Music Recognition Service

Maurice Adler      Charles Collicutt      James Huggett
Marvin Klein       Harry Moore           Motti Strom

Abstract

In this report we describe a music recognition service that aims to automatically identify a song given a short sample from that song. We describe an algorithm to extract the defining features of an audio sample in order to create an expressive signature from any audio input; an algorithm to rapidly find the signature of a sample in an extensive database of song signatures; and a system architecture that integrates these two core components with user-facing interfaces, including a client to record audio from a remote microphone and an Interactive Voice Response application that provides access via the telephone network.

Figure 1: A spectrogram of part of Equation by Aphex Twin
## Contents

1 Introduction .................................................. 3

2 Background ...................................................... 4
   2.1 Existing Commercial Solutions ............................ 4
       2.1.1 Shazam ............................................. 4
       2.1.2 Gracenote ......................................... 4
   2.2 Discrete Fourier Transform ............................... 4
   2.3 Signature Creation ....................................... 4
       2.3.1 Mel frequency cepstral coefficients ............... 4
       2.3.2 Haitsma and Kalker ............................... 5
   2.4 Searching ............................................... 5
       2.4.1 Hausdorff Distances ............................... 5

3 The Analysis .................................................... 5
   3.1 Input .................................................... 5
   3.2 Windowing ................................................ 5
   3.3 Spectral Analysis ....................................... 8
   3.4 Band Pass Filtering .................................... 8
   3.5 Filter Bank .............................................. 10
   3.6 Thresholding ............................................. 11

4 Signature Matching ........................................... 11
   4.1 The Database ............................................ 11
   4.2 Locating Samples ....................................... 11
   4.3 Binning .................................................. 12

5 Implementation ............................................... 13
   5.1 Architecture Overview .................................. 13
   5.2 Analysis Engine ........................................ 13
   5.3 SQL Database .......................................... 13
   5.4 Interfaces .............................................. 15
       5.4.1 Windows Client .................................... 15
       5.4.2 IVR Application ................................... 17

6 Evaluation ..................................................... 18
   6.1 Test Methodology ...................................... 18
   6.2 Statistics .............................................. 18
   6.3 Channels ................................................. 20
   6.4 Transitions ............................................. 20
   6.5 Pure Tones .............................................. 20
1 Introduction

Nowadays music is ubiquitous - whether it be in a nightclub, in our cars or from the television. Frequently we do not recognise a particular song but wish to know what it is, perhaps to purchase it later or to recommend it to a friend. The recording industry also has an interest in identifying songs in order to ensure its intellectual property is respected. Here we describe a system that is able to automatically identify a song from only a short fragment.

There are two fundamental tasks our system must perform. First, given some audio input, it must produce a digital representation of that input. We call that representation a ‘signature’, it is also often known as an audio ‘fingerprint’. Second, given the signatures for a number of music tracks and a short sample from one of those tracks, it must identify the track from which the sample comes. The nature of the second task places some constraints on our solution to the first task: the signatures our system produces must permit fast searching and must allow for noise in the audio input. That is, even though the sample may not precisely match a segment of the track from which it came as it may be corrupted by noise, the signature produced from the sample should be sufficiently similar to the signature of a segment of the track to allow it to be identified. Thus the signatures must be produced by an analysis that identifies the defining characteristics of the audio and is robust enough not to be significantly affected by superficial changes and noise. Relatively small signatures that successfully abstract the important features of the audio will also allow for much faster searching.

In §2, ‘Background’, we describe the current state of the art and the initial paths we took in research. In §3, ‘The Analysis’, we describe the algorithm we have designed for the creation of signatures. In §4, ‘Signature Matching’, we describe our algorithm for locating the signature of a sample in a database of song signatures. In §5, ‘Implementation’, we describe our implementations of the algorithms and the system infrastructure that integrates these two core components with our user interfaces. In §6, ‘Evaluation’, we analyse our techniques and assess our successes and failures. Finally, in §7, ‘Conclusions & Future Work’, we summarise what we have learned from this project and propose possible improvements and further developments of the project.
2 Background

2.1 Existing Commercial Solutions

2.1.1 Shazam

This is the most widely known music recognition service in the UK. Shazam allows a user to search its 3.5 million song database by holding a mobile phone up to the music. The result is sent by text message to the user. Shazam’s database contains hashes of high energy points on the spectrogram. This technique was developed in 2001 by Avery Wang [1].

