Written by
Ciumac Sergiu
ciumac.sergiu@gmail.com

Sound fingerprinting

June 4

2011

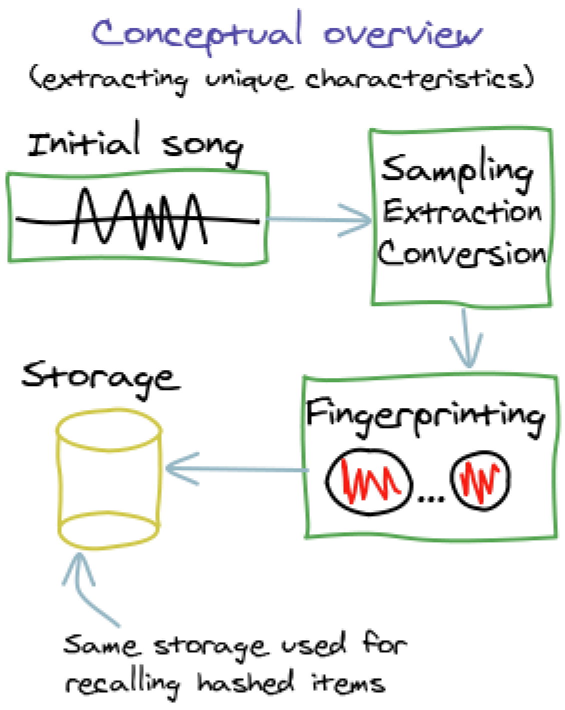
Digital signal processing, data mining, image retrieval.

Duplicates detector via sound fingerprinting.

Contents

Introduction

As a software engineer, I was always interested in how a computer can be taught to behave intelligently, at least on some simple tasks that we (homo-sapiens) can easily solve within frames of seconds. One of them applies to audio recognition, which in recent years has been analyzed thoroughly. For this reason, in this article you will be introduced to one of complex tasks which arise in the field of computer science: the efficient comparison and recognition of analog signals in digital format. As an example, consider an audio signal &psi$_1$, which you would like to compare to another &psi$_2$ in order to see if they both are coming from the same song or audio object. Any person could cope with this assignment with no problem at all, but computers unfortunately are not that intuitively "smart". The difficulty lies in the fact, that each of the signals might have different digitized formats, thus making their binary signatures totally different (resulting in an obsolete byte-by-byte comparison). The dissimilarity may also result, because of the combination of different internal characteristics of the same audio format (bit rate, sampling rate, number of channels (mono, stereo, etc.)). Even if you proceed with the conversion of the files to some predefined specifications (e.g. `44100 Hz`, stereo, `WAVE PCM` format), you still might bump into the problem of having different binary representation because of the possible time misalignment, noise, distortion, or "loudness levels" for the same song ("loudness" technically defined as amplitude level). Starting from 1970's the recording studios tried to push the loudness limit as far as they could, as the motivation was very simple: when comparing two recordings with different levels, listeners are likely to prefer the louder one (see loudness war). So, the aim of this article is to show an efficient algorithm of signal processing which will allow one to have a competent system of sound fingerprinting and signal recognition. The algorithm which is going to be described further, was developed by *Google* researchers, and published in *"Content fingerprinting using wavelets"*. Here, I'll try to come with some explanations of the article's algorithm, and also speak about how it can be implemented using C# programming language. Additionally, I'll try to cover topics of digital signal processing that are used in the algorithm, thus you'll be able to get a clearer image of the entire system. As a proof of concept, I'll show you how to develop a simple `WPF MVVM` application (if you don't know what is `MVVM` don't go Googling yet, as it is going to be explained further), those solely purpose will be detecting duplicate music files on your local drive, analyzing them by audio content.
In simple terms, if you want to compare audio files by their perceptual equality, you should create the so called "fingerprints" (similar to human fingerprints, which uniquely describe person's identity), and see if sets of these objects, gathered from different audio, match or not. Logically, similar audio objects should generate similar fingerprints, whereas different files should emanate unlike signatures. One of the requirements for these fingerprints is that they should act as "forgiving hashes", in order to cope with format differences, noise, "loudness", etc. The simplified concept of audio fingerprinting can be visualized below.



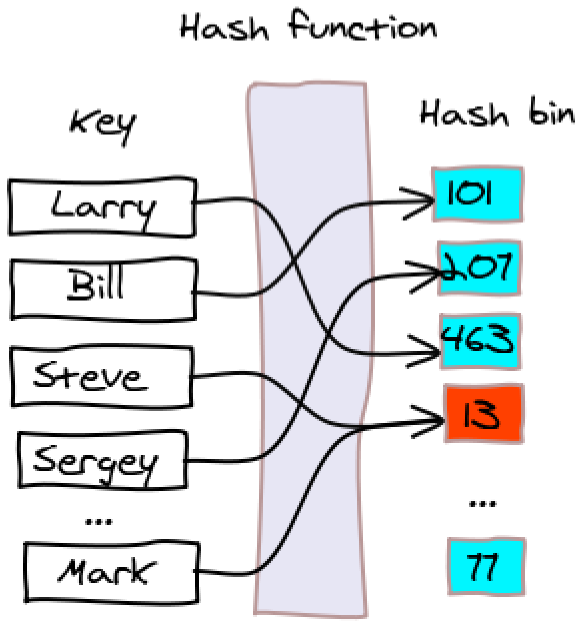
Basically, audio fingerprinting algorithms provide the ability to link short, unlabeled snippets of audio content to corresponding data about that content (it's all about finding unique characteristics of a song that can be later used to recall an unknown sample, or find a duplicate in the database of already processed songs). These types of algorithms can be used in a variety of different applications (please note, until *Content Fingerprinting Using Wavelets* was published, a number of other approaches have been developed):

- Automatic population the metadata of a song (*MusicBrainz*)
- Identifying a currently playing song (*Shazam*)
- Monitoring radio broadcasts (*Mediaguide*, *Nielson BSD*)
- Solving Peer-2-Peer copyright issues
- Managing sound effect libraries

With all the advantages that the sound fingerprinting system has, there are several challenges that it has to address. One of them is a huge database to search (imagine Youtube's database of video content that is monitored for audio copyright issues). Normally, each song will generate a big amount of fingerprints (in the described algorithm the granularity of each of them is going to be `1.48 sec`, so we'll have about 150 objects for 1 song). In conclusion, there will be a need in using an efficient search algorithm that works well, when the solution is scaled. The simplest approach is performing a simple comparison between the query point and each object in the database. But, because this method is too time-consuming, the *k-nearest neighbor* solution (`Min-Hash + Locality Sensitive Hashing`) will be explored.

Fingerprint as a hash value

Briefly, an audio fingerprint can be seen as a compressed summary of the corresponding audio object. Mathematically speaking a fingermark function F maps the audio object X consisting from a large number of bits to a fingerprint of only a limited number of bits. So, the entire mechanism of extraction of a fingerprint can be seen as a hash-function H, which maps an object X to a hash (a.k.a. message digest). The main advantage why hash functions are so widely used in the field of computer science is that they allow comparing 2 large objects X, Y, by just comparing their respective hash values H(X), H(Y). The equality between later pairs implies the equality between X, Y, with a very low error probability (even though collision cannot be eliminated entirely). Next, you can see a conceptual diagram of hash function's functionality.

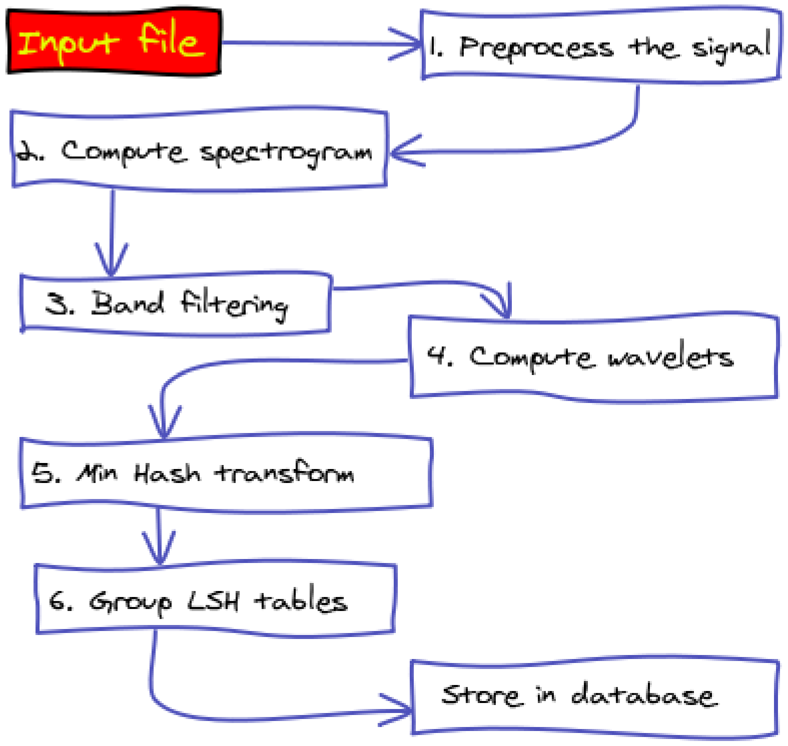


Here in the figure, string values act as keys, mapping into hash bins (integers), which are smaller by design and much easier to compare. Nevertheless, there is always a small probability that completely different keys will occupy the same bin (*Steve* and *Mark* collision).

