Matching Snippets

Background

When you use an app like *Shazam!*, you record a *snippet* of a song, and often within seconds, it identifies the song for you. How does it do that? In this challenge, we will construct a system to approximate this behavior using your own library of music or other sounds. This will involve various important ideas – including Fourier transforms, signal processing, hashing, and databases – and will be good practice for software design.

The starting point is a library of audio records which will be searched using a short recorded audio snippet as a key. We will call the audio records *songs*, here, for concreteness, although they can be any type of audio data or other signals. The search key – a short, noisy, and possibly contaminated recording – will be called a *snippet*.

A challenging part of this problem is that we do not know where in the song the snippet should be compared, making the search much hard. Also, the snippet need not be recorded at the same quality and can be contaminated with background sounds (e.g., the exercise bike at the gym) and noise. As a result, with a large song library, the search can be slow and ambiguous.

Our strategy will be to compare a *signature*, or *fingerprint*, derived from the snippet against comparable signatures pre-computed for all the songs. The signatures extract features from the audio and thus can be lower-dimensional, making comparison faster. They also can reduce the noise and the impact of extraneous audio frequencies, making comparison more reliable. If a few songs match based on the signatures, it is not too costly to do a more detailed comparison in those few cases, making the comparison more robust.

The program you write for this challenge assignment will handle two central tasks. First, it will manage a library of "songs," including methods to add, remove, and list songs from the library and to compute (or re-compute) signatures for the entire library. Second, it will search the library for matches to a given snippet. The program will be able to read data from various sources (e.g., files, URLs, streaming connections) and communicate its results in various (e.g., to standard output, to a web server, through an api).

An important method we will use here is a type of signal processing called *time-frequency analysis*. By looking at the distribution of power in the signal across time and frequency, we will be able to compare signals of different lengths in a meaningful way and will be able to clean the signals of extraneous information. Another key method is called *hashing*, the process of using a signature or fingerprint derived from a more complex object to speed comparison and search. Finally, we will use modern *databases* to optimize storage and search.

Mathematically, we will represent the audio signal as a regularly sampled time series $s_j = s(t_j)$, where s_j is the measured signal and $t_j = a + j\Delta$ are the sampling points, for $j = 0, \ldots, T-1$, starting at arbitrary time a with *sampling interval* $\Delta > 0$. (More commonly used in audio circles is the *sampling rate*, $\rho = 1/\Delta$.)

Given such a time series, we can use *spectral analysis* – using the famous Discrete Fourier Transform algorithm – to decompose the signal s into a linear combination of sinusoidal components at different frequencies. This gives us a "frequency-domain" signal s^* where $s^*_k = s^*(\nu_k)$ is the coefficient of the sinusoid at regularly spaced frequencies ν_k.

Because the snippet represents a small slice of time in the song, we want to refine the spectral analysis to account for time. The frequency-domain signal s^* captures the frequency domain structure of the entire song, but we want to distinguish what happens in different windows of time. So, we partition the signal s into (overlapping) windows and do a spectral analysis of each piece. Each sinusoidal component for each window represents a "time-frequency" cell, the structure of the signal in a small interval of time and a small range of frequencies. The collection of windowed, frequency-domain signals captures a great deal of information about the structure of a song. We will compute our signatures from those as inputs.

In slightly more detail, we will define a "window function" w that is 1 near its center and smoothly decays to zero. (More specifically, $w()$ is a smooth, compactly supported window function that approximates a Gaussian density.) We then compute *windowed* times series around center c_i: $v_{ij} = w(t_j - c_i)s(t_j)$, where c_j is a time representing the center of the window. The c_i's and $w()$'s are chosen so that the windows overlap; for instance, we might have the c_j's differ by 1 second while the windows are of width roughly 10 seconds. The window size corresponds roughly to the shortest snippet we will be able to match.

Each of these time series v_i is concentrated in time around a_i. We do a spectral analysis of each v_i to estimate the distribution of power in the signal in that window. We then partition the frequencies non-overlapping intervals, such as *octaves* which have lengths equal to successive powers of two. These window-by-octave cells gives us a breakdown of the signal's time-frequency content. We can, for instance, use the positions of the maximum power in each cell as a summary of the signal, but many summaries are possible. Each window thus gives a signature that is stored with the song.

When given a snippet we do the same windowed spectral analysis, but just for those windows that let us analyze the whole snippet. Suppose for discussions that the snippet is short enough that we can only produce one windowed time series from it. The spectral signature of this time series is then matched to the stored signature. Is there a match? If so, we return the song (or songs).

The remaining subsections in this Background give more details on some of these steps. On first reading skip to the description in the Task section.

