated with each song are stored in a simple text database for later processing by the query engine. Each feature extractor has its own database file associated with it, allowing the extractors to run independently and in parallel.

While the preprocessing step is expensive, it must be executed just once for each song; when new songs are added, the preprocessor is capable of only processing new entries rather than rerunning across the entire set. Each feature extractor takes approximately 30-45 seconds to run against a 5-minute song on a 500 MHz Pentium III system; the longest run time (for tempo extraction) is about 5 minutes.

The query engine reads the set of feature databases into main memory, storing each feature vector as an array of floating-point values. The set of vectors for a particular song treated as a point in an n-dimensional Euclidean space. The basic query model assumes that two songs S_1 and S_2 sound similar if their feature sets v_1 and v_2, respectively, are within distance ε as computed by the Euclidean distance:

$$d = \sqrt{\sum_i \left(v_1^i - v_2^i\right)^2}$$

where v_1^i denotes the ith component of v_1, i ranges over $(0 \ldots |v_1|)$, and $|v_1| = |v_2|$.

While it would be interesting to explore other similarity metrics for music, Euclidean distance is attractive for our feature extractors because it is intuitive both when implementing a feature extractors and when querying the database. One concern with this approach is that the magnitude of feature vectors must be normalized to prevent one feature from "weighting" the distance function more than others. For the results presented here, all features were normalized in the range $(0.0 \ldots 1.0)$. However, it may be desirable for the user to assign scalar weights to given feature vectors, in essence assigning them relative priorities in terms of computing the Euclidean distance between two songs. We have not yet explored this possibility in our prototype.

Rather than have the user specify the desired clustering distance ε, the query application allows the user to specify a value k indicating the number of similar songs to return for a given query. Given a query song Q and a value for $k > 2$, the similarity query is performed by a k-nearest-neighbor search in the n-dimensional space of song features. Various algorithms for computing nearest neighbors exist; our prototype uses a straightforward algorithm with early rejection. Section 4 discusses our implementation in more detail.

Because one of the goals of our system is to facilitate investigation of different feature extractors, the query engine allows the user to perform queries across a subset of the feature space. Each vector can be independently indexed by a text string identifying that feature; for example, tempo data is stored with a tag of "`av_tempo`". The query application allows the user to select the features used in the nearest-neighbor matching by specifying one or more of these feature tags. This allows the user to compare queries using different subsets of the feature space, in order to better understand the effect of each feature on the results of a query.

3 Feature Extraction

To be able to search through a collection of many gigabytes worth of music, we reduce each song to a collection of *features*. A feature is a small vector, which captures some aspect of a song which can be used to determine similarity. This is somewhat difficult because of the large amount of detail present in a given song. A key challenge is to remove extraneous detail that might overshadow similarities between songs, while retaining enough detail to effectively discriminate between qualitatively different songs. We achieve this by ensuring that each feature is very simple. It is more useful if a feature mistakenly classifies dissimilar songs as similar, than vice versa, since using a combination of features will allow us to eliminate these false matches.

This section describes the features we chose to ex-

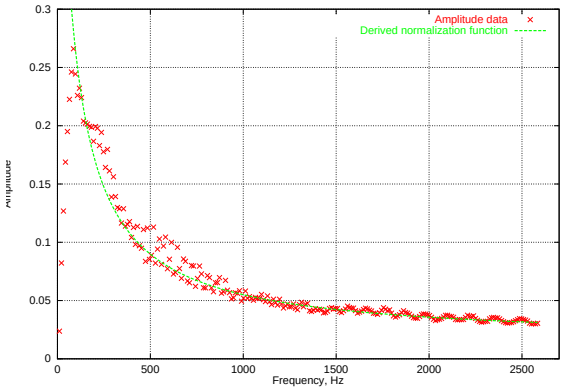

Figure 1: *Derivation of normalization function ν based on frequency averages.*

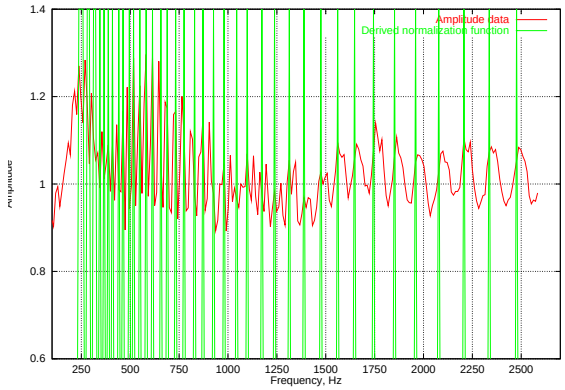

Figure 2: *Normalized frequency amplitudes and the notes of the Western scale.*

tract, as well as the mechanisms for extracting them. The types of data captured are tonal histograms, tonal transitions, noise, volume, and tempo.

3.1 Tonal histograms

The first feature extractor produces a histogram of frequency amplitudes across the notes of the Western music scale. Each bucket of the histogram corresponds to the average amplitude of a particular note (e.g., C sharp) across 5 octaves. This information can be used to help determine the key that the music was played in, as well as any dominant chords. It also has the property of determining how many different frequencies were active in the sample.

Frequency analysis of this sort should compensate for the characteristics of the human ear, which is more sensitive to high frequencies than to low ones. As such, music tends to exhibit greater amplitudes in the low frequency range. Therefore, we must attenuate the energy levels of the music before performing frequency analysis, in order to perform comparisons using the perceived frequencies that a human would hear. Otherwise, low notes would dominate the amplitudes in the frequency histogram.