So, why we need to develop an entirely new system based on fingerprints, and not just use cryptographic hash functions? The answer is pretty simple. Although the perpetual difference of an audio object compressed to MP3 format and the same one compressed to WAVE, is very small, the binary representation of those are totally different, meaning that H(MP3(X)) and H(WAVE(X)) will result in completely dissimilar message digests (hash bins). Even worst, cryptographic hash functions are usually very sensitive: a single bit of difference in the original object, results in a completely different hash value (thus a small degradation in the key will lead to totally different hash digest). Of course that doesn't please us. We would like to have a system that will not take into account any low level details (binary signatures), and will analyze only the audio signal as any human does (will act as a "forgiving hash", that won't pay much attention to irrelevant differences).

General schema (fingerprint creation)

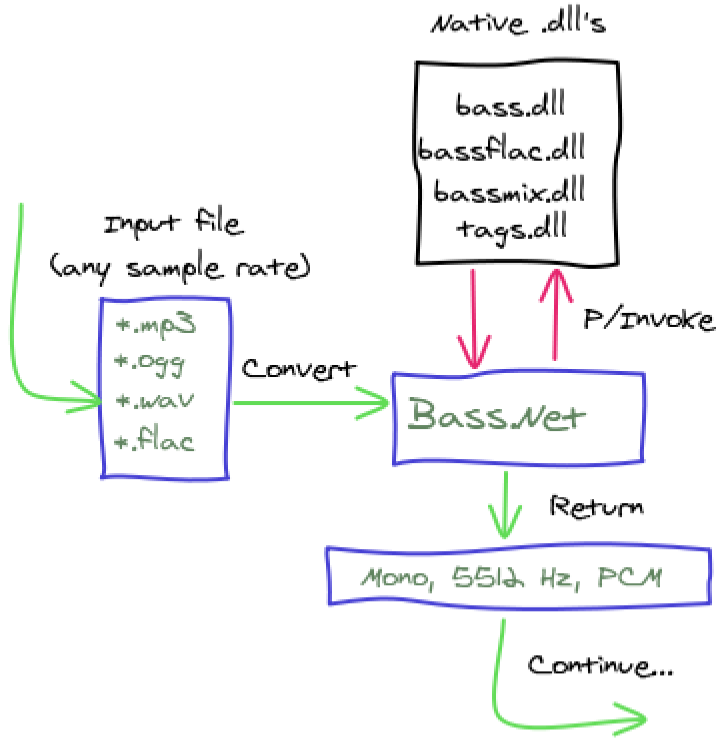
The framework which is going to be built (upon which the entire system will reside), will consist from several different conceptual parts. In next figure, you can visualize the activity diagram, which abstractly describes the logical flow of the fingerprint creation (this flow matches theoretical abstractions described in *Content Fingerprinting Using Wavelets*, the paper upon which the algorithm is build). Following, I will describe in deeper details each activity involved and what component is responsible for it.



Broadly speaking, the algorithm can be logically decomposed into 2 main parts: fingerprint creation (extracting unique perceptual features from the song) and fingerprint lookup (querying the database). Because lots of the components are involved in both activities, they will be described together.

Preprocessing the input signal

Merely, all audio files that any person has on his computer are encoded in some format. Therefore, the first step (1) in the algorithm is decoding the input file to `PCM` (Pulse Code Modulation format). The `PCM` format can be considered the raw digital format of an analog signal, in which the magnitude of the signal is sampled regularly at uniform intervals, with each sample being quantized to the nearest value within a range of digital steps. After decoding, goes the Mono conversion and sampling rate reduction. Particularly, the sampling rate, sample rate, or sampling frequency defines the number of samples per second (or per other unit) taken from a continuous signal to make a discrete signal. For time-domain signals, the unit for sampling rate is 1/s. The inverse of the sampling frequency is the sampling period or sampling interval, which is the time between samples. As the analyzed frequency band in the algorithm lies within the range of 318 Hz and 2000 Hz, downsampling the input to 5512 Hz is considered a safe and, at the same time a required operation. While reducing the sample rate (normally from 44100 Hz) we'll get rid of information which is not relevant from the human perceptual point of view, and will focus on important features of the signal. The preprocessing is done using Bass.Net library (in my opinion the best one for working with audio files).



When retrieving sample data, 32-bit floating-point samples will be gathered ranging from -1.0 to +1.0 (float/Single datatype - not clipped). You can retrieve sample data via the BASS_ChannelGetData method. Note, that any compressed file (e.g. MP3, OGG, WMA, FLAC) will be first decoded, such that you always will receive PCM samples (here lies the great benefit of using Bass library - it will decode the input file to PCM, with a great support of a lot of modern file formats). Also, please note that a "sample" consists of all bytes needed to represent the discrete signal for all channels (stereo/mono) (if it is retrieved as float, 4 bytes per sample will be used). Following are shown some conversion examples, so you'll be able play with the math easily:
Example: 16-bit, stereo, 44.1kHz

- 16-bit = 2 byte per channel (left/right) = 4 bytes per sample

- the sample rate is 44100 samples per second
- one "sample" is 1/44100th of a second long (or 1/44th of a millisecond)
- 44100 samples represent 1 second which is worth 176400 bytes

To convert between the different values, the following formulas can be used:

- milliseconds = samples * 1000 / samplerate
- samples = ms * samplerate / 1000
- samplerate = samples * 1000 / ms
- bytes = samples * bits * channels / 8
- samples = bytes * 8 / bits / channels

In terms of theoretical constructs, the `Nyquist-Shannon` sampling theorem states that perfect reconstruction of a signal is possible when the sample rate is greater than twice the maximum frequency of the signal being sampled, or equivalently, that the `Nyquist` frequency (half the sample rate) exceeds the bandwidth of the signal being sampled. If lower sampling rates are used, the original signal's information may not be completely recoverable from the sampled signal. That's why standard `Audio-CDs` use a sample rate of `44100Hz`, as the human ear can typically only recognize frequencies up to `20000Hz`.
Next, you can see the source code for decoding, mono conversion, and sample rate reduction using `Bass.Net` library.

```
/// <summary>
/// Read mono from file
/// </summary>
/// <param name="filename">Name of the file</param>
/// <param name="samplerate">Output sample rate</param>
/// <param name="milliseconds">Milliseconds to read</param>
/// <param name="startmillisecond">Start millisecond</param>
/// <returns>Array of samples</returns>
/// <remarks>
/// Seeking capabilities of Bass where not used because of the possible
/// timing errors on different formats.
/// </remarks>
public float[] ReadMonoFromFile(string filename, int samplerate, int milliseconds, int
startmillisecond)
{

    int totalmilliseconds = milliseconds <= 0 ? Int32.MaxValue : milliseconds +
startmillisecond;
    float[] data = null;
    //create streams for re-sampling
    int stream = Bass.BASS_StreamCreateFile(filename, 0, 0,
BASSFlag.BASS_STREAM_DECODE | BASSFlag.BASS_SAMPLE_MONO | BASSFlag.BASS_SAMPLE_FLOAT);
//Decode the stream
    if (stream == 0)
        throw new Exception(Bass.BASS_ErrorGetCode().ToString());
    int mixerStream = BassMix.BASS_Mixer_StreamCreate(samplerate, 1,
BASSFlag.BASS_STREAM_DECODE | BASSFlag.BASS_SAMPLE_MONO | BASSFlag.BASS_SAMPLE_FLOAT);
    if (mixerStream == 0)
        throw new Exception(Bass.BASS_ErrorGetCode().ToString());
    if (BassMix.BASS_Mixer_StreamAddChannel(mixerStream, stream,
BASSFlag.BASS_MIXER_FILTER))
    {
        int bufferSize = samplerate * 10 * 4; /*read 10 seconds at each iteration*/
        float[] buffer = new float[bufferSize];
        List<float[]> chunks = new List<float[]>();
        int size = 0;
        while ((float) (size)/samplerate*1000 < totalmilliseconds)
        {
            //get re-sampled/mono data
            int bytesRead = Bass.BASS_ChannelGetData(mixerStream, buffer, bufferSize);
            if (bytesRead == 0)
                break;
            float[] chunk = new float[bytesRead/4]; //each float contains 4 bytes
            Array.Copy(buffer, chunk, bytesRead/4);
            chunks.Add(chunk);
            size += bytesRead/4; //size of the data
        }

        if ((float) (size)/samplerate*1000 < (milliseconds + startmillisecond))
            return null; /*not enough samples to return the requested data*/
        int start = (int) ((float) startmillisecond*samplerate/1000);
        int end = (milliseconds <= 0) ? size : (int) ((float) (startmillisecond +
milliseconds)*samplerate/1000);
        data = new float[size];
        int index = 0;
        /*Concatenate*/
        foreach (float[] chunk in chunks)
        {
            Array.Copy(chunk, 0, data, index, chunk.Length);
            index += chunk.Length;
        }
        /*Select specific part of the song*/
        if (start != 0 || end != size)
        {
            float[] temp = new float[end - start];
            Array.Copy(data, start, temp, 0, end - start);
            data = temp;
        }
    }
    else
        throw new Exception(Bass.BASS_ErrorGetCode().ToString());
    return data;
}
```

If you'd like to develop mono/sample rate conversions by your own, you can use the following snippets of code in order to get the same results as with the `Bass` library (almost the same :)). The only thing to mention is that `Bass` allows you to set the filter for anti-aliasing (very useful for downsampling procedures), while the following code snippets do not take any action against it.
Here goes the method for sample rate reduction from `44100Hz` to `5512Hz` (using a standard low pass filter):