2.1.2 Gracenote

Audio CDs do not contain any meta-data about their content. The Gracenote Compact Disc Database (CDDB) is an online database of information about CDs, allowing users to obtain album names, track names, artist names, years of release and suchlike. The CDDB uses the “Table of Contents” present on all audio CDs to obtain a (probably) unique list of track offsets for each CD and then employs a fuzzy search to match CDs with their meta-data. Although this is similar in some respects to our system, it does not solve the problem that we aimed to solve.

2.2 Discrete Fourier Transform

The Fourier Transform [2] is a function that converts data between time and frequency domains (see Equation 1). The Discrete Fourier Transform (DFT) works on discrete data. A Fast Fourier Transform (FFT) is an algorithm to perform a DFT in $O(n \log n)$ time. It decomposes a signal into its component sine waves and so can be used to find the amplitudes and phases of different frequencies that make up a sound. The FFT is the heart of our analysis algorithm.

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

(1)

2.3 Signature Creation

2.3.1 Mel frequency cepstral coefficients

Mel frequency cepstral coefficients (MFCCs) [3] are commonly used in speech recognition applications. They are computed by first converting a periodogram to the mel scale and then applying a Discrete Cosine Transform (another transform
in the DFT family) to the result. We make use of this during our analysis (see §3.5).

2.3.2 Haitsma and Kalker

In [4] Haitsma and Kalker present a fingerprint extraction and search algorithm based on bitmaps produced from the differential of points in the spectrogram. Their matching algorithm is based on Hamming distances.

2.4 Searching

2.4.1 Hausdorff Distances

Because the fingerprints can be expressed as point-sets, we initially envisioned using a point set distance measure such as the Hausdorff distance to compare signatures. The Hausdorff function is a max-min function for calculating the distance between two point sets (see Equation 2).

$$h(A, B) = \max_{a \in A} \left\{ \min_{b \in B} \{d(a, b)\} \right\}$$

Finding the closest match to a sample is then a problem of minimising the Hausdorff distance. We used an efficient method developed by Huttenlocher and Rucklidge [5] which is based on tessellating the sample over the space of possible transformations of the model set, which in our case is one-dimensional. The problem with this technique is that the complexity of the search increases linearly with the number of songs in the database.

3 The Analysis

3.1 Input

Our analysis program receives the audio input as pulse-code modulated (PCM) data with 16 bits per sample. The modular design of our system allows us to use various different sources for the audio (e.g. a local microphone, a WAV file, a VoIP interface etc.) as long as we have an appropriate module to pass on the audio for analysis in the simple PCM format. Thus, as far as the analysis program is concerned, the input is a time-series of 16 bit amplitude values.

3.2 Windowing

The first step in the analysis is to divide the input into a sequence of segments or ‘frames’ (the reason for this division will become apparent later.) This process
Figure 2: Windowing

is known as ‘windowing’ as one can imagine each frame to be the segment of the input data viewable through a window, with the window being repeatedly moved along the length of the data to create a series of segments. So, if we consider the input as a vector of amplitude values, this windowing results in a matrix with as many rows as there are frames and the size of each row (the number of columns) will be equal to the size of the window (see Figure 2). Each subsequent operation in the analysis procedure will be applied separately to each row/frame.

However, windowing will cause an error when we later use a Discrete Fourier Transform (DFT) to determine the component frequencies of each frame (see §3.3.) The windowing will cause the DFT to have non-zero values at frequencies other than the real frequency components of the input; these extraneous values are known as ‘leakage’. Leakage can cause two similar types of error. First, if there are two different component frequencies, one with a high amplitude and one with a low amplitude, the leakage from the larger component may mask the smaller component so that it is not detected as a separate frequency. Second, if there are two similar component frequencies then, even if they have similar amplitudes, the leakage from the two components may be enough to make it impossible to resolve them into two separate frequencies. Different windowing techniques vary in their ability to cope with the two different types of leakage error: typically, a technique that copes well with one type of error will cope poorly with the other.