The tonal feature extractor determines frequency amplitudes by performing an FFT on 0.1 sec samples for 16 sec in the middle of the song. The frequency amplitudes A are then normalized using the function:

$$A'_\phi = \frac{A_\phi}{\nu(\phi)}$$

where the normalization factor ν is

$$\nu(\phi) = \frac{43.066}{75.366 + \phi} + 0.015$$

which is parameterized by ϕ, the frequency of the sample being normalized. ν was determined by applying an inverse linear regression against amplitude-versus-frequency data averaged across 100 random songs, as shown in Figure 1. Another way to determine ν would be to use established models of frequency sensitivity in the human ear, as in [8]. However, determining this function empirically allows us to adapt it to actual music characteristics, which often exhibit some compensation for recording and amplification technology.

Figure 2 shows a plot of normalized frequency ranges across the same set of 100 songs used to determine ν. Superimposed onto this figure are the frequencies of notes in the Western musical scale; we can see that frequency peaks occur clearly at the corresponding notes.

Two types of information are extracted from the normalized frequency data. The first is the *low frequency average*, which is the average amplitude of normalized frequencies from 40Hz to 1KHz. This feature is meant to distinguish "bass-heavy" music (such as rock and rap) from "lighter" music such as classical and jazz. This value is again normalized to compensate for differences in volume levels between recordings.

The second type of feature is a histogram of tonal amplitudes across multiple octaves, as described earlier. Each bucket of the histogram corresponds to the average frequency amplitude of a given note across 5 octaves, ranging from 130Hz to 4.16KHz, with middle C being 260 Hz. If a measured frequency is within 1/2 of a Hz of a note in the musical scale, it is counted as that note. We ignore frequencies less that 130 Hz because the precision of the FFT below that frequency range is not ac-

curate enough to distinguish between two adjacent notes. Each interval of the FFT is 10Hz, yet at 100Hz, only 5Hz separates the musical notes.

3.2 Tonal transitions

Music, to some approximation, can be summarized as a collection of frequency transitions over time. Gibson [5] has investigated the use of frequency transitions to identify individual songs given a short sample; applying these techniques for similarity matching is the focus of our tonal transition feature extractor. This extractor produces a set of 306 values, each of which corresponds to the total number of tonal transitions in a given frequency range for a 10-second sample of the song. Five such feature vectors, corresponding to a 50-second sample, are extracted for each song.

The tonal feature extractor works as follows. Let a song be modeled as a finite sequence of samples S_t. A parameter s can be used to skip s samples from the beginning of the song. The actual implementation sets s to correspond to 20 seconds. S_{t+s} is then segmented into a sequence of blocks B_k with block size b where $0 \leq k \leq \lfloor \frac{|S|-s}{b} \rfloor$. FFT is applied to each block and only the amplitude component is used. Thus, $\mathcal{B}_k = |FFT(B_k)|$

The frequency amplitudes from the FFT are mapped onto the Western musical scale for 6 octaves ranging from 53.8Hz (A 3 octaves below middle C) to 3.43kHz (A 3 octaves above middle C). This range was chosen as it captures the majority of the frequencies emitted by acoustic instruments as well as the human voice, ignoring high-order harmonics [9].

Tonal transitions are determined by comparing frequency amplitudes between two consecutive blocks, namely from $\mathcal{B}_k$ to $\mathcal{B}_{k+1}$. Let $f_\phi(k)$ equal the amplitude of frequency ϕ in $\mathcal{B}_k$. If

$$\frac{f_\phi(k+1) f_\rho(k)}{f_\rho(k+1) f_\phi(k)} > 1 \tag{1}$$

there exists a frequency transition from ϕ to ρ from block $\mathcal{B}_k$ to block $\mathcal{B}_{k+1}$. Since a relative value is used for determining frequency transition, effects from white noise and band filtering can be avoided.

A large block size b corresponds to high resolution in the frequency domain. However, a large b will not capture tonal transitions at short time scales. We choose $b = 8192$ which corresponds to 0.18 sec at a sampling rate of 44.1kHz. As a result, frequency transitions faster than 0.18 sec will not be captured. However, we believe that this resolution is adequate to capture the primary tonal properties of music.

Let n be the number of notes that can be captured per octave and o be the number of octaves to be examined. Across the o octaves, the number of all possible tonal transitions is

$$d = o \times n \times (o \times n - 1)$$

Let TT be a function that applies the tonal transition equation given in Equation (1) d times over all possible tonal transitions and stores each occurrence on a bit vector $\vec{v}$ with length d. That is,

$$TT(\mathcal{B}_k, \mathcal{B}_{k+1}) \rightarrow \vec{v} \in \{0,1\}^d$$

Since $d = O((o \times n)^2)$, it is desirable to make n smaller, in order to reduce the dimensionality in the feature space. In our case, each octave is divided into 3 segments each, yielding 18 tones. As a result, $d = 18 \times 17 = 306$.

By applying TT to the entire sequence $\mathcal{B}_i$, a sequence $\vec{V}$ of bit vectors is generated. If two songs are similar, the $\vec{V}$ of the two songs should also be similar using a metric like hamming distance. However, this approach is not effective since it requires a pairwise comparison and is prone to phase errors.

