Last Lab: Shazam! (Audio Recognition)

Weeks of 9–13 and 16–20 November
EE20 Fall 2015
University of California, Berkeley

In this lab, we're going to build an audio recognition tool. You will be given a "database" of a few songs, but feel free to add more songs if you have any available. (Classical or jazz music is generally harder for computers to recognize than more heavily instrumented genres like pop or rock.) To generate a query, you will randomly select a clip from one of the songs in your database and corrupt it with a bit of noise to model the conditions under which a query would be recorded in the real world—typically on someone's smartphone with a large amount of background noise. Your engine should then be able to figure out which song the noisy clip comes from.

Logistically, this lab is 2 weeks long and has two checkoffs—part A has you extracting the relevant information from raw audio files, and part B has you actually implementing the recognition and search algorithms.

But before we can do that, we need to take a detour into the area of spectrograms, plots that allow you to simultaneously visualize the frequency and time domains.

1 Preliminaries: Spectrograms

Spectrograms are plots that tell us about both the frequency and the time information in a signal. They are typically represented as colored plots with time on the horizontal axis and frequency on the vertical axis. Each column of the spectrogram describes the spectrum of a short segment of the signal. This fixed number of samples is known as a *window*. Figure 1 shows how to find a spectrogram for signal $x(n)$. We're going to step through this process in Matlab.

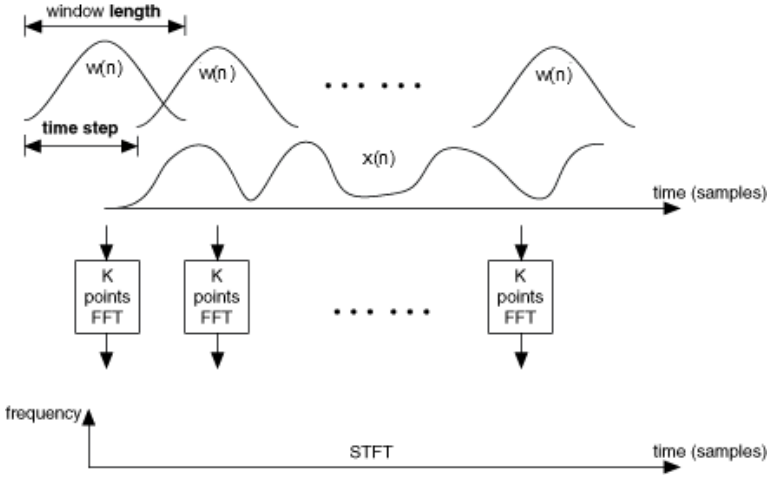

Figure 1: Diagram of spectrogram computation. Spectrograms give us a combination of time and frequency information by calculating spectra for different "chunks" of the signal. Image modified from http://zone.NI.com.

1.1 Windowing

Your code for this part should go in the starter file `spectrograms.m`. For this part, we will use a sampling rate of $f_s = 10$, making the sampling period $T_{samp} = 0.1$ seconds.

1. Create a time vector `t` that lasts 5 seconds and increases by increments of `T_samp`. From this, generate a sine wave of frequency $\frac{2\pi}{5}$ radians/sec. Check that you've actually generated this correctly with the command `plot(t, x)`. This plot represents $x_c(t) = \sin\left(\frac{2\pi}{5}t\right)$. How many signal periods do you expect to see over your 5 second plot?
2. What should the spectrum of x_c look like?
3. Now take the DFT of `x` using `X = fft(x)`.
4. Generate a frequency vector `omega` to plot `X` against. The output of `fft` corresponds to frequencies from 0 to 2π, so omega should start at 0 and go up to one frequency before 2π, in steps of $\frac{2\pi}{N}$ (where N is the length of the input `x`) so that there are exactly N points in `omega`.
   For example, if you calculate the DFT of a signal with length $N = 10$, the frequency steps will be $\frac{2\pi}{10} = \frac{\pi}{5}$, so the frequency vector will be `(0:9) * 2*pi/10`.
5. Plot the magnitude of the DFT and confirm that it looks more or less as you would expect. (The magnitudes should be slightly inaccurate because the irrational π cannot be accurately represented by the computer.) To get the magnitude of `X`, you can use `abs(X)`.
6. Now create a similar signal `xhat` that looks almost like `x`, except it lasts 6 seconds instead of 5 seconds (you'll also want to define a new vector `that`). Plot `xhat` in the same figure as `x` using the command `plot(that, xhat, 'r')`. How many signal periods do you see? Is the number of periods an integer?
7. Again, calculate the DFT of `xhat`. Since `xhat` is a different size than `x`, you will need to generate a new frequency vector `omegahat` in order to plot `Xhat`. Does the range of frequencies change with respect to `w`? Does the spacing of the frequency components change?