So far, we have described windowing as simply a division of the input data into separate frames. That is actually a specific case of a more general procedure. In general, a window can be thought of as a function that is zero outside a certain interval. Thus, the simple windowing procedure described so far could be thought
of as a function that is 1 during the interval that we wish to retain and 0 everywhere else. If we then multiply our input by that function the result will be a single segment. By repeatedly shifting the interval before multiplication we can obtain a sequence of frames as described above. A window of this sort is known as a rectangular window due to the shape of its graphical representation (see Figure 3a). There are many other well-known windows commonly used in signal processing, each of which has different characteristics.

A rectangular window copes very well with components of similar frequency and amplitude (i.e. it is not subject to the second type of leakage error described above) but it is poor at resolving components of disparate frequencies and amplitudes. There are other windows (such as the Blackman-Nuttall window and the Flat Top window - see Figure 3b) that have the reverse characteristics. We have chosen a moderate window that lies between the two extremes: the Hann window (see Figure 3c, Equation 3).

$$w(n) = 0.5 \left( 1 - \cos \left( \frac{2\pi n}{N-1} \right) \right)$$  \hspace{1cm} (3)

In practice, the input vector is divided into frames as initially described then the resulting frame matrix is right-multiplied by a matrix that represents a Hann window.
The aim of the analysis is to produce results which allow one to map from the signature of a frame to its offset within the song. A problem occurs when the sample being analysed is not aligned in the same manner as the song stored in the database. The misalignment can be up to half the size of a frame. To minimise the effects of this problem we apply an overlapped window, which means that the start of each frame is generated from data that made up the end of the last frame.

Large windows allow us to distinguish between a large number of different frequencies but prevent us from accurately determining the timing of transitions. Small windows have a lower frequency resolution but a greater time resolution. Using a tone generator we played back and recorded tones at 4000 Hz, 7000 Hz and 10000 Hz. Figure 4a shows the spectrogram created by a long FFT window. The lines are very sharp, indicating a high frequency resolution. On the other hand, there is a great deal of overlap between the beginning of one tone and the end of the preceding tone. Figure 4b shows the spectrogram created by a short FFT window. The frequency resolution is much worse but this time the frequency changes are very distinct. We found a good compromise was to use a length of about 100 ms. We calculate the window size based on the sample rate to allow songs sampled at different rates to be compared reliably without resampling.

3.3 Spectral Analysis

The second step in the analysis is to turn the time-series into a frequency-series. For this we employ a Discrete Fourier Transform. We actually use the Fast Fourier Transform [6] algorithm provided by the FFTW [7] library to transform each frame from a vector of real values representing a time-series into a vector of complex values representing a frequency-series. The magnitudes of these values represent the energy present at different frequencies within the samples. The phase information is of little use and is discarded. Subject to the inaccuracies caused by leakage (see §3.2) we now have the component frequencies of the input data. However, our data still contains noise.

3.4 Band Pass Filtering

Many of the frequencies identified by the DFT are not actually useful for matching. We can identify certain frequency bands that contain the vast majority of the noise present in the signal. There are also certain frequencies that we know will not be handled effectively by certain types of hardware and software. For example, low quality speakers and microphones have known deficiencies in their frequency responses - this problem is particularly evident when dealing with signals transmitted by telephone. There are similar known problems with certain software systems: for example, most VoIP codecs capture at only 8 kHz and even
Figure 4: Effect of window size on spectrogram
so-called ‘wideband’ systems such as Skype capture at only 16 kHz. So at this point we take a sub-matrix of our frame matrix, discarding those frequencies that experimentation shows are the least useful. We have found that we achieve the best results by retaining frequencies from 60 to 700 Hz.

### 3.5 Filter Bank

We now use a filter bank to group related frequencies into ‘bins’. The filter bank is represented by a matrix with one column for each bin and one row for each frequency sampled by the DFT. We can then take the scalar product of a column with a frame to produce a value representing the amplitude of the frequencies in that frame that fit inside that bin. Depending on the values we use for each row of the filter bank matrix, we can distribute the bins across the frequency spectrum as we wish. We chose to use a distribution based on the ‘mel scale’, which is a perceptual scale of pitches judged by listeners to be equidistant from each other [8].
We use a triangular filter as used in the computation of MFCCs (see §2.3.1) so that no frequencies are given undue weight. Each filter is normalised to have an integral of 1 to prevent high frequencies with wider filters from dominating (see Figure 5).