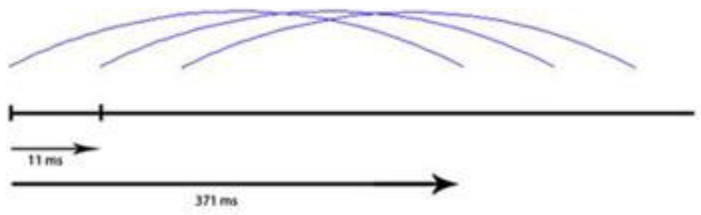
```


/// <summary>
/// Low pass filter for downsampling 44100Hz to 5512Hz
/// </summary>
readonly double [] _lpFilter44 =  {
        -6.4796e-04, -1.4440e-03, -2.7023e-03, -4.4407e-03, -6.1915e-03, -6.9592e-03
        , -5.3707e-03,  2.3907e-18,  1.0207e-02,  2.5522e-02,  4.5170e-02,  6.7289e-02
        ,  8.9180e-02,  1.0778e-01,  1.2027e-01,  1.2467e-01,  1.2027e-01,  1.0778e-01
        ,  8.9180e-02,  6.7289e-02,  4.5170e-02,  2.5522e-02,  1.0207e-02,  2.3907e-18
        , -5.3707e-03, -6.9592e-03, -6.1915e-03, -4.4407e-03, -2.7023e-03, -1.4440e-03
        , -6.4796e-04

     };
/// <summary>
/// Convolve a sample with a filter at a given point
/// </summary>
/// <param name="samples">Samples to be convolved</param>
/// <param name="startIndex">Starting index of a sample</param>
/// <param name="filter">Filter used in convolving</param>
/// <returns>Convolved sample</returns>
static float ConvolvePoint(float[] samples, int startIndex, double[] filter)
{
    float sum = 0;
    int len = filter.Length;
    for (int i = 0; i < len; i++)
        sum += (float) (samples[i + startIndex]*filter[i]);
    return sum;
}

/// <summary>
/// Downsample a signal from 44100Hz to 5512Hz
/// </summary>
/// <param name="samples">Samples to be downsampled</param>
/// <returns>Downsampled signal</returns>
public float[] Downsample(float[] samples)
{
    const int multiplier = 8;
    int nsamples = samples.Length;
    int newnsamples = nsamples/multiplier - 3;
    float[] newsamples = new float[newnsamples];
    for (int i = 0; i < newnsamples; i++)
        newsamples[i] = ConvolvePoint(samples, multiplier*i, _lpFilter44);
    return newsamples;
}
```

Next, if you wish to build your own method to convert a signal to single channel (mono), this method can help you out. Please note that at the input a PCM stereo signal is supplied as byte array (opposite to float decoding, used previously)

```
/// <summary>
/// Mono conversion of a stereo signal decoded in bytes
/// </summary>
/// <param name="data">PCM stereo signal</param>
/// <returns>PCM mono signal</returns>
public byte[] StereoToMono(byte[] data)
{
    byte[] mono = new byte[data.Length / 2];
    for (int i = 0; i < mono.Length / 2; i++)
    {
        int left = (data[i * 4] << 8) | (data[i * 4 + 1] & 0xff);
        int right = (data[i * 4 + 2] << 8) | (data[i * 4 + 3] & 0xff);
        int avg = (left + right) / 2;
        short m = (short)avg; /*Mono is an average between 2 channels (stereo)*/
        mono[i * 2] = (byte)((short)(m >> 8));
        mono[i * 2 + 1] = (byte)(m & 0xff);
    }
    return mono;
}
```

Spectrogram creation

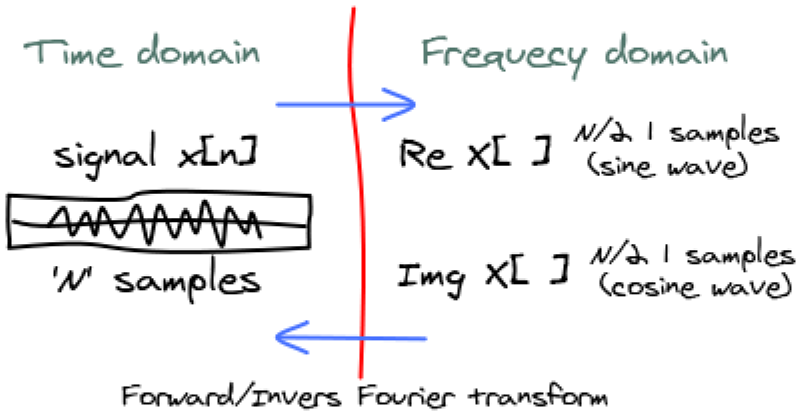
After preprocessing the signal to `5512 Hz PCM`, the next (2) step in the audio fingerprinting algorithm is building the spectrogram of the audio input. In digital signal processing, a spectrogram is an image that shows how the spectral density of a signal varies in time. Converting the signal to spectrogram involves several steps. First of all the signal should be sliced into overlapping frames. Each frame should be then passed through `Fast Fourier` transform, in order to get the spectral density varying in time domain. The parameters used in this transformation steps will be equal to those that have been found to work well in other audio fingerprinting studies (specifically in *A Highly Robust Audio Fingerprinting System*): audio frames that are `371 ms` long (`2048` samples), taken every `11.6 ms` (`64` samples), thus having an overlap of `31/32`. The mechanism of slicing and framing the input signal can be visualized as follows:



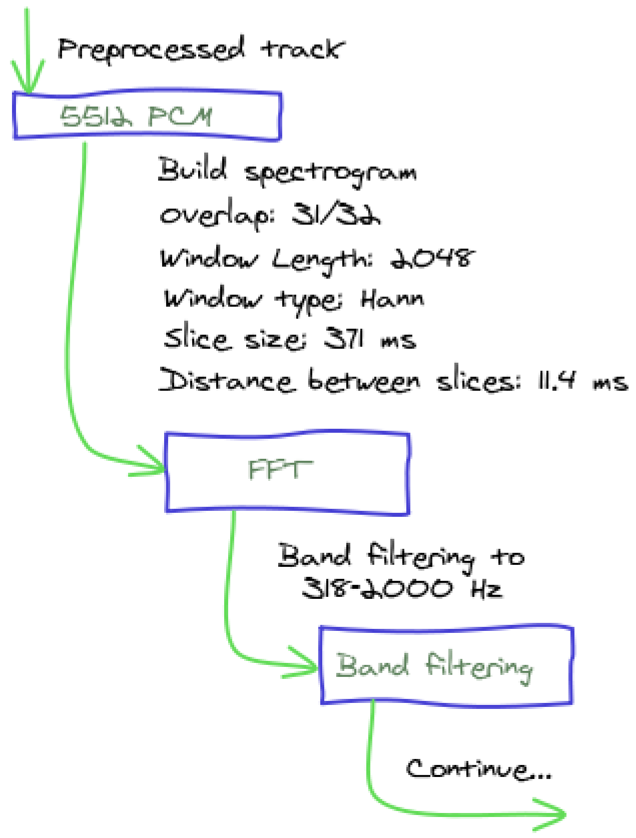
One important consequence of the slice length/spacing combination of parameters (`371 ms` slices each `11.6 ms`) is that the spectrogram varies slowly in time, providing matching robustness to position uncertainty (in time). Building the spectrogram of the image involves the most expensive (from CPU point of view) operation in the entire algorithm - `Fast Fourier Transform` (O(n*log(n))). I should point that, before applying this transformation, each slice of the input signal (`371 ms`) should be weighted by `Hann Window` function. Weighting the signal by a window function is done, in order to narrow the domain of a function `F` to a specified range (usually `[-1, 1]`). For instance, a function that is constant inside the interval and zero elsewhere is called a rectangular window, which describes the shape of its graphical representation. When another function or a signal (data) is multiplied by a window function, the product is also zero-valued outside the interval: all that is left is the part where they overlap; the "view through the window". The method for weighting is very simple, as it uses the `Hann` function formula: `w(n) = 0.5*(1-cos(2*Π*n/(N-1)))`

```
// <summary>
/// Gets the corresponding window data (according to specified length)
/// </summary>
/// <param name="length">Length of the window</param>
/// <returns>Window function</returns>
public double [] GetWindow(int length)
{
    double[] array = new double[length];
    //Hanning window
    for (int i = 0; i < length; i++)
        array[i] = 0.5 * (1 - Math.Cos(2 * Math.PI * i / (length - 1)));
    return array;
}
```

As I've mentioned earlier, in order to get the spectrogram of the input signal, the so called `Fast Fourier Transform` is applied on each of the sliced frames. Fourier analysis is a family of mathematical techniques, all based on decomposing signals into sinusoids. As shown in the following image the discrete Fourier transform changes an `N` point input signal into two point output signals. The input signal contains the signal being decomposed, while the two output signals contain the amplitudes of the component sine and cosine waves. The input signal is said to be in the time domain, while the output is in frequency domain. This is because the most common type of signal entering the `FFT` is composed of samples taken at regular intervals of time (in our case `371 ms` slices).



The frequency domain contains exactly the same information as the time domain, just in a different form. If you know one domain, you can calculate the other. Given the time domain signal, the process of calculating the frequency domain is called decomposition, analysis, the forward `FFT`, or simply, the `FFT`. If you know the frequency domain, calculation of the time domain is called synthesis, or the inverse `FFT`. Both synthesis and analysis can be represented in equation form and computer algorithms. I've used the Exocortex DSP library in order to apply this transformation.