One naive method is to count the number of tonal transitions over a fixed period T. If two songs are similar, they should have similar number of tonal transitions over T. The tradeoff is that temporal information is lost due to summation.

Let FT be the feature extractor function which performs vector summation over a sequence of vectors and $\vec{F}$ be a sequence of feature vectors.

$$\vec{F}_K = FT(\vec{V}_{K \times T \ldots (K+1) \times T - 1})$$

where $0 \leq K \leq \lfloor \frac{|S|-s}{b} \rfloor$.

Since error due to summation approximation is cumulative, error increases as T increases. Thus, T should be tuned by experiment. In our case, T is chosen to correspond to 10-second samples of a song, and F contains at most five feature vectors which correspond to 50-second samples of a song.

3.3 Noise

Another good measure of how similar two songs are is their relative noise level. It's unlikely that a person will characterize a song with pure sound similar to one with a lot of noise. To convert this to an objective metric, though, it is necessary to define what is meant by a "noisy" or a "pure" sound. A sensible definition should recognize a pure tone (i.e. a perfect sine wave) as pure, and random noise (e.g. random samples) as highly noisy.

Examining these two extremes, we notice that a pure tone can be identified by a highly regular Fourier transform. In the ideal case, it consists of a single spike at the corresponding frequency. The Fourier transform of random samples, on the other hand, should exhibit no visible structure. In general, we can define a pure sound as one where only a few dominant frequencies are present in the Fourier transform, and a noisy one where a large number of frequencies are present.

Now we are faced with the task of using this definition to algorithmically reduce a sound sample to a real number representing its noise level. Simply counting the number dominant frequencies is problematic, since it's not always clear which frequencies are dominant, and which are not. For example, a discrete Fourier transform of an instrument such as a violin playing a single tone will produce a peak at the frequency of the tone, as well as smaller peaks at several nearby frequencies. Similar difficulties arise in examining a Fourier transform of a sample with several instruments playing distinct notes, some louder than others.

We decided to count peak frequencies by weighting each frequency by the magnitude of the corresponding Fourier coefficient; for example, one loud tone and two quieter ones with half the magnitude will count as a total of two peaks. Mathematically, we simply compute the sum of the magnitudes of the coefficients corresponding to each dominant frequency. A useful observation is that a non-dominant frequency would have a nearly negligible contribution to such a sum, since its magnitude is much smaller than that of a dominant frequency. Because of this, we do not perform thresholding to select the dominant frequencies, and add up all the Fourier coefficients instead. We then normalize the sum according to the maximum Fourier coefficient, so that songs which are recorded at a louder volume will not appear more noisy. Therefore,

$$noise(f) = \frac{\sum_{i=1}^{n} |\hat{f}(i)|}{\max_{i=1}^{n} |\hat{f}(i)|}$$

where n is the highest coefficient calculated by the discrete Fourier transform. This metric applied to a pure tone will return 1, and applied to a random set of Fourier coefficients will return approximately $1/2n$. A loud pure tone combined with some random noise at a half the volume will result in $1/4n$ — more noisy than a pure tone, but less noisy than just random noise. In general, the behavior of this metric closely approximates the subjective notion of how noisy a sound is.

Of course, since a Fourier transform is taken over a small sample of a song, we need to extend this metric to the entire song. The average noise level over the entire song is an obvious number to extract. The maximum level is also useful, since a song which has some noisy parts and some melodious ones should be classified as dissimilar from a song which has a uniform medium noise level. This is especially useful since most songs have quiet intro and outro sections, which tend to be less noisy. The standard deviation of the noise level across the song is also an interesting number — it distinguishes songs which are uniformly noisy from more varied ones. We use all three of these numbers as separate feature-space dimensions.

3.4 Volume

It is potentially useful to measure the variations in the volume level of a song. A primitive way to calculate the volume of a sound sample is to calculate the average change in the threshold value:

$$volume(S) = \sum_{i=2}^{n} |S_i - S_{i-1}|/(n-1)$$

(where S is a vector representing the raw sound samples). This function does not necessarily directly correspond to the DSP literature notion of volume, but it does have the property that it returns higher values for loud songs and lower ones for quiet ones. For our purposes, this is sufficient.

As with the noise function, we can calculate the maximum and average volume levels, as well as the standard deviation. However, the average volume level is more reflective of the recording equipment used, rather than the style of the song. For example, two copies of a song, with one recorded at half the volume, will produce distinct results in all three of the metrics. This is especially problematic because there is a large amount of variance in the average volume level of commercially available recordings. It is therefore more useful to normalize these values with respect to the average volume level. We thus obtain the *relative volume* metric, which contains the maximum volume, and the standard deviation relative to the average volume level. This metric better identifies the amount of variation in volume of a song. The absolute volume metric may still be useful for purposes of creating a mix of songs: a selection of many different recordings will likely have songs with significantly different volume levels. One can either use the absolute volume data to pick songs with similar volume levels, or to aid in performing automated volume adjustments.

3.5 Tempo and Rhythm

The specific rhythmic qualities of a piece of music have a good deal of influence on how humans experience it. Fast songs are often upbeat and energetic, while slower songs are often more peaceful. The syncopated rhythms of jazz songs feel very different from the more straightforward even tempos of rock.