After this we simply left-multiply the filter bank matrix by the frame matrix to obtain a new frame matrix whose rows are now ‘bin-series’.

3.6 Thresholding

Finally, we turn the matrix into a signature that we will output. For each frame, we calculate the mean value in that frame. Then, for each value in the frame, we output a 1 if the value is greater than the mean or a 0 if it is not. Thus, if our filter bank had 32 bins, then we will now output one 32 bit number for each frame. This has the effect of further smoothing our data, thus reducing the effects of noise, and of producing a signature that is over 1000 times smaller than the original PCM input, which allows for fast searching over a very large database.

4 Signature Matching

4.1 The Database

Our database can best be thought of as a multi-map. It is a mapping from signatures to tracks and track offsets. For instance, if we look up a signature (of some fixed length) then it will return a pair of numbers (a track number and the offset within that track at which the signature can be found) for every occurrence of that signature.

4.2 Locating Samples

In our implementation the database keys are 32 bit numbers. That is, given a 32 bit signature, it will return a list of pairs, one for each occurrence of that signature in any of the tracks in our database. Each pair consists, as described above, of a track number and the offset within that track at which the signature can be found.

Having run a sample through our analyser we have a list of 32 bit signatures, each one associated with an offset within the sample. Then, for each signature, we do the following:

1. Look up the signature in the database to obtain a list of \( \langle \text{track} \#, \text{offset} \rangle \) pairs.

2. For every pair in the list that we have just received, subtract the offset of the signature within the sample from the offset in the pair.
If we concatenate the lists obtained from each signature, we will have a long list of \((\text{track}\# , \text{offset})\) pairs. However, due to the subtractions performed in the second step above, some of these pairs will be identical. We now find which of the pairs appears most frequently in the list; the track number in this pair will be the track number of the track containing the sample.

To see why this is so, consider a sample with five 32 bit signatures, which we might call A, B, C, D and E, like so:

<table>
<thead>
<tr>
<th>Signature</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
</tr>
</tbody>
</table>

Now suppose that when we look up signature A in the database we find a list of pairs containing the pair \((3, 4)\), meaning that A appears in track 3 at an offset of 4. If the entire sample can be found in track 3 at an offset of 4, then when we look up B we should receive a list of pairs containing the pair \((3, 5)\). Similarly, when we look up C we should receive a list of pairs containing the pair \((3, 6)\). If we subtract the offset of each signature within the sample from the offsets in the pairs, as described in step 2 above, then we will end up with \((3, 4)\) every time. If signature A is also found in some other track then when we look it up in the database we may also receive, say, \((7, 3)\). But, unless signature B is found at \((7, 4)\) (or C at \((7, 5)\) and so on), that pair will only appear once in our list. Thus the pair that appears most frequently corresponds to the closest match.

Because the signatures are checked only for an exact match, correct hits occur only on around 1% of the signatures. However, this is enough to get a very accurate match and allows for extremely fast searching (see §6 for further information). This system is highly robust to noise and can cope with long periods where the music is completely inaudible as long as there are at least 5 to 10 seconds in the sample where it can be heard.

### 4.3 Binning

One phenomenon that is immediately apparent when using this method is that many high-scoring matches are often found right next to each other. This appears to be caused by a combination of certain notes lasting for an extended duration and the misalignment of frames with that duration. We take advantage of this phenomenon by ‘binning’ our results - we count similar offsets together. This is easily implemented by integer division of the offsets by a small number and causes the correct match to stand out even further from the false positives.
5 Implementation

5.1 Architecture Overview

We decided early on that a client/server architecture would be necessary. The client, in whatever form, simply records the audio and sends the audio to the server. The server passes the audio data to the analysis engine to generate signatures, these signatures are then passed to the database for matching, and the results returned back to the client. Rather than design our own communication protocol we decided to use web services with SOAP. In this way any developer, given the WSDL, is able to create their own client to our back-end. The web service and the analysis engine communicate through an anonymous pipe.