For some reason, `Xhat` and `X` do not look the same! This is quite strange. Remember that the signal only had a single frequency present, but this is not accurately reflected in the sampled spectrum. This is a *windowing* effect. Recall that when we take the DFT of a finite-length signal, it's as if we're calculating the DTFS of a periodic signal with one period equal to our finite-length signal. So if our signal is a sinusoid, but its length is not an integer multiple of that sinusoid's period, then periodically extending that signal will result in calculating the spectrum of a signal that is not a sinusoid; this introduces some unexpected frequency components to the spectrum. Figure 2 illustrates this graphically.

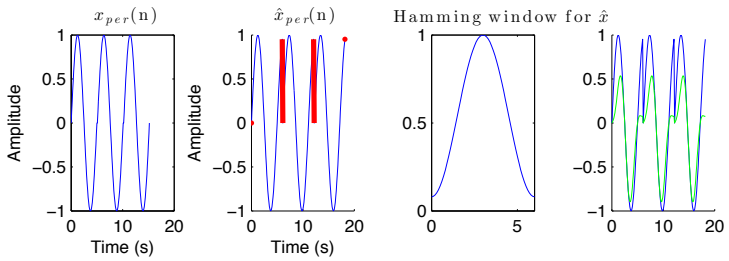

Figure 2: Periodic extension of an incomplete cycle of a sinusoid. (*Left*) periodic replication of a single period of a sinusoid results in a smooth curve. (*Middle left*) Periodic replication of a non-integer number of sinusoid periods results in large, high-frequency discontinuities marked in red. (Middle right) Hamming window example, where the window length is set for $\hat{x}(n)$. (*Right*) Applying the Hamming window before periodic replication (green) results in a much smoother curve as the Hamming window function is tapered at the edges of the window.

This artifact is an obvious problem for spectral analysis. Our continuous-time signal contains only a single frequency, yet depending on how we set the window size, we may mistakenly believe that there are multiple frquencies in the signal!

One solution to this problem is to apply a *windowing function* $w(n)$. Rather than take the DFT of $\hat{x}(n)$ directly, we take the DFT of $\hat{x}(n)w(n)$, where $w(n)$ is some function which smoothly tapers off. For instance, the Hamming window shape is shown in Figure 2. When we periodically replicate $\hat{x}(n)w(n)$, we get a much smoother function, shown in Figure 2 on the right in green.

Let's continue coding:

1. Take `xhat` from the previous step and multiply it pointwise by a Hamming window of the same size. Hamming windows can be generated using the function `w = hamming(L)`, where `L` specifies the number of samples in the Hamming window. You may need to transpose `w` using Matlab's transposition operator, the apostrophe `'`.

2. Plot the DFT of the Hamming-windowed version of `xhat`. How does this look different from the spectrum of `xhat`? In what ways is this better? In what ways is this worse? Keep in mind that the "true" signal is a single-frequency signal.

1.2 Exploring spectrograms

In the last section, our signal contained only one frequency. What happens if there are different frequencies present at different times? Code for this section should go in the cell marked "Exploring Spectrograms" in the same file.

1. In your Matlab command window, type

   ```
   load testAudio.mat
   ```

   This gives you access to all the variables saved in `testAudio.mat`. Listen to `testAudio` (which was generated with a sampling frequency of `fs`) using the command

   ```
   sound(testAudio, fs);
   ```

   What do you expect the spectrum of this audio segment to look like? Plot the spectrum of `testAudio` and make sure that you understand why it looks as it does.

2. Now listen to the audio segment called `scrambledAudio` (which will already be present in the workspace after the `load` command) and look at its spectrum. Is it different? Why or why not?