Band filtering

After the FFT transform is accomplished on each slice, the output spectrogram, is cut such that 318 Hz - 2000 Hz domain is obtained (researchers decided that this domain is enough for a relevant representation of perceptual content of a file). Generally, this domain can be considered to be one of the most relevant frequency space for Human Auditory System. While the range of frequencies that any individual can hear is largely related to the environmental factors, the generally accepted standard range of audible frequencies is 20 to 20,000 Hz. Frequencies below 20 Hz can usually be felt rather than heard, assuming the amplitude of the vibration is high enough. Frequencies above 20,000 Hz can sometimes be sensed by young people, but high frequencies are the first to be affected by hearing loss due to age and/or prolonged exposure to very loud noises. In the next table, you can visualize the frequency domains and their main descriptions. As it can be seen the most relevant audio content lies within 512-8192Hz frequency range, as it defines the normal speech.

Frequency ranges:

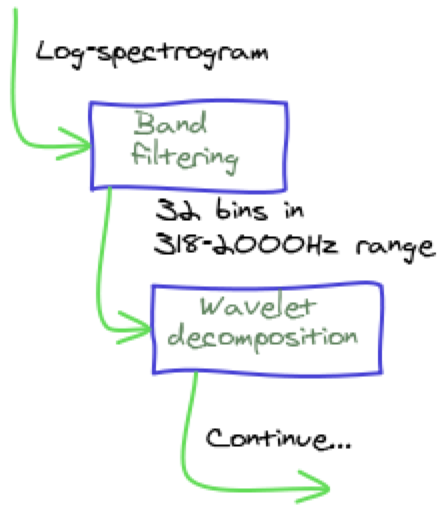
Frequency (Hz)	Octave	Description
16 - 32	1	The human threshold of feeling, and the lowest pedal notes of a pipe organ.
32 - 512	2 - 5	Rhythm frequencies, where the lower and upper bass notes lie.
512 - 2048	6 - 7	Defines human speech intelligibility, gives a horn-like or tinny quality to sound.
2048 - 8192	8 - 9	Gives presence to speech, where labial and fricative sounds lie.
8192 - 16384	10	Brilliance, the sounds of bells and the ringing of cymbals. In speech, the sound of the letter "S" (8000-11000 Hz)

So, getting back to the algorithm, in order to minimize the output dimensionality (each frame after `FFT` transform is a vector of size `[0 - 1025]`), the specified range of `318-2000Hz` should be encoded into `32` logarithmically spaced bins (so the `2048` samples in time domain are reduced to `1025` bins in frequency domain which are then summed in `32` items in logarithmic scale). Thus, the algorithm reduces each time frame by a factor of `64x`. All in all, the use of logarithmical spacing in frequency was selected based for simplicity, since the detailed band-edge locations are unlikely to have a strong effect under such coarse sampling (only `32` samples across frequency). Next, is shown the code for spectrogram creation that uses all the features discussed in previous steps.

```
/// <summary>
/// Create log-spectrogram
/// </summary>
/// <param name="proxy">Proxy used in generating the spectrogram</param>
/// <param name="filename">Filename to be processed</param>
/// <param name="milliseconds">Milliseconds to be analyzed</param>
/// <param name="startmilliseconds">Starting point</param>
/// <returns>Logarithmically spaced bins within the power spectrum</returns>
public float[][] CreateLogSpectrogram(IAudio proxy, string filename, int milliseconds,
int startmilliseconds)
{
    //read 5512 Hz, Mono, PCM, with a specific proxy
    float[] samples = proxy.ReadMonoFromFile(filename, SampleRate, milliseconds,
startmilliseconds);
    NormalizeInPlace(samples);
    int overlap = Overlap;
    int wdftSize = WdftSize;
    int width = (samples.Length - wdftSize)/overlap; /*width of the image*/
    float[][] frames = new float[width][];
    float[] complexSignal = new float[2*wdftSize];   /*even - Re, odd - Img*/
    for (int i = 0; i < width; i++)
    {
        //take 371 ms each 11.6 ms (2048 samples each 64 samples)
        for (int j = 0; j < wdftSize /*2048*/; j++)
        {
            complexSignal[2*j] = (float) (_windowArray[j]*samples[i*overlap + j]);
/*Weight by Hann Window*/
            complexSignal[2*j + 1] = 0;
        }
        //FFT transform for gathering the spectrum
        Fourier.FFT(complexSignal, wdftSize, FourierDirection.Forward);
        frames[i] = ExtractLogBins(complexSignal);
    }
    return frames;
}
```

Wavelet decomposition

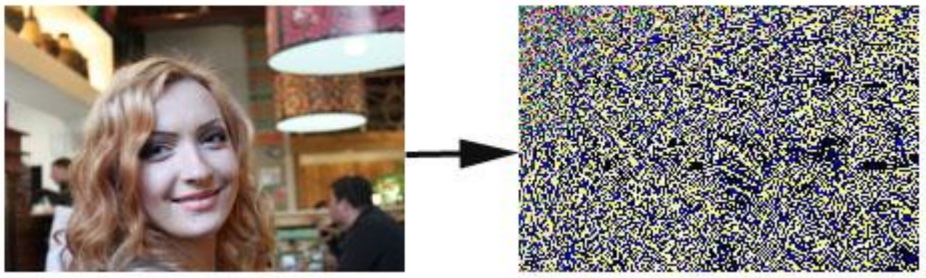
Once the logarithmic spectrogram was "drawn" (having 32 bins spread over 318-2000Hz range, for each 11.6 ms), the next step in the algorithm (4) is dividing the entire image into slices (spectral sub-images 128x32) that correspond to 1.48 sec granularity. In terms of programming after you build the log-spectrogram, you got its entire image encoded into [N][32] double-dimensional float array, where N equals to the "length of the signal in milliseconds divided by 11.6 ms", and 32 represents the number of the frequency bands. Once you get the divided spectral sub-images, they will be further reduced by applying Haar wavelet decomposition on each of them. The use of wavelets in audio-retrieval task is done due to their successful use in image retrieval *(Fast Multiresolution Image Querying)*. In other words, a wavelet is a wave-like oscillation with amplitude that starts at zero, increases, and then decreases back to zero. It can typically be visualized as a "brief oscillation" like the one seen on a seismograph or heart monitor. Wavelets are crafted purposefully to have specific properties that make them useful for signal processing. They can be combined by using a "shift, multiply and sum" technique called convolution, with portions of an unknown signal to extract information from this signal. The fundamental idea behind wavelets is to analyze according to scale. So, turning back to our algorithm, for each of the spectral images, a wavelet signature is computed. Instead of using the entire set of wavelets, we only keep the ones that most characterize the song (Top 200 is considered a good choice), thus making the signature resistant to noise and other degradations. One of the interesting things found in the previous studies of image processing, was that there is no need in keeping the magnitude of top wavelets; instead we can simply keep the sign of it (+/-).



This particular framework will use the following sign encoding: positive number - `01`, negative number - `10`, zero - `00`. The 2 most important features of this bit vector are that it is sparse and resistant to small degradations that might appear in the signal. Sparsity makes it amendable to further dimensionality reduction through the use of `Min Hash`. It's worth mentioning that at this stage (after Top-wavelet extraction), the length of the bit vector we get for 1 fingerprint is `8192` (where only `200` elements are equal to `1`, others equal to `0`). Following is presented the source code for Haar-Wavelet decomposition:

```
/// <summary>
///   Haar wavelet decomposition algorithm
/// </summary>
public class HaarWavelet : IWaveletDecomposition
{
    #region IWaveletDecomposition Members

    /// <summary>
    ///   Apply Haar Wavelet decomposition on the frames
    /// </summary>
    /// <param name = "frames">Frames to be decomposed</param>
    public void DecomposeImageInPlace(float[][] frames)
    {
        DecomposeImage(frames);
    }

    #endregion

    /// <summary>
    ///   Decomposition taken from
    ///   Wavelets for Computer Graphics: A Primer Part by Eric J. Stollnitz Tony D.
DeRose David H. Salesin
    /// </summary>
    /// <param name = "array">Array to be decomposed</param>
    private static void DecomposeArray(float[] array)
    {
        int h = array.Length;
        for (int i = 0; i < h; i++)           /*doesn't introduce any change in the
final fingerprint image*/
            array[i] /= (float)Math.Sqrt(h);  /*because works as a linear normalizer*/
        float[] temp = new float[h];

        while (h > 1)
        {
            h /= 2;
            for (int i = 0; i < h; i++)
            {
                temp[i] = (float)((array[2 * i] + array[2 * i + 1]) / Math.Sqrt(2));
                temp[h + i] = (float)((array[2 * i] - array[2 * i + 1]) /
Math.Sqrt(2));
            }
            for (int i = 0; i < 2*h; i++)
            {
                array[i] = temp[i];
            }
        }
    }

    /// <summary>
    ///   The standard 2-dimensional Haar wavelet decomposition involves one-
dimensional decomposition of each row
    ///   followed by a one-dimensional decomposition of each column of the result.
    /// </summary>
    /// <param name = "image">Image to be decomposed</param>
    private static void DecomposeImage(float[][] image)
    {
        int rows = image.GetLength(0); /*128*/
        int cols = image[0].Length; /*32*/

        for (int row = 0; row < rows /*128*/; row++) /*Decomposition of each row*/
            DecomposeArray(image[row]);

        for (int col = 0; col < cols /*32*/; col++) /*Decomposition of each column*/
        {
            float[] column = new float[rows]; /*Length of each column is equal to
number of rows*/
            for (int row = 0; row < rows; row++)
                column[row] = image[row][col]; /*Copying Column vector*/
            DecomposeArray(column);
            for (int row = 0; row < rows; row++)
                image[row][col] = column[row];
        }



    }
}
```

Next, you can see an image which was transformed by using `Haar` decomposition (before and after). In the case of audio fingerprinting, by applying this kind of transformation on the spectrogram of an audio input, we'll see easily identifiable patterns for consecutive fingerprints. An example will be given shortly in the future.