Extracting rhythmic information from raw sound samples is a difficult task. Several studies have focused on extracting rhythmic information from digital music representations such as MIDI, or with reference to a musical score [16]. Neither approach is suitable for analyzing raw sound data. For the purposes of our analysis, we adopted the algorithm proposed by Scheirer [17]. This technique breaks an input signal into several bands, each representing one-octave ranges. The algorithm then uses banks of resonators on each input to settle in on the position of the downbeat over time.

The output of this algorithm allows us to extract several interesting features from a music sample. First, we can determine the average tempo of a song. Additionally, we can examine how the tempo varies throughout the song. Finally, as a measure of the rhythmic complexity of the song, we track how well the algorithm was able to settle down to a particular tempo.

The tempo feature extractor determines the tempo of the song from three 10-second samples, at 60, 120, and 180 seconds into the song.[1] It then outputs 4 numbers: the average tempo of the 3 samples, the spread and deviation of the tempo across samples, and the fraction of samples for which the tempo extraction appeared not to work well, as indicated by the inability of the algorithm to settle in on a steady beat by the end of the 10-second sample. Note that it is impossible to measure the actual effectiveness of the algorithm for every sample, since this involves listening to the sample and deducing the tempo manually. In the samples that we listened to, however, a wide variation in the output of the tempo algorithm typically coincided with the failure of the algorithm, often due to complex rhythms or noisy music.

[1] For shorter songs, additional samples are chosen by halving the start position of the sample until chosen start point is at least 10 seconds from the end of the song.

4 The Jukebox Query Engine

In order to study the effectiveness of our feature extraction and similarity-matching algorithms, we have integrated the query engine into the Ninja Jukebox [6], a scalable online digital music repository consisting of nearly 7000 songs stored in MP3 format. The Ninja Jukebox consists of a cluster of workstations each hosting a collection of music data on local disk, as well as a Java-based scalable service platform, the MultiSpace [7], which manages cluster resources and facilitates application construction and composability. The Jukebox service itself is entirely coded in Java, using Java Remote Method Invocation (RMI) for communication between distributed components (e.g., the music locator service on each node of the cluster).

The Jukebox user interface is a Java application which establishes a connection to a centralized Jukebox directory service to select songs, and which receives MP3 audio data streaming over HTTP to an external player application. Access to both the music directory and each song in the Jukebox requires access controls based on client authentication using public key certificates. The Jukebox also includes a simple collaborative-filtering feature allowing users to express song preferences.

Integrating the query engine into the Ninja Jukebox consisted of two steps: first, adding the similarity query functionality to the Jukebox service itself, and second, enhancing the Jukebox user interface to accept queries and return results.

The Jukebox Query Engine is implemented in Java and runs alongside the other Jukebox subservices in the MultiSpace cluster environment. When the engine is started, it reads the feature database (stored as a collection of text files) and exports a Java RMI interface with a number of methods which can be invoked by a client application. These methods are:

- `int numSongs()` — return the number of songs in the database;
- `String[] getFeatures()` — return a list of feature tags;
- `String[] getSongNames()` — return a list of songs;
- `getSong(String name)` — return the `Song` data structure for a given song;
- `Song[] query(Song query, int k)` — return the k nearest songs which match the query song;

- `Song[] query(Song query, String features[], int k)` — return the k nearest songs which match the query song, using only the given features.

Having the Query Engine export a simple RMI interface allows multiple client applications to be built which are able to perform remote queries on the entire Ninja Jukebox.

Feature extraction from the Ninja Jukebox is performed in parallel, with each node of the Jukebox running each of the feature extractors against the subset of the music database stored on that node. The scripts performing the feature extraction are engineered to be highly resilient to failures in the feature extractors themselves, as well as in the underlying system software. The feature extraction scripts perform checkpointing so that they may be stopped and restarted at any time without loss of data. This functionality is vital considering the expense of losing feature data (i.e., re-running the feature extractors, which are slow).

4.1 Query algorithm

The Query Engine performs similarity matches on the database using a brute-force k-nearest neighbor algorithm. Our nearest-neighbor algorithm keeps track of the current result set of size k, and is optimized to quickly reject points which have a distance exceeding the maximal distance of points in this set. This early rejection prevents the n-dimensional distance from being fully computed for each point.

This algorithm requires at most $O(n \cdot d)$ operations where n is the number of songs in the database and d is the number of feature dimensions stored per song. Clearly there are more efficient algorithms for computing nearest neighbors. Most deterministic algorithms have a query time of at least $\Omega(\exp(d) \cdot \log(n))$ [4, 1], while Kleinberg's ε-approximate algorithm [12] has a query time of $O((d \log^2 d)(d + \log n))$ and a preprocessing step which requires $O((n \log d)^{2d})$ storage.

However, we feel that our use of a brute-force technique is reasonable for two reasons. First, it performs very well given the size of the Ninja Jukebox, with a query time of less than 1 second in most cases. Secondly, our algorithm allows queries on subsets of the feature space. Many of the approximate nearest-neighbor algorithms require preprocessing of the data set which lose information which is needed to independently identify feature dimensions for such subset queries. Subset queries are an important tool for studying the effectiveness of our feature extractors, so we are willing to trade performance for this functionality. Moreover, the design of our system makes it easy to "drop in" different nearest-neighbor algorithms with relative ease.

songs	features	time
2500	10	4 msec
2500	24	7 msec
2500	100	22 msec
2500	1000	189 msec
2500	1248	235 msec
10000	10	18 msec
10000	24	29 msec
10000	100	85 msec
10000	1000	786 msec
10000	1248	948 msec
50000	10	91 msec
50000	24	143 msec
50000	100	426 msec
50000	1000	3806 msec
50000	1248	14738 msec

Figure 3: *Performance of Query Engine on synthetic data set.*

Another issue is large amount of storage required by the approximate algorithms. For interactive response times, it is desirable to trade CPU utilization for memory, in order to keep the entire database in core. We feel that our feature extractors have reduced the dimensionality of the search space sufficiently that more expensive (and also more straightforward) nearest-neighbor algorithms are adequate.