3. The last step shows that we can use spectra to understand what frequencies are present in a signal, but not to understand *when* in the signal those frequencies occur. Now we're going to start looking at the spectrogram of this signal. To do this we will use the Matlab built-in function `spectrogram`:

   ```
   [S F T P] = spectrogram(x, window, noverlap, NFFT, fs)
   ```

   In this function, `x` is the input signal, `window` is the length of the Hamming window (in samples), `NFFT` is the number of samples to take the DFT of, and `fs` is the sampling frequency. `noverlap` is the number of overlapping samples between two adjacent sliding windows. `fs` only affects the way the spectrogram is plotted, not the spectrogram itself. The relation between `NFFT` and `window` may be confusing—isn't the window size exactly the number of points? This is typically the case, so you can just make all your calls with `NFFT = window`.
   The function has 4 outputs: `S` is the 2D spectrogram matrix, `F` is a vector of the output frequencies for which the spectrogram was computed, `T` is a vector of times. `P` is a matrix of the same size as `S` that describes the power content at each frequency and time (no complex values). So to plot a spectrogram, you can use the following commands:

   ```
   imagesc(T,F,10*log10(P));
   axis tight;
   axis('xy');
   xlabel('Time (sec)');
   ylabel('Freq (kHz)');
   ```

   Try this out with the `testAudio` signal. Use the provided parameters for `window`, `NFFT`, and `noverlap`. Can you tell what the structure of the audio clip is from the spectrogram? Why are there vertical bands when the note changes?

4. Now generate a new figure and plot the spectrogram of the `scrambledAudio` signal, repeating the process from the last step. Is there a difference between the `scrambledAudio` spectrogram and the `testAudio` spectrogram?

1.3 Spectrograms and noise

So spectrograms teach us about both the time and the frequency content of signals. We saw in this section that using only frequency-domain information (i.e. the DFT of the whole audio clip) isn't enough to understand the structure of an audio clip. On the other extreme, why can't we use only time-domain information to understand the structure of the signal? One very important reason is that time-domain representations of audio make it hard to understand what pitches are being played, especially in the presence of noise. In Figure 3, you can see the time-domain representation of a clean sine wave at 440 Hz and the same wave with noise added. However, if we look at the spectrogram, we can see fairly clearly that there is a frequency component at 440 Hz. In real audio signals, there would be many frequencies present at the same time, so the time version would look even messier but we would still be able to see the distinct frequencies in the spectrogram.

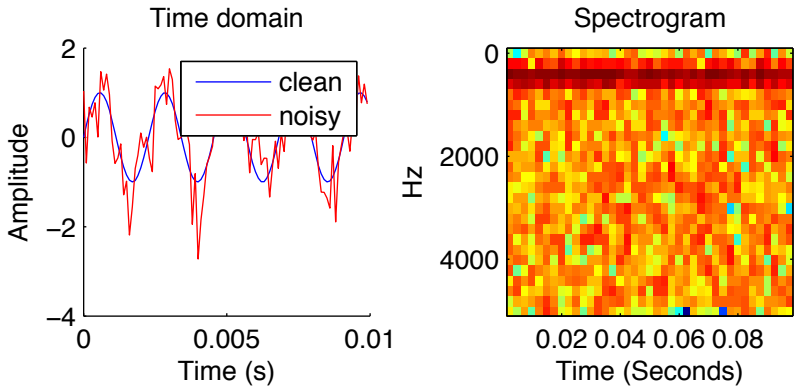

Figure 3: Noise and spectrograms. (Left) A clean 440 Hz sine wave and one with Gaussian noise added. The addition of the noise disrupts the temporal sinusoidal structure. (Right) Spectrogram of the same noisy signal (over a longer time axis) shows that in the frequency domain the 440 Hz sine wave is still clearly visible.

2 The Approach: Audio Recognition

There are many algorithms for efficient audio recognition, like those used by Shazam or SoundHound. Most of these algorithms use spectrograms at their core. Conceptually, the flow of most algorithms is as follows:

1. Beforehand, find the time-frequency features (in our case, simplified spectrograms) of each song in the database.
2. During the query, collect a few seconds of sample audio from the noisy query song.
3. Find the time-frequency features of the sample audio clip.
4. Match the features of the sample clip to a segment of one of the database songs.

This algorithm flow is depicted graphically in Figure 4.