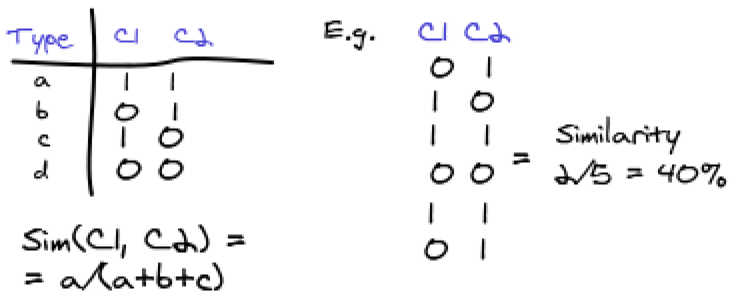
In the above paragraphs I've described the algorithm for fingerprint creation, following you can visualize the method that performs the actual task. As you can see, first the log-spectrogram is created, and then the image is sliced into `1.48 sec` spectral images (from which the actual fingerprints will be built). I should emphasize that the "distance" between 2 consecutive fingerprints can be established though the use of `IStride` interface. Essentially, there is no best answer of how many seconds you should skip between 2 items, as it is mostly detected empirically through testing. In *Content Fingerprinting Using Wavelets* a static `928 ms` stride was used in database creation, and a random `0-46 ms` stride was used in querying (random stride was used in order to minimize the coarse effect of unlucky time alignment).

```

/// <summary>
/// Create fingerprints according to the Google's researchers algorithm
/// </summary>
/// <param name="proxy">Proxy used in reading from file</param>
/// <param name="filename">Filename to be analyzed</param>
/// <param name="stride">Stride between 2 consecutive fingerprints</param>
/// <param name="milliseconds">Milliseconds to analyze</param>
/// <param name="startmilliseconds">Starting point of analysis</param>
/// <returns>Fingerprint signatures</returns>
public List<bool[]> CreateFingerprints(IAudio proxy, string filename, IStride stride,
int milliseconds, int startmilliseconds)
{
    float[][] spectrum = CreateLogSpectrogram(proxy, filename, milliseconds,
startmilliseconds);
    int fingerprintLength = FingerprintLength;
    int overlap = Overlap;
    int logbins = LogBins;
    int start = stride.GetFirstStride() / overlap;
    List<bool[]> fingerprints = new List<bool[]>();
    int width = spectrum.GetLength(0);
    while (start + fingerprintLength < width)
    {
        float[][] frames = new float[fingerprintLength][];
        for (int i = 0; i < fingerprintLength; i++)
        {
            frames[i] = new float[logbins];
            Array.Copy(spectrum[start + i], frames[i], logbins);
        }
        start += fingerprintLength + stride.GetStride() / overlap;
        WaveletDecomposition.DecomposeImageInPlace(frames); /*Compute wavelets*/
        bool[] image = ExtractTopWavelets(frames);
        fingerprints.Add(image);
    }
    return fingerprints;
}
```

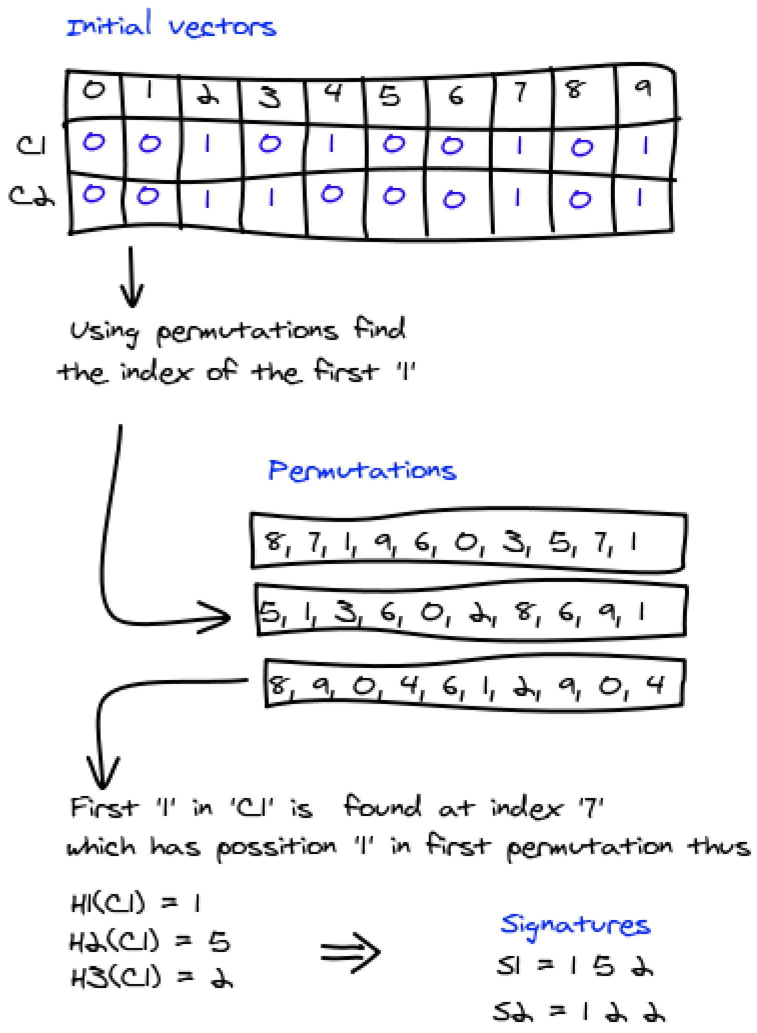
This method returns the actual fingerprints that will be further reduced in dimensionality through the use of `Min Hash + LSH` technique.

Min-Hashing the fingerprint

At this stage of the processing we got fingerprints that are `8192` bits log, which we would like to further reduce in their length, by creating a compact representation of each item. Therefore, in the next step (5) of the algorithm we explore the use of `Min-Hash` to compute sub-fingerprints for these sparse bit vectors. Its usage has proven to be effective in efficient search of similar sets. It works as follows: think of a column as a set of rows in which the column has a 1 (consider `C1` and `C2` as 2 fingerprints). Then the similarity of two columns `C1` and `C2` is `Sim(C1, C2) = (C1∩C2)/(C1∪C2)`



This similarity index is called the Jaccard similarity: It is a number between 0 and 1; it is 0 when the two sets are disjoint, 1 when they are equal, and strictly between 0 and 1 otherwise. Before exploring how Min-Hash works, I'll point that a very useful characteristic that it has: the probability that MinHash(C1) = MinHash(C2) equals exactly to Jaccard similarity. This will allow us to use the algorithm by efficiently hashing similar items (with a similarity index more than 60-70%) into the same bins (exactly what we are interested in - "forgiving hashing", method that won't pay much attention to small differences between 2 keys). The Min-Hash technique works with binary vectors as follows: Select a random, but known, reordering of all the vector positions. Then, for each vector permutation measure in which position the first '1' occurs. Because the system will work with fingerprints of 8192 bit size, the permutation vector will contain random indexes starting from 0 to 8192. For a given permutation, the hash values agree if the first position with a 1 is the same in both bit vectors, and they disagree if the first such position is a row where one but not both, vectors contained a 1. Note that this is exactly what is required; it measures the similarity of the sparse bit vectors based on matching "on" positions. We can repeat the above procedure multiple times, each time with a new position-permutation. If we repeat the process p times, with p different permutations, we get p largely independent projections of the bit vector (e.g. p=100 permutations).



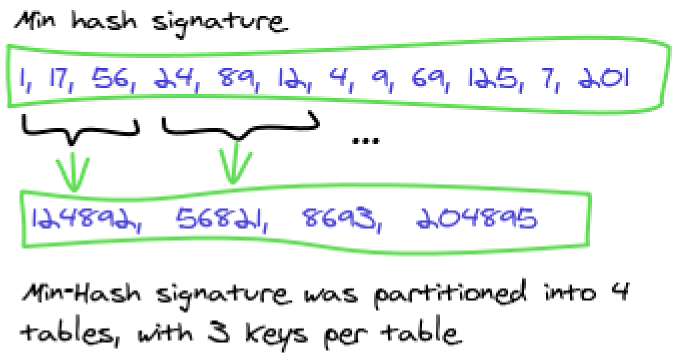
Following one can see the method which min hashes a given fingerprint.

```
/// <summary>
/// Compute Min-Hash signature of a given fingerprint
/// </summary>
/// <param name="fingerprint">Fingerprint to be hashed</param>
/// <returns>Min hashes of the fingerprint (size equal to number of
permutations)</returns>
public int[] ComputeMinHashSignature(bool[] fingerprint)
{
    bool[] signature = fingerprint;
    int[] minHash = new int[_permutations.Count]; /*100*/
    for (int i = 0; i < _permutations.Count /*100*/; i++)
    {
        minHash[i] = 255; /*The probability of occurrence of 1 after position 255 is
very insignificant*/
        int len = _permutations[i].Length;
        for (int j = 0; j < len /*256*/; j++)
        {
            if (signature[_permutations[i][j]])
            {
                minHash[i] = j; /*Looking for first occurrence of '1'*/
                break;
            }
        }
    }
    return minHash; /*Array of 100 elements with bit turned ON if permutation captured
successfully a TRUE bit*/
}
```

If you wonder how the permutations were generated, I can disappoint you that the procedure is not straightforward at all. The method was based on *Permutation Grouping: intelligent hash functions for audio and image retrieval* paper. It explores how we can generate random permutations that are more or less identically distributed. Rather than simply selecting high entropy permutations to place together, a more principled method is to use the Mutual Information (MI) between permutations to guide which permutations are grouped. Mutual information is a measure of how much knowing the value of one variable reduces the uncertainty in another variable. To determine whether there is sufficient mutual information variance to use this as a valid signal, for 100 permutations, where examined the mutual information between all pairs (100*99/2 samples). In order to create groups of low mutual information permutations to put together into hashing chunks, the greedy selection procedure was used, that is loosely based on the algorithm used in *Approximating Discrete Probability Distributions with Dependence Trees*. The utility that was used in generating the permutations alongside with a lot of other useful features related to this research can be found here (project hosting used in writing the application and associated tools). I won't describe the entire permutation grouping algorithm as it is out of the scope of this article. If you are interested, you can research this topic using the material found in this section.