Figure 3 shows the performance of the Query Engine using a synthetic data set consisting of 2500, 10000, or 50000 songs with a varying number of dimensions stored per song. The feature vectors are initialized with random values in the range $(0 \ldots 1)$. The total number of songs in the real Jukebox Query Engine is 7090, with 1248 feature dimensions stored per song. All but 24 of those dimensions are tonal transition features. All measurements were taken on a dual-processor 500 MHz Pentium III system running Linux 2.2.13 and IBM JDK 1.1.8, which employs a highly optimized Java JIT compiler, the same configuration running the actual Jukebox Query Engine.

Note that the performance of the Query Engine is roughly linear in $(n \cdot d)$. If the frequency-transition features are left out of the database, performance is very good with a query time of less than 150 msec even for a Jukebox of 50000 songs — 7 times more than that in the actual Jukebox. Incorporating the tonal transition features yields an acceptable query time of less than 1 sec for a 10000-song database,

Figure 4: *The Jukebox Query Engine user interface.*

which rises to 14.7 sec for 50000 songs. The performance discontinuity between 1000 and 1248 features on the 50000-song database (3.8 and 14.7 sec, respectively) is due to operating system paging behavior, since in the latter case the database size exceeds the physical memory of the machine, which is 512 MBytes.

Note that the query algorithm is trivially parallelizable. In our prototype implementation, the Query Engine is centralized, but considering that the Jukebox service itself is distributed on a cluster of workstations, the Query Engine could be as well. In this case the number of songs per engine could be reduced significantly, yielding faster query times even with many feature dimensions per song. In the current Ninja Jukebox, there are 3 cluster nodes storing about 2500 songs each.

4.2 Query Engine User Interface

The user interface for the Query Engine is built into the original Jukebox user interface application. At any time while listening to a song, the user can click the "find similar" button, which displays a window such as that shown in Figure 4. The left panel allows the user to select the features upon which to submit queries, or "All" for all features. The slider allows the number of results to be returned to range from 2 to 100. After clicking "search", the list of matches is displayed in the right panel along with the computed distance from the query song. The user can then select one or more songs from the results list and click "add to playlist" to add them to the Jukebox's list of songs to play.

In addition to the graphical UI, we have implemented a text-based query client allowing queries to be submitted from the command line or embedded in scripts.

5 Results

This section summarizes our results for each of the feature extractors in turn, followed by a quantitative analysis based on genre classification.

Analyzing the effectiveness of our work was difficult because of the inherently subjective nature of similarity. We were not able to perform an end user study; instead, much of the analysis consisted of performing queries using a selection of the features, and then manually examining the results. This provided anecdotal evidence which allowed us to examine how well each feature performed. To be able to perform a quantitative analysis, we hand-classified a set of songs into genres and measured how well we could identify songs within the same genre as similar.

5.1 Tonal histograms

The tonal extraction and analysis is exceptionally successful at determining similar songs when dealing with classical music. This is due to the fact that classical music is composed of pure tones generated by instruments or a well-trained human voice. However, as the tones become less distinct, the frequency analysis begins to perform poorly. This occurs when there are vocal harmonies, as in a cappella music, or notes that are slurred together, as with a jazz trombone. When this occurs, many different frequency ranges are active. Any two pieces that have this characteristic will be registered as similar regardless of the genre of the music.

An an example query over classical music, we searched for songs similar to a particular Bach piece. The results contained songs from that same CD as the top four matches. The query also returned piano solos by other artists. Likewise, a query on a piece by Mozart returned six other Mozart songs. Of the top 10 matches, all but one song were either other pieces by Mozart or piano sonatas by other artists. The result of this query contained many of the other songs on the same album because the entire album was in the same musical key. This led to similar frequency analysis results for all the songs.

In addition to classical music, many forms of pop music work well with frequency analysis techniques. For example, a query against a pop, male vocal song produced results where every song in the top 10 was a male vocal with guitar and drum accompaniment. Along the same lines, a query of ten closest matches against a typical rap song returned two other similar songs by the same artist, seven other rap songs, and one techno song, which was rhythmic and bass heavy. The use of the low-frequency average feature contributed significantly here.

However, when a song containing many vocal harmonies, such as a Beach Boys song, is queried, the results are not as good. The songs returned also

contain many harmonies, but they span many different genres. When the music contains certain instruments, such as harmonicas, a similar effect occurs — the closest matches don't exhibit much similarity, but they all tend to contain harmonicas. This is because many different frequency ranges are active in this type of music, regardless of the exact style.

5.2 Tonal transitions

Query results against tonal transition data for classical, folk, and pop songs are quite promising. That is, they return songs in the same category which share a similar tonal structure.