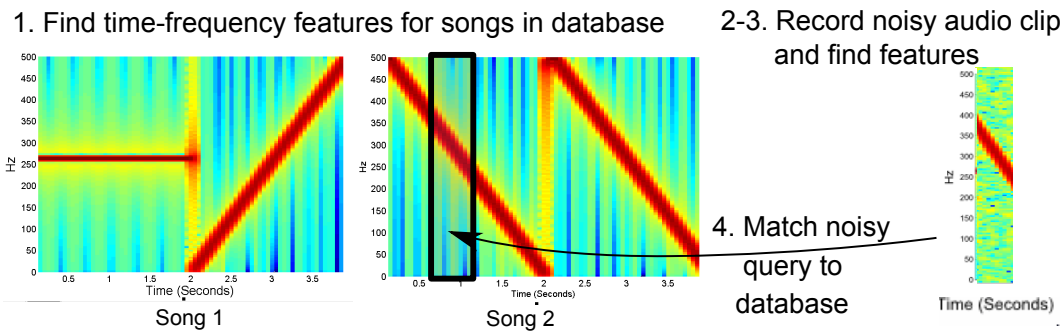

Figure 4: High level diagram of how to do an audio search.

This description almost suggests that we could find the correct audio file by doing an image similarity search—i.e. by interpreting each spectrogram as an image, and treating the query spectrogram as a subsection of an image. This is a valid interpretation, but there are two important observations to make. One is that such a search must be resistant to noise. The other is that the search must be very efficient—tools like Shazam are meant to search through very large databases.

You will implement a simple but effective audio search algorithm by Jaap Haitsma and Ton Kalker.[1] We will step through this algorithm in segments.

2.1 The Starter Files

This lab comes with 7 different Matlab files and some data (besides the files `spectrograms.m` and `testAudio.mat`, which we've already done with). You'll have to implement the following (for which skeletons are given):

extractFeatures.m (the core of your algorithm) takes audio data, computes a spectrogram, and compresses it down to a feature list.

makeDatabase.m takes a list of audio files and extracts the features of all of them into a single structure array that acts as a database of songs that the program can recognize.

generateQuery.m (used to test your algorithm) takes audio data, adds noise, extracts its features, and returns a random segment of its feature list.

shazam.m is the main file of this lab: generates a database, then makes some test queries against it.

The next 3 files are given, and you will probably use them in implementing the algorithm for the lab.

ber.m computes the Bit Error Rate (BER) between two matrices: the percentage of bits that differ between the two.

getFiles.m generates a list of the names of the audio files in a directory; this is used by the starter code to get a list of filenames to feed into `makeDatabase`, but you won't need to use it yourself.

playAudio.m takes an array representing some audio, and plays back the first t seconds of it; you will use this for testing, but not in the final product.

There is also a directory called **data** which contains the audio data for the songs your engine will recognize, and a simple test file named **tone.wav**, which contains a single pure tone.

[1] Haitsma, Jaap, and Ton Kalker. "A highly robust audio fingerprinting system." Proc. ISMIR. Vol. 2. 2002.

3 The Backend: Feature Extraction

We'll start with extracting features for the database of songs. "Feature extraction" is a term commonly used in the field machine learning to describe any preprocessing steps you apply to your raw data before applying an algorithm for learning, classification, regression, etc.

We mentioned that spectrograms can be thought of as a set of time-frequency features. However, the search algorithm needs to be efficient and robust to noise. To make the algorithm efficient, we need to reduce the amount of information stored for each song. To make the algorithm resistant to noise, we need to introduce a feature extraction procedure which is not fooled by noise. Fortunately, these requirements can be satisfied in a single procedure. You will now simultaneously introduce quantization and reduce the dimensionality of features by converting each column of the spectrogram into a feature vector that is 16 bits long. This number 16 is arbitrary—it was simply observed to work well in practice.