Reading Audio Data

Music is a good test bed for this application because its audio signal has a nice structure. You can use music or other sounds for testing your code. Bird calls are another possible data source with similarly interesting structure (see the Macaulay Library). But any type of audio data is fair game. For the purposes of discussion, here we will talk about it in terms of songs, but you can substitute whatever data you like.

For practical purposes, to analyze songs, we need to decode the data representation, and we will need audio files. The MP3 format is common, but the WAV format will be easier to use. There are a variety of ways to convert the one to the other, including iTunes and the

popular VLC media player; see `http://www.wikihow.com/Convert-MP3-to-WAV` for a list of good options. While you can certainly invoke external commands to handle this conversion in your program, even better would be to handle that conversion with a library.

Once a song is in `.wav` format, we can get access to the time series using some easily available libraries. In R, the `tuneR` package does the job; in Python, the `wave` package; in C/C++, the `libsndfile` library; in Java/Clojure/Scala, the `javax.sound.sampled.AudioInputStream` class does the trick.

The `.wav` file produces a time series of regularly spaced amplitudes. The *sampling rate* parameter tells you how frequently those samples were obtained; it also determines the maximum (unaliased) frequency you can get in the spectral analysis phase later.

Ideally, your music library is available as a collection of audio files (e.g., in iTunes). However, if your music (or sounds) are only available via streaming or in videos, you will need to capture the audio. For this purpose, Audacity (link) is a useful tool; it is free, open-source, cross-platform, and commonly used. Here are some links to how to do this:

- Audacity tutorial: recording on windows (link)
- Audacity tutorial: recording on a Mac (link)
- Article on alternate methods (link)
- Another article on methods for capturing audio (link)
- Article on capturing audio from streaming video (link)

To maintain fair use and avoid copyright issues, you should start with music that you legitimately have access to. You should not redistribute or post any captured audio. You should use any captured songs solely for the purpose of testing your software and ideally would delete the files when the project is complete.

Windowing

Our strategy is to use a window function that approximates a Gaussian, because the Gaussian gives the best concentration over time and frequency. (Lookup the *Gabor transform* for more on this.) With a finite time series, however, we cannot use a Gaussian without truncation, which causes distortion that spreads the power inefficiently. So, we approximate the Gaussian with a compactly supported function that decays to zero smoothly. Two reasonable options are the Hanning window:

$$w(t) = \begin{cases} \frac{1+\cos(2\pi t/h)}{2} & \text{if } |t| \le h/2 \\ 0 & \text{otherwise,} \end{cases}$$

and the Blackman window

$$w(t) = \begin{cases} \frac{0.84+\cos(2\pi t/h)+0.16\cos(4\pi t/h)}{2} & \text{if } |t| \le h/2 \\ 0 & \text{otherwise.} \end{cases}$$

We center our windows at evenly spaced points some small time Δ apart; i.e., $c_j = j\Delta$. This gives us two tuning parameters: h specifies the *window size* and Δ (or equivalently $\rho = 1/\Delta$) specifies the *window shift*. We want the window size h large enough that the windowed time series has enough information about the signal's frequency content. But we also want h small enough that we can match against brief snippets of a song, because roughly speaking, h gives the length of the smallest snippet we can analyze. Similarly, we want the window shift Δ to be small enough that we can ignore the misalignment between the recorded snippet and the windowed time series in our library, but we do not want it so small that we have to compute too many windowed time series. The choice of the window size (h) and shift (Δ) tuning parameters will need to be empirically drive, but 10s and 1s, respectively, seem like good starting points.

Spectral Analysis

Spectral analysis of a signal estimates the density of signal power at each frequency. There is a great deal we could say about that, but for our purposes, we can keep it simple.

Given a signal $x_j = x(t_j)$ of length n, we can compute what is called the **periodogram**, which is the vector $(x_k^* : \ -n/2 < k \leq n/2)$, where result:

$$x_k^* = \frac{1}{n} \left| \sum_{j=0}^{n-1} x_j e^{2\pi i j k/n} \right|^2 ,$$

This is just the squared modulus of the signal's Fourier Transform and it estimates the *power* in the signal at frequency $\frac{k}{n} f_{\text{Nyq}}$. Here $f_{\text{Nyq}} = 2\rho = 2/\Delta$ is called the Nyquist frequency; it is half the sampling rate of the time series, or equivalently the reciprocal of twice the time interval between measurements. In practice for signal analysis, we often further smooth the periodogram – for instance with a kernel – to obtain a better estimate of the signal's spectral density. This can markedly improve the results but is considered an extension here (see Part 3).

You will need to write a function to compute the periodogram of a windowed time series. In R, the periodogram can be computed with the function `spec.pgram`; in Python, the periodogram can be computed with the function `periodogram` in the `scipy.signal` package.