We used a modular design for our system. By giving each module a well defined function and interface we achieved our goal of high cohesion and low coupling. This allows us to easily combine different modules in different ways. For instance, we have two different client modules for capturing audio (a Windows client that uses a local microphone and a telephony client) and many more could easily be created by any designer with our API. We were also able to plug different analysis engines into our test framework during the development phase. See Figure 6.

5.2 Analysis Engine

This was written in C++ for speed and to take advantage of the FFTW [7] library and the uBLAS [10] linear algebra library. The Hann window and the filter bank matrices are precomputed by a separate application and stored as resources to reduce startup overheads.

If an FFT is given purely real input, the output will be symmetrical. We take advantage of this redundancy by only computing half the coefficients.

FFTW has a capability to optimise itself for a particular machine by remembering ‘wisdom’ about that machine’s characteristics. Our application takes advantage of this by managing a wisdom file for each machine it is run on.

5.3 SQL Database

Given the matching algorithm described above (see §4) the database consists of a number of (signature, offset, song) triples. Our ultimate target was to match against a library of 500 songs - in total we determined there would be at least 2 million signatures. Rather than design our own database software we decided to go with a well tested commercial solution. Microsoft SQL Server integrates excellently with our web service and contains a useful query analyser that helps
Figure 6: A system schematic
us optimise our queries. We also store metadata for the songs in a separate table in the database.

The signatures of the sample fragment and their offsets are loaded into an in-memory table named ‘#temp’. A stored procedure (see Listing 1) is then executed. The inner select query is the essence of the search algorithm, performing the offset subtraction and grouping together the identical offset/song pairs. The outer select query simply performs a join with the metadata table to return artist names and tracks.

Listing 1: Search Algorithm as a stored procedure

```sql
SELECT artist, trackname, hitcount, offset
FROM metadata,
    (SELECT TOP 10 song, COUNT(*) AS hitcount, (signatures.offset - #temp.offset) AS offset
     FROM signatures, #temp
     WHERE signatures.signature = #temp.signature
     GROUP BY (signatures.offset - #temp.offset), song
     ORDER BY hitcount DESC)
AS matches
WHERE metadata.songid = matches.song
ORDER BY hitcount DESC
```

The process of loading the signatures of the audio library into the database posed some interesting challenges. We expected that the length of time for analysis would be proportional to the size of the audio data (i.e. the length of the song). However, we noticed that for certain songs analysis took much longer - disproportionate to the length. We discovered that this was being caused by intensive paging; when there was not enough physical memory the large intermediary matrices were having to be paged to disk. We overcame this problem by blocking the data we send to the analysis module - so, rather than sending the whole song at once for signature creation, we send at most a fixed number of bytes. Figure 7 shows page file and CPU usage both before and after blocking is employed. Notice how the CPU is left relatively idle while the paging activity takes place.

5.4 Interfaces

5.4.1 Windows Client

The windows client (see Figure 8) is written in C# and makes use of the .NET framework. The user clicks a button to start recording audio and then clicks to stop recording. The recorded audio sample is then sent to the web service. The most likely match is displayed, and the user is given the option of seeing the
Figure 7: Effect of blocking on pagefile and CPU usage during analysis.
further results. The client ensures connection to the web service upon loading. Multithreading ensures responsiveness during the web service call. Basic statistics, such as the amount of data sent and the time taken to search, are displayed in the status bar. A simple visualization is also used during the recording process, primarily for aesthetic reasons but also to reassure the user that his or her microphone is functioning.

5.4.2 IVR Application

Voxeo is a company that provides an IVR (Interactive Voice Response) platform that allows developers to create applications that interface with the standard telephone network. Written in VoiceXML, our application (see Listing 2) simply prompts the user to hold his or her phone to the music and then, when the call is finished, sends the recorded audio to our web service along with the Caller ID. Voxeo provide us with a telephone number to access our application.
Listing 2: VoiceXML application

```xml
<?xml version='1.0'?>
<vxml version="2.0">
  <var name="CallerID" expr="session.callerid"/>
  <catch event="connection.disconnect.hangup">
    <submit expr="'voxeo.aspx?'+CallerID" method="post"
      namelist="CallersMessage" enctype="multipart/form-data"/>
    <exit/>
  </catch>
  <form id="main">
    <record name="CallersMessage" beep="true" maxtime="30s"
      finalsilence="2500ms" type="audio/wav">
      <prompt>
        Welcome. Please hold your phone to the speakers.
      </prompt>
    </record>
  </form>
</vxml>
```