A query of a classical cello piece by Bach performs very well. The two most similar songs returned are cello pieces by Bach. A number of classical and soft guitar pieces are also in the list. A new age and a techno song are also returned. Despite the fact that different instruments are used, songs that contain similar tonal transitions should be considered similar. This, in fact, is observed from the result.

A query of a folk song also gives promising results. In one such query, all the songs returned are can be classified as folk. Since we are counting the number of tonal transitions over a fixed time period, it is expected that songs with similar temporal and tonal transitions should also be considered similar by this feature.

Pop and new-age songs are often complex and contain different streams of tonal transitions concurrently. The "noisiness" of the tonal transition data means that results for these genres are not very specific; however, other features could be used to help restrict the search within a particular genre.

Electronic and rock music often share similar tonal transitions. Therefore, it is difficult for this feature extractor to find songs specific to these genres, although the tonal transitions are matched well. Also, these genres tend to involve a great amount of noise in the frequency domain, which introduces errors into tonal extraction. It should be possible to couple the use of tonal transitions with the noise extractor for better results.

The limitation of not being able to capture tonal transitions faster than 0.18 seconds does not seem impose significant errors. Nonetheless, it may be interesting to explore the tradeoff between resolution within the frequency domain and the ability to detect faster tonal transitions.

5.3 Noise

The use of the noise level feature produced surprisingly good results. Out of a random sample of queries using this feature, more than half produced a list of songs including one or two songs from the same artist. Among the other songs returned, most exhibit some similarity of style. For example, most of the results for an a capella song query involve a strong vocal component. Similarly, a query of an electronic song results in other music of the same genre.

It was interesting to note that the noise level seems to capture, to some extent, the "instrumentality" of the song. A query of a song with a strong piano presence will result in many other piano pieces; a song with soft vocals and a guitar accompaniment will return many other such songs. This can likely be explained by the fact that the frequency signature of an instrument is definitive enough that the number and size of peaks, which contribute to the noise metric, are similar for songs using this metric. An interesting observation is that there is no apparent distinction between male and female vocals; this is not surprising considering that the feature is not dependent on the frequency of a sound, but rather on its purity characteristics.

The query results are especially good for songs at the extremes of the noise metric. Queries of highly melodic and ambient songs pick other such songs with high accuracy; the same is true for very noisy and harsh songs. Pop rock songs produce generally worse results, presumably because of the high variance in noise within the style. However, in all queries there are some results which would not be classified as similar by any human. For example, a heavy metal song had similar noise level as a '70s pop song, and a Beatles song was classified as similar to a rap song. What's interesting to note, though, is that even among the "incorrect" results from a query, there seems to be high amount of correlation. The heavy metal song query above returned several songs by the same '70s pop band, along with other heavy metal songs.

In general, a result to any given query will return a list of songs, which appears to be comprised of one or more clusters of subjectively similar songs. One of the clusters includes the original query, and others are sometimes related to the original song, and sometimes very different. This behavior can be explained by the fact that the noise feature reduces a song to three real numbers. While those numbers may be able to assist in finding similar songs, there are bound to be cases when significantly different

styles of music will result in similar noise feature. This means that the noise feature can be used to identify subclasses of songs, but it is unable to distinguish between certain subclasses. However, this is still a useful result, since we can use other features to separate these subclasses, hopefully leaving only truly similar songs.

5.4 Volume

The volume level feature produced mixed results. From experiments, there appears to be some correlation between the queries and the results. In particular, the number of songs selected which are by the same artist is higher than that expected in a random distribution. However, the correlation is small enough that it is difficult to analyze the rest of the results from a query. There are many results which are highly dissimilar from the original query, and it is difficult to judge how similar the rest are, due to the subjective nature of similarity.

These results are perhaps not surprising. The variation in volume level, unlike some of the other features, is not something that is very noticeable or memorable to a human; it is difficult to identify a style of music with high or low volume variation. Furthermore, there is a large amount of variance in the feature based on the recording. For example, a live and a studio recording of the same song are bound to have different values for the volume level feature. It may still be possible, however, to exploit the small amount of correlation exhibited by this feature, if it is used in the similarity search but with a low weight relative to other features.

5.5 Tempo and Rhythm

The use of the tempo features was modestly successful. The structure of the tempo extraction algorithm caused it to give poor results for rhythmically active songs: features such as runs of eighth or sixteenth notes, trills, and syncopated rhythms all tended to confuse the algorithm. Additionally, the beat in the extremely fluid songs of Enya and other new age artists tended to be too subtle for the algorithm to detect. Another artifact of the tempo extraction algorithm was that it took a very long time to run. Processing only three 10-second samples required between 3 and 5 minutes. Doing a full analysis of a 4 minute song would take between 24 and 40 minutes, which made this infeasible for our analysis. Combined with the noise inherent in the tempo extraction algorithm, the small number of samples lead to a high degree of variability in the tempo data.

The noise in the tempo data limits the usefulness of the tempo measurements when used by themselves to perform similarity queries. Performing hand measurements of a the query results from a song with an actual tempo of 138 beats per minute, we obtained result songs with actual tempos between 102 bpm and 156 bpm. This gives a rough idea of the variability in this measure.

The spread measurement attempted to capture how the tempo changed through the course of the song. Unfortunately, the noise in the tempo data rendered this useless by itself: a query for a song with a steady tempo throughout produced songs with both steady and variable tempos. The usefulness of this measure was also limited to a certain extent by our data set. The predominance of rock songs (which typically maintain a steady tempo throughout) in our music collection meant that the data set as a whole had little variation on this measure. If we had been able to analyze whole songs, a better measure might have been how many places the tempo predicted in the algorithm seemed to change for an extended period of time.