Solving the k-nearest neighbor problem using Locality Sensitive Hashing

Generally stated, up until this moment we've been able to reduce the dimensionality of the initial vector through the use of different algorithms, as `Fourier` transform, `Haar` wavelet decomposition, selecting `Top-T` wavelets which contain the most relevant information, and `Min-Hash` algorithm. From the initial spectral sub-image (`128x32` floats) we've gathered a vector with `100` 8-bit integers which represent the fingerprint itself. Unfortunately, even with this high degree of dimensionality reduction, it is hard to create an efficient system that will have to search within a vector of length `100`. That's why, in the next step (6) of this article we explore the use of `Locality Sensitive Hashing`, which is an important class of algorithm for finding nearest neighbors. It proved to be efficient in the number of comparisons that are required and provides noise-robustness properties. Particularly, the nearest neighbor problem is defined as follows: given a collection of n points, build a data structure which, given any query point, reports the data point that is closest to the query. This problem is of major importance in several areas; some examples are data compression, databases and data mining, information retrieval, image and video databases, machine learning, pattern recognition, statistics and data analysis. Typically, the features of each object of interest (document, image, etc.) are represented as a point in R^d and the distance metric is used to measure the similarity of objects. The basic problem then is to perform indexing or similarity searching for query objects. The number of features (i.e., the dimensionality) ranges anywhere from tens to millions. For example, one can represent a 1000x1000 image as a vector in a 1 000 000-dimensional space, one dimension per pixel. In order to successfully apply the concept of locality sensitive hashing, we need to hash the points using several hash-functions to ensure that for each function the probability of collision is much higher for objects that are close to each other than for those that are far apart (Min-Hash functions adjust to this requirement). Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. The nearest neighbor problem is an example of an optimization problem: the goal is to find a point which minimizes a certain objective function (in this case, the distance to the query point). To simplify the notation, we say that a point p is an R-near neighbor of a point q if the distance between p and q is at most R. The easiest way to imagine it is as follows: a classical SQL query on a table will return the exact matches that you supply in the where clause. As opposite, the k-nearest neighbor algorithm will return you not just the exact matches, but also those that are similar to the query parameter. As an example, consider that you would like to select all the books from the database that contain the word "port" in their text. Instead of returning the exact matches, the algorithm will return you not only the books that contain the initial word in it but also those that contain the words: sup**port**, re**port**, air**port**, de**port**, **port**folio, etc. Why this might be useful? There are certain applications that can use this feature for solving some specific problems (finding similar text, websites, images, audio items, etc.). In our case it is not just a way of finding similar audio objects, but also a way of increasing the search speed (if the application is scaled, searching through a database that contains vectors of 100 8 bit values, is not a feasible solution).



In audio fingerprinting algorithm discussed in this article, we use L hash functions from Min-Hash algorithm discussed in the previous section (where each hash function represents a concatenated value of B min hashes). For instance, 25 hash functions (represented as 4 concatenated min-hashes), and 25 corresponding hash tables, are enough to efficiently partition initial data points into corresponding space address. Candidate neighbors can be efficiently retrieved by partitioning the probe feature vector in the same way the initial audio object was divided, and collecting the entries in the corresponding hash bins. The final list of potential neighbors can be created by vote counting, with each hash casting votes for the entries of its indexed bin, and retaining the candidates that receive some minimum number of votes, v. If v = 1, this takes the union of the candidate lists, else if v=L, this takes the intersection. Following one can see the source code for LSH table composition out of Min-Hash values:

```
/// <summary>
///   Compute LSH hash buckets which will be inserted into hash tables.
///   Each fingerprint will have a candidate in each of the hash tables.
/// </summary>
/// <param name = "minHashes">Min Hashes gathered from every fingerprint [N =
100]</param>
/// <param name = "numberOfHashTables">Number of hash tables [L = 25]</param>
/// <param name = "numberOfMinHashesPerKey">Number of min hashes per key [N =
4]</param>
/// <returns>Collection of Pairs with Key = Hash table index, Value = Hash
bucket</returns>
public Dictionary<int, long> GroupMinHashToLSHBuckets(int[] minHashes, int
numberOfHashTables, int numberOfMinHashesPerKey)
{
    Dictionary<int, long> result = new Dictionary<int, long>();
    const int maxNumber = 8; /*Int64 biggest value for MinHash*/
    if (numberOfMinHashesPerKey > maxNumber)
        throw new ArgumentException("numberOfMinHashesPerKey cannot be bigger than
8");
    for (int i = 0; i < numberOfHashTables /*hash functions*/; i++)
    {
        byte[] array = new byte[maxNumber];
        for (int j = 0; j < numberOfMinHashesPerKey /*r min hash signatures*/; j++)
        {
            array[j] = (byte)minHashes[i * numberOfMinHashesPerKey + j];
        }
        long hashbucket = BitConverter.ToInt64(array, 0); //actual value of the
signature
        result.Add(i, hashbucket);
    }
    return result;
}
```

Retrieval

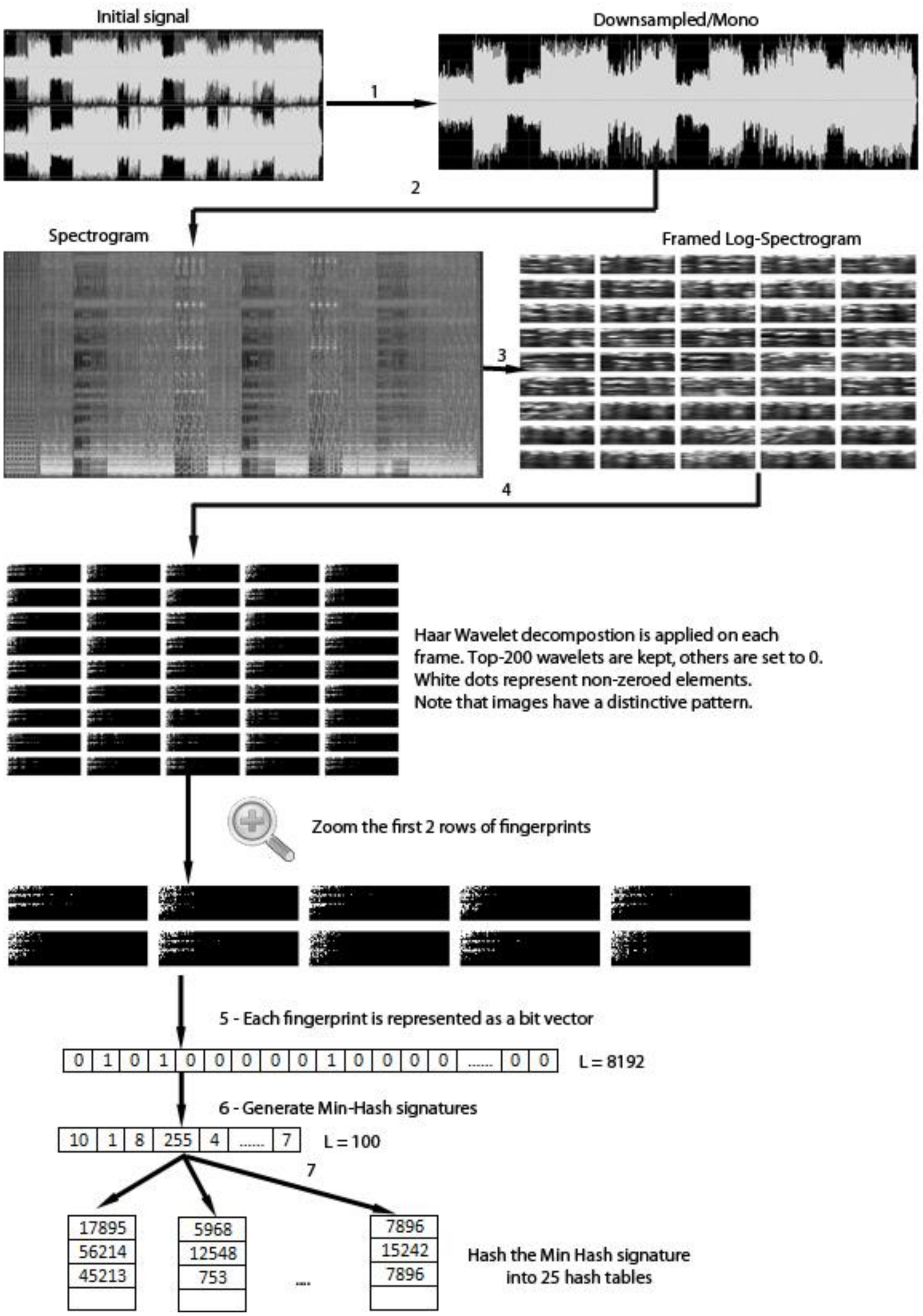
After the min-hashed fingerprint was partitioned into `LSH` tables, the process of extracting signatures from audio files can be considered completed. Now, how can we find similar songs or retrieve information about an unknown audio by having only a small snippet of it? The overall retrieval process (detection of the query song) is similar to creational one, with an exception that not the entire song is fingerprinted at this time. You may select as many seconds to fingerprint as you want, making a tradeoff between accuracy and processing time. The steps are repeated till the fingerprint is partitioned into hash tables. Once done, the query is sent to the storage to see how many hash tables contain corresponding hash values. If more than `v` votes are acquired during the query (more than `v` tables have returned the same fingerprint), the fingerprint is considered a potential match (in *Content fingerprinting using wavelets* a threshold of 2-5 votes was used). All matches are selected, and analyzed in order to find the best candidate (a simple `Hamming` distance calculation between the query and potential candidates can be done in order to see what item is the most relevant one). In case of developing a duplicates detector algorithm, the principle of extraction is a bit different. Number of threshold votes is increased to `8` (`8/25` tables should return the same result), and at least `5%` of total query fingerprints should vote for the same song. Once this criterion is met, the song is considered a duplicate. Overall, all these parameters are defined empirically, so you might find even better configuration that will work more accurate as the one described here. As an example, you may lower the threshold votes, in order to obtain results that will contain also the remixes of the songs matched as duplicates (it's always a trade-off between false-positives and the objective function). Following is shown the code for the extraction schema:

```
/// <summary>
/// Get tracks that correspond to a specific hash signature and pass the threshold
value
/// </summary>
/// <param name="hashSignature">Hash signature of the track</param>
/// <param name="hashTableThreshold">Number of threshold tables</param>
/// <returns>Possible candidates</returns>
public Dictionary<Track, int> GetTracks(int [] hashSignature, int hashTableThreshold)
{
    Dictionary<Track, int> result = new Dictionary<Track, int>();
    for (int i = 0; i < _numberOfHashTables; i++)
    {
        HashSet<Track> voted = new HashSet<Track>();
        for (int j = 0; j < _numberOfHashTables; j++)
        {             if (_hashTables[i].ContainsKey(hashSignature[j]))
            {
                var tracks = _hashTables[i][hashSignature[j]];
                foreach (var track in tracks)
                {
                    if (voted.Contains(track)) continue;
                    if (!result.ContainsKey(track))
                        result[track] = 1;
                    else
                        result[track]++;
                    voted.Add(track);
                }
            }
        }
    }
    Dictionary<Track, int> filteredResult = result.Where(item => item.Value >=
hashTableThreshold)
        .ToDictionary(item => item.Key, item => item.Value);
    return filteredResult;
}
```