6 Evaluation

6.1 Test Methodology

We designed an application which plays a random snippet for each of the 500 songs and re-records it through a microphone. The test program contacts the matching server which returns the best match for each sample, the test application then reports whether or not it was correct and compiles the statistics. See Figure 9.

6.2 Statistics

Our goal in the project was to achieve 80% accuracy in matching 20 second samples against a database of 500 songs. We exceeded this target and can successfully match 5 second samples with 89% accuracy and 10 second samples with 99% accuracy in a moderately noisy environment. The following stats were recorded using our testing framework:

<table>
<thead>
<tr>
<th>Sample length (seconds)</th>
<th>10</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit rate</td>
<td>99%</td>
<td>89%</td>
</tr>
<tr>
<td>Match Time (seconds)</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Due to time constraints each of these tests was performed only once, for one sample at a random offset for each of the 500 songs.
Figure 9: The testing application
6.3 Channels

During signature creation, the left and right channels of stereo input are combined. This means that the channels must be balanced in the same way in a sample as they were in the audio used to create the signatures in our database. Obviously, this is not a problem for tracks recorded in mono or for tracks with few differences between the two channels. However, if the two channels differ significantly then it is very important that the sample contains a fair mix of both channels. There are a number of situations in which this might not be the case in practice - for instance, a user may hold their microphone close to one speaker to drown out background noise - which would cause a problem for our system.

6.4 Transitions

Samples containing significant transitions in the music perform better than more monotone samples. Because the matching algorithm matches offsets, a uniform, monotone beat is difficult to place accurately. In fact, the sample is often matched at multiple offsets in the song with a difference of the period of the beat. In samples containing transitions, it is unlikely that many songs will have similar patterns on both sides of the transition, greatly improving the accuracy of the system.

6.5 Pure Tones

Pure tones (those close to sine waves) such as those created by a synthesiser produce a single clear peak in the spectrogram. This energy peak is easy to distinguish. However, complex waveforms such as the human voice, comprised of a combination of many different frequencies, despite being more susceptible to noise, have a more unique pattern so fewer are needed to return a successful match. See Figure 10 for an example of these two types of sound pattern.

6.6 Failed Matches

There are two types of sample that frequently fail to match correctly. First, samples which contain only repeated sounds with no distinctive transitions, such as repetitive guitar strumming or electronic beats, produce few useful signatures. Second, when there are very similar segments in two different songs then a sample may match both equally well and the system will be unable to distinguish between them. In both cases the problem is greatly reduced when longer samples are available. In the latter case, it is clear that the longer the sample the less likely there will be two possible matches in two different songs. In the former case, the
generation of a greater number of signatures and greater likelihood of transitions within the sample greatly improve the accuracy of matching.

7 Conclusions & Future Work

The algorithms described in this paper seem to be highly robust and efficient, which were our major goals.

Future extensions would obviously include increasing the size of the database. The use of only 500 songs in this implementation was not a limitation of the algorithm but simply because these were the songs that were easily available to us and because we did not want to waste too much time building databases.

We would also run many more large scale tests with varying parameters to optimise the analysis engine. We may be able to further improve the frequency ranges chosen for the band pass filter and the filter bank and we may be able to find an even better windowing method. There are also some possibilities for improving the search without reducing its efficiency. For example, we could try matching against multiple subsets of bits in the signature rather than checking for an exact match.

We believe that the algorithm would scale well to large databases due to its low complexity. There are also many possibilities for parallelisation, for example the search algorithm would fit well into the MapReduce idiom [11].

For this project we focussed mainly on producing a system that functions in a similar manner to Shazam. With suitable resources and some further optimization we believe our system could out-perform Shazam, which requires samples of a
minimum of 20 seconds in length. Our system could also easily be converted to check for intellectual property violations.

References