Spectrograms

We are now in the situation where for each window center, we have a periodogram describing the frequency structure of the signal within that window. In what follows, the (optionally smoothed) periodogram of a windowed time series will be called a *local periodogram*, or the *local periodogram centered at c* if referring to a specific window centered at c.

To help visualize the time-frequency structure of the song, we can construct a matrix S, which some call the *spectrogram*, where row j contains the local periodogram centered at c_j. Plotting this matrix as an image, with window centers along the horizontal axis and frequency along the vertical axis, gives a visual summary of the song. See `shazam/Resources/Ch5extract.pdf` for example pictures of spectrograms. One of the management sub-tasks of your program can be to plot the spectrogram for a song in your library.

This is not necessary for our main task but can be useful for data validation and visualization and is often quite interesting. (Note for R users: the `specgram` function in the `signal` library can also be used to compute spectrograms.)

Computing Signatures (aka Fingerprints)

Now that we can describe a song's time-frequency structure, we want to compare that structure to the time-frequency structure of the recorded snippet and find the song that contains a segment that matches as closely as possible. We would *like* to use the local periodograms to make this comparison, but this can be inefficient in terms of computation and storage.

So instead, we consider *signatures* – lower-dimensional summaries of the local periodograms – as alternative keys for this comparison. We will compute a signature for each local periodogram (i.e., each window center) for both the library songs and the snippet. We want our signatures to contain as much information as possible about the time-frequency structure, be robust to noise in original signal, and be as concise as possible to minimize storage and computational cost.

Finding a good signature is a component of the task in Part 3. Some reasonable choices to consider initially are:

1. the unsmoothed local periodogram itself,
2. the smoothed local periodogram,
3. a list of pairs (f, p) containing the frequency (scaled to $[0, 1]$) and power (scaled to $[0, 1]$) at peaks in the local periodogram,
4. a list of (positive) frequencies f (scaled to $[0, 1]$) at which the local periodogram has a peak, and
5. a vector (p_k) where each p_k is the maximum power in the local periodogram (smoothed or unsmoothed) within the kth among m pre-specified, non-overlapping frequency intervals (such as successive octaves, frequency bands doubling/halving in width, down to some minimum frequency $2^{-(m+1)} f_{\text{Nyq}}$).

The first two have good information content and reasonable robustness to noise, with the second being somewhat better in both regards; but both are likely too high-dimensional to be practical. The last three are much sparser, tend to be robust, and likely contain useful information about the structure. The scaling of frequency and power to $[0, 1]$ (relative to f_{Nyq} and the local periodogram's maximum power, respectively) focuses on the "shape" of the signal and eliminates extraneous details. Either of the last two choices seem like good starting points.

There are other signatures possible. The paper `shazam/Resources/Wang03-shazam.pdf` has an approach related to #4 above. Instead of matching one local periodogram at a time, that paper uses the list of peak coordinates in a *strip* of the periodogram (successive sequence of rows in S, or a vertical strip in the plot), relative to the time at which the strip begins. A snippet of a given length has its signature (for the corresponding number of windows) compared only to signatures in the library for strips of the same length.

You will need to write a function or class to compute a signature from a local periodogram.[1] Keep in mind that you might want to experiment with different signature methods, so your design should make it easy to replace or configure the code to use different signatures.

Organizing the Library for Search: Hashing

Once the signatures are computed for the songs in the library and for the recorded snippet, we will want to find the song that has a signature closest to those of the snippet. Keep in mind that a snippet's signature will be compared to the signature for each window of the song, and to be close, a snippet and song are close if there is window location that makes the two signatures close mathematically. (Put shortly, for each song, we take the minimum distance over all signatures associated with that song.) This is effectively a nearest neighbor problem, and we can organize the library in several ways in a data structure that facilitates this search.

The data structure must support a search that takes a signature and a threshold ϵ and an optional number of matches N (default 1). The function should return `NULL` if no match is found (i.e., the closest signature is farther than ϵ from the given signature). Otherwise, for the N songs with the (approximately) closest matching signatures, it should return the matching signature, the corresponding song id (e.g., the primary key in your song database table), and the corresponding window center (e.g., a_j for the associated local periodogram). The function will return the closest match found if the signatures are within ϵ, otherwise `NULL`.

The simplest search approach, of course, is linear search through a list of signatures. While the performance of this approach will be too slow in practice, *it is a good starting point* for getting the rest of your code running. Specifically: we recommend that you start with brute-force linear search (over a database table, say) until everything else is working.