The final two tempo measurements (the average percent deviation with each sample, and the proportion of samples with high deviation) give an indication of how predictable the tempo was, according to our algorithm. The small number of samples limited the discrimitave power of the proportion of high deviation samples metric. The average percent deviation metric did a reasonable job of identifying rhythmically complex songs, precisely because the tempo extraction algorithm tends to be confused by these songs.

Although the tempo data were not good at identifying "similar" songs by themselves, they were useful in helping to narrow down searches when combined with other features. In our experience, adding average tempo and average percent deviation information to the frequency results seemed to give better matches for songs with complicated rhythms. In the queries we examined, adding these measures improved the effectiveness of the frequency measures by around 10%. The significance of this is somewhat limited due to the subjective nature of the evaluation and the noisiness of the tempo data. However, it does suggest that the tempo data could help fine tune the analysis. We expect that with less noisy tempo data the impact would be more dramatic.

5.6 Genre classification results

Quantitatively measuring the effectiveness of our feature extractors is a difficult task, as the notion of

"similarity" in music is highly subjective. However, it is possible to perform a coarse analysis of our feature extractors based on genre classification.

We have categorized by hand a selection of 100 random albums from the Ninja Jukebox totalling 1225 songs. Each album was placed into one of the genres rock, classical, electronic, soul, pop, folk, or indie. Each song in the list was then queried against this reduced database, using a selection of feature-space subsets. The genre of each song returned in each query was recorded. By comparing the distribution of genres for query results against that of the song database as a whole, a measure of genre selectivity for different feature extractors can be obtained.

For each genre G, we can compute the χ^2 metric:

$$\chi^2(G) = \sum_g \frac{(N_g - n_g)^2}{n_g}$$

where N_g is the percentage of query results in genre g (given a query song in genre G) and n_g is the percentage of songs in genre g. A large value of $\chi^2(G)$ indicates that a particular set of feature extractors is highly selective for songs in genre G. A small value of $\chi^2(G)$ indicates that those feature extractors are non-selective; that is, that the distribution of query results closely matches the genres of songs in the database as a whole.

Genre	**songs**
Classical	7.59%
Electronic	19.43%
Folk	9.63%
Indie	14.61%
Pop	8.82%
Rock	38.29%
Soul	1.63%

Figure 5: *Distribution of songs by genre in Ninja Jukebox subset.*

Genre	χ^2 $(k = 10)$	χ^2 $(k = 20)$	**match**
Classical	2.593	1.765	50.11%
Electronic	0.102	0.049	31.05%
Folk	0.203	0.109	22.71%
Indie	0.244	0.139	31.23%
Pop	0.251	0.110	22.22%
Rock	0.064	0.038	48.74%
Soul	0.947	0.295	13.00%

Figure 6: χ^2 *by genre using low-frequency features.*

Genre	χ^2 $(k = 10)$	χ^2 $(k = 20)$	**match**
Classical	4.153	2.783	61.50%
Electronic	0.171	0.065	34.82%
Folk	0.633	0.436	31.92%
Indie	0.368	0.221	35.39%
Pop	0.378	0.212	25.56%
Rock	0.119	0.078	51.96%
Soul	5.358	2.674	30.50%

Figure 7: χ^2 *by genre using low-frequency, noise, and tempo features.*

Genre	χ^2 $(k = 10)$	χ^2 $(k = 20)$	**match**
Classical	1.536	1.018	39.57%
Electronic	0.120	0.072	28.95%
Folk	0.449	0.300	27.88%
Indie	0.319	0.120	33.45%
Pop	0.635	0.407	30.51%
Rock	0.096	0.086	46.07%
Soul	4.364	2.564	27.50%

Figure 8: χ^2 *by genre using low-frequency, noise, tempo, and tonal transition features.*

Figure 5 shows the distribution of songs, by genre, in the subset of 100 albums chosen from the Ninja Jukebox. Figure 6 shows the $\chi^2(G)$ results for 3 feature vectors representing low-frequency information. Results for $k = 10$ and $k = 20$ are shown. Also shown is the percentage of results which matched the query song genre for $k = 10$. As we can see, both classical and soul albums are selected highly by these features, while electronic and rock albums are not. Combining low-frequency information with noise and tempo data produces somewhat better results, as shown in figure 7. Adding tonal transition data produces somewhat worse results, as shown in Figure 8.

It is apparent that using too many features for a query can decrease selectivity. This is because any error in a feature set is magnified by its number of dimensions; in the case of tonal transitions, which comprises 1224 dimensions, any noise in this feature will have a stronger adverse impact on the results. Testing all combinations of features would help determine which are better for genre classification; we have not yet performed this analysis.

Note that even in cases where χ^2 is low, the percentage of results which match the query song's genre is high — nearly 52% for rock in Figure 7. That is, although the χ^2 might be low due to a close match with the distribution of songs in other genres, the "target" genre hit rate is high. One reason

for the poor selectivity may be that the genre classification is rather coarse. What our analysis here does not capture is the subjective similarity of the query results — for example, whether results from a loud, fast rock song are themselves also loud and fast. Moreover, as a music archive navigation aid, we believe users will be able to tolerate a certain error in the query results.