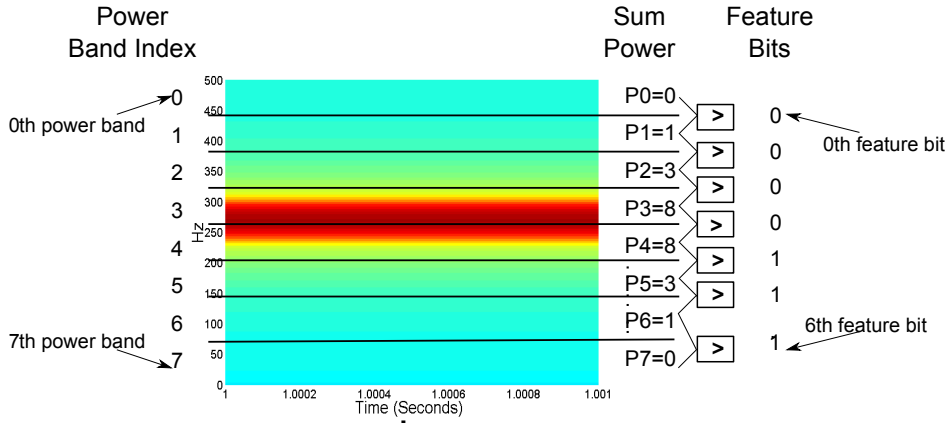

Figure 5: How to extract a 7-bit feature from a single column of a spectrogram.

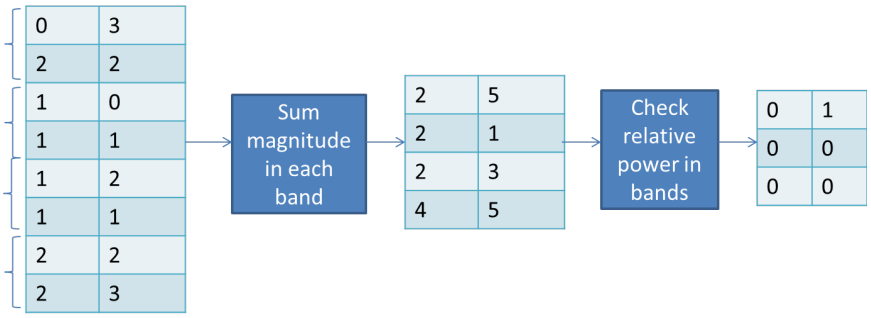

Figure 6: How to extract 3-bit features from a spectrogram with 2 columns.

1. We'll use a function `extractFeatures`, located in the file `extractFeatures.m`., to implement our feature extraction procedure. Read the description of the input arguments to `extractFeatures.m`.

2. Find the spectrogram of the audio signal. You can use an overlap factor of half the window length.

3. The procedure to go from spectrogram to features is depicted in Figures 5 and 6. Briefly, you should group your frequency bands into 17 different groups (the exact sizes of these groups aren't important, provided they're similar and you're consistent about the size of each band). Then collapse each group to a single number per time column by adding up all the power within each group (the power of a function $X(\omega)$ is $|X(\omega)|^2$). Finally, for each column, compare the total power of each frequency band to that of the frequency band above it. If the ith frequency band has more power than the $(i+1)$st

one, assign a 1 to the ith bit, and a 0 otherwise. Here are some facts which you may or may not find useful in implementing this procedure:

(a) The function `sum(x, dim)` returns the sum of a matrix `x` in dimension `dim`. If `dim=1`, it sums the rows of each column, while if `dim=2`, it sums the columns of each row.

(b) For identically-sized matrices `A` and `B`, the Matlab command `A>B` returns a binary matrix in which the `(i,j)` index is 1 if `A(i,j)>B(i,j)` and 0 otherwise

4. After this step you should have a set of 16-bit feature vectors. For each song, there should be as many feature vectors as there are columns in the spectrogram. Verify that this is the case.

5. Currently your function outputs a matrix of 1's and 0's. Before exiting the function, convert the 16-bit vectors (i.e., the feature you extracted from each spectrogram column) to decimal numbers using the function `bi2de`. This function can accept a binary matrix and it converts to a decimal number by interpreting the 1s and 0s as bits of a single number. So now, `extractFeatures` should return a 1D vector of 16-bit numbers.

3.1 Testing the feature extractor

To check that your feature extraction code works properly, start by loading and listening to `tone.wav`. Now extract your features from `tone.wav` by calling your function from the command window:

```
[tone, fs] = audioread('tone.wav');
test_features = extractFeatures(tone(:,1), 1500, 16)
```

Now ensure that the returned features make sense, keeping in mind that windowing may generate some somewhat odd-looking features; comparing very small amounts of power is basically nondeterministic, and (as you know from the preliminary section) a windowed sinusoid has small nonzero values for most of its DFT. You might want to try different window sizes if the features look too noisy and random.

You can visualize the returned features with the command `imagesc(de2bi(test_features,16)')`. This will display an image of the binary `test_features` matrix with yellow "1" bits and blue "0" bits (though colors may vary depending on your computer). What do you think the features should look like for `tone.wav`? Does this look like your spectrogram? You can use Figure 5 to reason about how the features should look.