Summary

There are many steps required in building the fingerprints, so its not uncommon to lose the connections between all of them. In order to simplify the explanation, next you can see a generalized image will help you in visualizing them all together. It is a full example of processing a `44100 Hz, Stereo, .mp3` file (Prodigy - No Good). Specifically, the activity flow is as follows:

1. It can be seen from the image that the input file has 2 channels (stereo). In the first step, it is downsampled to `5512 Hz, Mono`. Pay attention that the shape of the input signal doesn't change.
2. Once the signal is downsampled, it's corresponding spectrogram is built.
3. The spectrogram is logarithmized and divided into spectral images, which will be further representing the fingerprints themselves.
4. Each spectral-frame is transformed using `Haar` wavelet decomposition, and `Top-200` wavelets are kept in the original image. Note a distinctive pattern between consecutive fingerprints. Having similar wavelet signatures across time, will help us in identifying the song even if there is a small time misalignment between creational and query fingerprints.
5. Each fingerprint is encoded as a bit vector (length `8192`).
6. `Min-Hash` algorithm is used in order to generate corresponding signatures.
7. Finally, using `Locality Sensitive Hashing`, `Min-Hash` signature for each fingerprint is spread in `25` hash tables, which will be later used in lookup.


Initial signal
Downsampled/Mono
1
2
Spectrogram
Framed Log-Spectrogram
3
4
Haar Wavelet decompostion is applied on each frame. Top-200 wavelets are kept, others are set to 0. White dots represent non-zeroed elements. Note that images have a distinctive pattern.
Zoom the first 2 rows of fingerprints
5 - Each fingerprint is represented as a bit vector
0 1 0 1 0 0 0 0 0 1 0 0 0 0 ...... 0 0
L = 8192
6 - Generate Min-Hash signatures
10 1 8 255 4 7
L = 100
7
17895
56214
45213
5968
12548
753
....
7896
15242
7896
Hash the Min Hash signature into 25 hash tables

MVVM

The application, which will use the described algorithm, is going to detect duplicate files on your local machine. The general task is very simple. First, it will process all the audio files from selected folders, and then will try to detect which of them match to the same signature, thus being duplicates one to each other. It will be built using `WPF` framework, and specifically the `MVVM` pattern, which becomes more popular with the expansion of last. The `Model-View-ViewModel (MVVM)` is an architectural pattern used in software engineering that originated from *Microsoft* as a specialization of the *Presentation Model* design pattern introduced by *Martin Fowler*. `MVVM` was designed to make use of specific functions in `WPF` to better facilitate the separation of `View` layer development from the rest of the pattern by removing virtually all "code behind" from the `View` layer. Elements of the `MVVM` pattern include:

- `Model`: as in the classic `MVC` pattern, the model refers to either (a) an object model that represents the real state content (an object-oriented approach), or (b) the data access layer that represents that content (a data-centric approach).
- `View`: is responsible for defining the structure, layout, and appearance of what the user sees on the screen. Ideally, the view is defined purely with `XAML`, with no code-behind other than the standard call to the `InitializeComponent` method in the constructor.
- `ViewModel`: it acts as an intermediary between the `View` and the `Model`, and is responsible for handling the view logic. The `ViewModel` provides data in a form that the view can easily use. The `ViewModel` retrieves data from the `Model` and then makes that data available to the `View`, and may reformat the data in some way that makes it simpler for the `View` to handle. The `ViewModel` also provides implementations of `Commands` that a user of the application initiates in the `View`. For example, when a user clicks on a button in the `UI` that can trigger a command in the `ViewModel`. The `ViewModel` may also be responsible for defining logical state changes that affects some aspect of the display in the `View` such as an indication that some operation is pending.

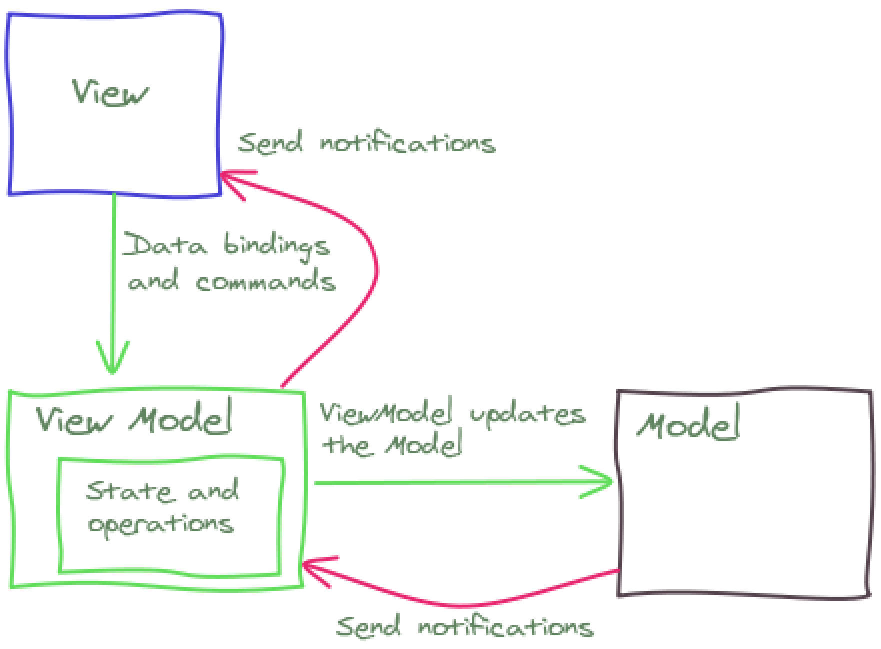
In addition to understanding the responsibilities of the three components, it's also important to understand how the components interact with each other. At the highest level, the `View` knows about the `ViewModel`, and the `ViewModel` knows about the `Model`, but the `Model` is unaware of the `ViewModel`, and the `ViewModel` is unaware of the `View`. `MVVM` relies on the data binding capabilities in `WPF` to manage the link between the `View` and `ViewModel`. The important features of an `MVVM` application that make use of these data binding capabilities are:

- The `View` can display richly formatted data to the user by binding to properties of the `ViewModel` instance. If the `View` subscribes to events in the `ViewModel`, then the `ViewModel` can also notify the `View` of any changes to its property values using these events.

- The `View` can initiate operations by using `Commands` to invoke methods in the `ViewModel`.
- If the `View` subscribes to events defined in the `ViewModel`, then the `ViewModel` can notify the `View` of any validation errors using these events.

The `ViewModel` typically interacts with the `Model` by invoking methods in the model classes. The `Model` may also need to be able to report errors back to the `ViewModel` by using a standard set of events that the `ViewModel` subscribes to (remember that the `Model` is unaware of the `ViewModel`). In some scenarios, the `Model` may need to be able to report changes in the underlying data back to the `ViewModel` and again the `Model` should do this using events. The clear separation of responsibilities in the `MVVM` pattern results in a number of benefits.