6 Related Work

Our work utilizes simple similarity matching techniques to find songs with features similar to a query song. The features we have chosen were motivated both by their potential to represent meaningful aspects of human musical experience, and by the feasibility of extracting them from our music data. To the best of our knowledge, similarity matching techniques have not been previously applied to searching acoustic music.

Much of the work on categorizing and describing music does so at a very high level, and can't be applied directly to acoustic music. Kaper [11] presents useful a discussion of ways in which humans experience music, but does not suggest mechanisms for extracting these features from recorded music. There is a broad literature on computer generated music, which also suggests interesting features of music which are important for generation [2, 14]. Unfortunately, we have not found it feasible to reverse this analysis to extract features from recorded music.

Dannenberg et. al. describe techniques extracting high-level style information from MIDI samples [13]. Dannenberg also provides a nice summary of high level music features in [3]. Unfortunately, these works relied on high-level music representations such as MIDI or musical scores. The work which is most relevant for our purposes is Dubnov et. al. [18] which includes an interesting discussion of possible acoustic characterizations of timbre and other features of recorded music. Applying their techniques might yield additional useful features for our similarity analysis.

The problem of extracting tempo from music data has been studied directly in the literature. Eric Scheirer proposes a mechanism for extracting tempo information from MIDI music through the aid of a musical score [16]. The tempo extraction algorithm used in our work was adopted from the more recent work of Scheirer, and works with acoustic music data, rather than MIDI [17] [15].

7 Conclusions

We believe that out music query engine serves as a useful tool for navigating large digital music archives. Rather than relying upon text-based data alone, directly indexing music based on acoustic properties gives the user of a music archive the ability to perform queries with greater information content — namely, another song.

Our feature extractors have demonstrated excellent results, although this success is somewhat difficult to quantify. As with other problem domains, having a human-classified "ground truth" dataset against which to measure results would be helpful. However, subjectively speaking, the application of several straightforward feature extractors — frequency data, amplitude, and tempo information being foremost —- has produced good results.

We believe that this study is unique in that it is the first to apply similarity queries against a large digital music archive. There is much future work to be done in this area. Fine-tuning the existing feature extractors, as well as developing new ones which better encapsulate the aural properties of music from a human listener's perspective, will be essential. Better tools are needed to help develop new feature extractors for music, and to understand the results they produce; this problem is complicated by the inherently temporal and non-spatial nature of music.

Other possibilities include performing queries on data gathered from entire albums as well as artists, rather than on individual songs. Of geometric interest is the investigation of non-Euclidean distance metrics as well as the application of efficient neighbor-finding algorithms. Also, the incorporation of other forms of data into an inclusive music search facility is of interest. Synthesizing a music search engine from acoustic data, text indices, and collaborative filtering data will probably prove to yield the best overall results.

References

[1] K. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Computing*, 17:830–847, 1988.

[2] Jeffrey S. Rosenschein Claudia V. Goldman, Dan Gang and Daniel Lehmann. Netneg: A hybrid interactive architecture for composing polyphonic music in real time. In *Proceedings of the International Computer Music Conference*, pages pages 133–140, August 1996.

[3] Roger B. Dannenberg. Recent work in music understanding. In *Proceedings of the 11th Annual Symposium on Small Computers in the Arts*, pages pages 9–14, November 1991.

[4] D. Dobkin and R. Lipton. Multidimensional search problems. *SIAM J. Computing*, 5:181–186, 1976.

[5] David Gibson. Name That Clip: Content-based music retrieval. `http://www.cs.berkeley.edu/~dag/NameThatClip/`, 1999.

[6] I. Goldberg, S. Gribble, D. Wagner, and E. Brewer. The ninja jukebox. In *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[7] S. Gribble, M. Welsh, D. Culler, and E. Brewer. Multispace: An evolutionary platform for infrastructural services. In *Proceedings of the 16th USENIX Annual Technical Conference*, Monterey, California, 1999.

[8] Gebeshuber I.C. and Rattay F. Modelled human hearing threshold curve, 1998.

[9] PSB Speakers International. The frequencies - and sound - of music. `http://www.psb-speakers.com/frequenciesOfMusic.html`.

[10] ISO/IEC 13818-3. Information Technology: Generic coding of moving pictures and associated audio - audio part. International Standard, 1995.

[11] H. G. Kaper and S. Tipei. Abstract approach to music. In *Preprint ANL/MCS-P748-0399*, March 1999.

[12] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proc. 29th ACM Symposium on Theory of Computing*, 1997.

[13] Belinda Thom Roger B. Dannenberg and David Watson. A machine learning approach to musical style recognition. In *International Computer Music Conference*, pages pages 344–347, September 1997.

[14] B. J. Ross. A process algebra for stochastic music composition. `http://www.cosc.brocku.ca/Research/TechRep/`, February 1995.

[15] Eric D. Scheirer. Pulse tracking with a pitch tracker. In *Proc 1997 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, October 1997.

[16] Eric D. Scheirer. *Using Musical Knowledge to Extract Expressive Performance Information from Audio Recordings*. 1997.

[17] Eric D. Scheirer. Tempo and beat analysis of acoustic musical signals. In *J. Acoust. Soc. Am. 103:1*, pages pages 588–601, January 1998.

[18] Naftali Tishby Shlomo Dubnov and Dalia Cohen. Hearing beyond the spectrum. In *Journal of New Music Research, Vol. 25*, 1996.