4 Exercises, Part A

You should have this done by the end of the first week.

1. Demonstrate that your feature extractor is working properly.

2. Show that you are generating the song database correctly.

5 The Frontend: Shazam

5.1 Constructing the database

Open the function `makeDatabase.m`. In this function, we want to extract the features from each song in the database and store them in an appropriate structure. Start out by looping through the files in `files`. Note that `files` is a cell array (essentially an array of pointers to heterogenous data, like a Python list), so to access the ith filename, use `files{i}`. Cell arrays are good for storing a list of differently-sized data. For each filename,

1. Load the audio and sampling rate from the file.

2. Currently, the audio signal has way too much information for us to process it efficiently; the sampling rate is much higher than it needs to be (for the purposes of identification, that is—for general purposes, the sampling rate here is just right, about equal to the Nyquist rate of the input). *Resampling* is the name given to procedures where we change the effective sampling rate of a discrete-time signal. Change the sample rate to 4000 Hz using the `resample` function (`help resample`).

3. Extract the features of the audio using the `extractFeatures` function you just wrote.

4. Store the results in the struct array `songs`. Each element in this struct array should have two attributes: `songs(i).name` and `songs(i).features`. The syntax is

```
songs(i).name = ....
songs(i).features = ....
```

   in each iteration of your loop. The function `makeDatabase(...)` will return this structure of songs as the database description.

5.2 Simulating Query Generation

Now that you have a complete description of the database in terms of feature vectors, it's time to create a noisy query clip for the system to recognize.

1. Open the file `shazam.m`. In the section titled "Query Generation", start by selecting a random filename from `filelist`. Call this randomly selected filename `queryFilename`. You can look up the function `randi` to do this.

2. Now open `generateQuery.m`. This function is designed to extract a clip from the randomly selected file. You will fill in the missing parts of this function (you should be able to reuse a lot of code from before). Start by opening the input `queryFilename` as before and keep only the first column, then resample it to 4000 Hz as before. Call this variable `clean`, since it represents your clean data.

3. Now you're going to add some Gaussian noise to the query audio file, and store the noisy version in a variable called `noisy`. Look up the function `awgn` to do this. The parameter SNR is short for *signal to noise ratio*, which specifies the ratio of clean signal power to noise power and determines the noise level to be added to the signal. SNR is specified in decibels (dB), which are just a log scale: the power ratio m is written as $10 \log_{10} m$, for example a ratio of 2 is written in decibels as about 3. You can use the SNR that is provided in the function input parameters, but make sure that you remember to tell awgn to measure the input rather than just assuming its power is 1; otherwise, you will generate far too much noise and see very low success rates. Try listening to the noisy audio. With SNR set to 3 (the default), is the noise about half as loud as the song?

4. Extract the features from the noisy version of the audio.

5. From the set of noisy audio features, randomly choose a set of 32 consecutive features to return. These are the noisy audio features that you will use to identify the song.

5.3 Search Algorithm

You can use a simple algorithm to find the location in the database that matches best. For each song in the database, at each position that could possibly correspond to the query, measure the difference between the query and this section of the song using the provided function `ber`; the song which contains the minimum of these differences is your match, provided the difference is less than a threshold. This threshold is set to 50% (i.e. less than half of the bits differ) by default, but you can experiment with it.

6 Measuring performance

1. Go to `shazam.m`. After finding the match check whether it is correct, i.e. the same filename as `query_filename`. (Use `strcmp`, which returns whether its two argument strings are equal.) If this is *not* the case, there is probably a bug in your code somewhere. You can try a few things while debugging, like not adding any noise to the query audio, and/or taking the audio sample from the beginning of the song instead of from the middle. If you do not use any noise (SNR of ∞, written in Matlab as `Inf`), then you should always get a perfect match!

2. Once you have the search working for one song, modify `shazam.m` so that you will test multiple songs. What percentage of songs are being classified correctly?

3. Try lowering the SNR so that you can hear a lot more noise in the signal. Now what kind of recognition rates are you getting?

7 Exercises, Part B

This is your checkoff for the second week.

1. How many seconds of audio are you using to determine what the song is? (The total length in real time of the query.)

2. Create a plot showing recognition rates as a function of SNR.