- During the development process, developers and designers can work more independently and concurrently on their components. The designers can concentrate on the `View`, and if they are using *Expression Blend* they can easily generate sample data to work with, while the developers can work on the `ViewModel` and `Model` components.
- The developers can create unit tests for the `ViewModel` and the `Model` without using the `View`. The unit tests for the `ViewModel` can exercise exactly the same functionality as used by the `View`.
- It is easy to redesign the user interface (`UI`) of the application without touching the code because the `View` is implemented entirely in `XAML`. A new version of the `View` should work with the existing `ViewModel`.
- If there is an existing implementation of the `Model` that encapsulates existing business logic, it may be difficult or risky to change. In this scenario, the `ViewModel` acts as an adapter for the model classes and enables you to avoid making any major changes to the `Model` code.



Following is shown a simple example of how a ViewModel can be bound to a View, thus allowing developers to completely separate the underlying logic of the application from the UI.

```
 <Border CornerRadius="5">
        <ItemsControl ItemsSource="{Binding Workspaces}"
Margin="10,10,10,10"></ItemsControl>
 </Border>
```

In the underlying code you can bind chosen ViewModel to MainWindow by just adding it to the Workspaces collection of the MainWindowViewModel.

```
/// <summary>
/// Main window view model
/// </summary>
public class MainWindowViewModel : ViewModelBase
{
    ObservableCollection<ViewModelBase> _workspaces;

    /// <summary>
    /// Parameterless constructor
    /// </summary>
    public MainWindowViewModel()
    {
        /*Adding PathList view model to workspaces collection*/
        ViewModelBase pathList = new PathListViewModel();
        Workspaces.Add(pathList);
    }

    /// <summary>
    /// Workspaces
    /// </summary>
    public ObservableCollection<ViewModelBase> Workspaces
    {
        get { return _workspaces ?? (_workspaces = new
ObservableCollection<ViewModelBase>()); }
        private set { _workspaces = value; }
    }
}
```

The interaction between the UI and the underlying ViewModel is done using Commands, which ultimately replace the use of event-handlers and publish-subscribe model. This has a number of great benefits. One of them is that you can perform automated testing on your UI logic, with no need in hooking and other nasty workarounds. In the MVVM pattern, the responsibility for implementing the action lies with the ViewModel, and you should avoid placing code in the code-behind classes. Therefore, you should connect the control to a method in the ViewModel using a binding. Thus, on your View you define the command that will be invoked once an action is performed. As an example consider the following button that is bound to StartCommand command.

```
 <Button Command="{Binding StartCommand}"  Content="Start" />
```

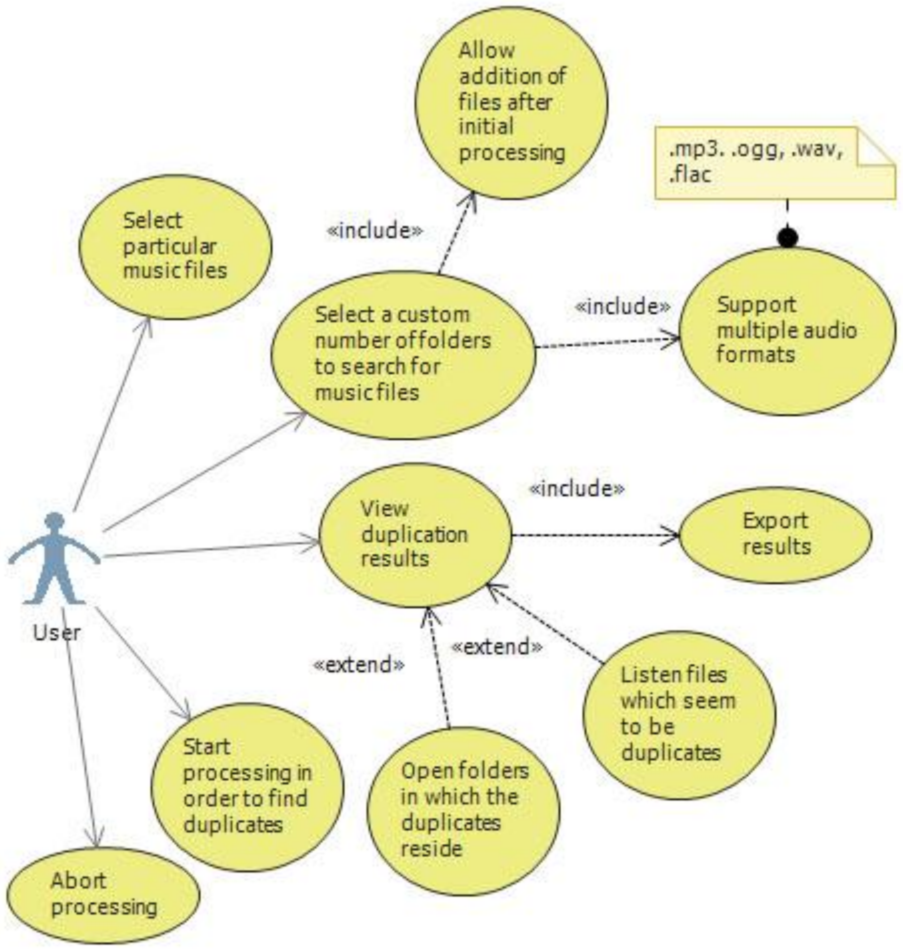
Once bound you should have in the corresponding ViewModel class, the definition of StartCommand command.

```
/// <summary>
/// Start processing the songs
/// </summary>
public ICommand StartCommand
{
    get
    {
        return _startCommand ?? (_startCommand = new RelayCommand(
            StartProcessing, CanStartProcessing));
    }
}
```

StartProcessing and CanStartProcessing are just delegates to methods that will be invoked once a user clicks that button (CanStartProcessing is invoked at each UI refresh in order to see whether the user is allowed to perform the operation). This is very useful, once you develop a method which is dependent upon some earlier event (such as processing a song is dependent upon its selection, thus the start command will be disabled until a user selects a song). And yet, once you choose developing an application using MVVM pattern, you should take into consideration several issues that you'll most probably dive in. One of them is instantiating new windows and showing them to the user. This act as simple as it was in classical event based programming should now be carried on with much more interesting approach. Because, you cannot directly instantiate window objects in ViewModel (remember the pattern, ViewModel has no clue about the View), you should perform this using Dependecy Injection and Service Contract pattern. Why using these two? It is simple, once you implement this approach you'll be able to inject any behavior you want once the request of showing a dialog is sent. As an example, imagine you have an automated test that checks the logic within your ViewModel. Consider your ViewModel allows a user to save a file on a disk. On production environment you would like to instantiate the SaveFileDialog and allow the user to select the folder in which to store the file. However, in testing environment, you are not allowed to do so (automated testing requires no user interaction, thus no tester is going to select the name and the folder where to store the file). By implementing DependencyInjection, you'll be able to inject mock behavior in testing environment with no instantiation of pop-up windows.
Coming back to the algorithm; as this article's purpose is to introduce you to audio fingerprinting concepts, I won't dive into details of MVVM programming. If you are interested, I suggest you to read about MVVM as it brings a lot of new opportunities for developers. You can check the following article for greater details.

Duplicates detector

In the next use case diagram you can see the tasks that one will be allowed to perform using *Duplicates Detector* application. Their number is limited due to the fact that this is an explanatory article which focuses on the algorithm of audio fingerprinting.


Allow addition of files after initial processing
.mp3. .ogg, .wav, .flac
Select particular music files
«include»
«include»
Select a custom number of folders to search for music files
Support multiple audio formats
«include»
View duplication results
Export results
User
«extend»
«extend»
Listen files which seem to be duplicates
Start processing in order to find duplicates
Open folders in which the duplicates reside
Abort processing

In order to start processing you can select one or more root folders that contain music files. The application will perform an in depth search looking for audio files within root and child folders. Once the search ends, it will display total number of files found. If more than 2 files are discovered, you will be allowed to start searching the duplicates. The fingerprints will be stored in RAM, thus on application exit all the data will be lost.

Conclusion

The aim of this article was to introduce you to the concept of audio recognition and analysis. As this topic becomes more popular these days, (alongside with similar topics related to data mining) you can further explore its theoretical constructs in order to get even better lookup rate. If you take a look on the results that were gathered by Google researchers, you will most probably be amazed by the accuracy of the last. I do fully realize that the results presented in this work are not that accurate as those written in *Content fingerprinting using wavelets*, as there are still things that can be improved. Specifically, I would pay more attention to the following: normalization procedures applied on the amplitude level of the song and wavelets, `Min-Hash` random permutations generator, `Haar` wavelet extraction.

The complete research code alongside with the associated tools can be found here. I would like you to be aware that the project hosting structure is messy (as any research code :)), so it might take you a while to understand it. Also, if you consider improving the algorithm, you would rather want to take a look on the possibility of using neural networks as forgiving hash functions generator (an algorithm that I find very interesting). I encourage you in commenting this article, as I am open to any consistent suggestions related to implementation details. Thank you for your attention.

The works that inspired me

- Content fingerprinting using wavelets
- Permutation grouping: intelligent hash function design for audio and image retrieval
- A highly robust audio fingerprinting system
- Learning "Forgiving" hash functions: algorithms and large scale tests
- Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions
- Similarity search in high dimensions via hashing