If the signature dimensions are not too high in general, KD-trees are an option (see the exercise `kd-tree`). PostgreSQL also has some modules that can be used for nearest-neighbor search on vectors (see `btree_gist`, which is available built-in, and `sp_gist`), making it possible to keep the signatures in a suitable SQL database and get the performance similar to a kd-tree.

A good approach here is to use Locality-Sensitive Hashing (see the class lecture notes and the exercise `lsh-simple`). The `FALCONN` library discussed in class is immediately useful here. It can be directly used from Python or C/C++. For R users, the `Falconnr` package at `https://github.com/genovese/falconnr` gives a flexible wrapper around the `FALCONN` library. See the `README` and vignettes for details.

Database Schema

The library will be a collection of audio files along with a database containing essential information about each song. The database can be structured in several ways, some better

[1] More generally, you could write a function to take the spectrogram matrix and produce a collection of signatures. If it returns a list of signatures with one per local periodogram, it will mimic our approach, but it can also extend to operate like in the Wang paper and return signatures for every strip.

than others, but a schema needs to be designed to represent the information needed for search and output.

For each song in the library (and for each song added later), at least the following information is needed:

1. A unique identifying key, which is not reused even if a song is deleted.
2. The song's title, artist, date, album, et cetera.
3. An address (e.g., on the file system or server) for the song's WAV audio file. (This could also be maintained as a blob of data in a database field.)
4. External source for the original file (e.g., a URL), if applicable.
5. The song's local periodograms (or equivalently the spectrogram).
6. The song's sampling rate, start time, window centers, and length.
7. The signatures derived from the song, either addresses for the files or raw data. (More than one field may be required if several signature types are used simultaneously.)

Initially (Part 2), a simple data arrangement is fine; in Part 3, the database design will be part of the task.

Matching

A schematic of the overall process is as follows.

First, the library is constructed. For each song in the library (and for each song added later):

1. The song's information (e.g., title and artist) is entered into the data base.
2. The song's signal is converted into numerical form (e.g., MP3 $\rightarrow$ WAV $\rightarrow$ numeric vector).
3. The song's local periodograms (or equivalently the spectrogram) are computed.
4. The signature is computed for each local periodogram.
5. The signatures are stored (along with song id and window center) in the search data structure, which should be persistent (i.e., saved to disk in files or in a database).
6. The signatures should also be accessible by window center (index), so it might be useful to store them in a database table as well. (For instance, if you find a match at one window, you want to check the match of the song at the next window against the next window of the snippet.)

Second, the recorded snippet is given:

1. Compute local periodograms for each window that fits in the snippet.

2. Compute corresponding signatures.

3. Starting from the snippet's first window, find songs that match successive windows in the snippet, using the previous set of matches (initially the whole library) at each stage.[2]

4. Return the relevant information of the matching songs, or an indication that none matched.

Task

The task is to write a system to implement this basic algorithm (or some variant) and handle a variety of required tasks to use the system. Here, we will call the program `freezam`, but you can name it as you like. This should be a command-line program that handles options according to good, modern convention. The command-line program can be used (see Part 4) to spawn a web server that handles requests in a distributed fashion. As described above, your system will need several distinct tasks, including managing the library, analyzing songs, and recognizing snippets.

Part I: Design, Interface, and Testing

Design

Your goal here is to create the structure and organization of your program. This includes identifying entry points into your code, writing stub functions for the main components of your code, and organizing those functions into modules that separate concerns and simplify dependencies.

The *entry points* are the main ways your program or outside services access the functionality you provide. This might include a "main" function that is invoked when your program is run from the command line, a library entry point that can be used by other software using your code as a package, and a web server that accepts requests over the network. All of these can coexist in your code, though they will likely be in different modules.

A *stub* function has a name and calling signature (e.g., argument names and types), and a default return value (which may be incorrect at the moment). Writing the stubs lets you define and organize the functionality your code will use/provide and allows you to write real tests (that will fail initially). Identify the main computational steps and have at least one function for each. Remember that your functions should do one thing well and present a clean and consistent interface. Group the functions thematically. For instance, functions related to processing audio files might go in one module and functions that handle spectral analysis in another. This enables you to use those modules throughout your code and keeps your dependencies manageable. Since your program needs to support a variety of distinct tasks, another criterion for organization might be to group modules related to each task and then have utility modules that are used by all tasks. (In Python for instance, you can have

[2]The Wang paper did not need this step as it used most of the snippet at once to compute a single signature.

sub-packages within your main package directory; so you can have a sub-package for each task and sub-packages for utility code used by the others.)

Some potential functions to consider:

☐ Input/output functions to read data from a variety of sources. Consider using a class or a "function factory" that lets you pick the function used based on configuration of the program. (For instance, are you reading data from a file or from a network socket? You will need a different function for each.)

☐ A function to convert different audio formats (e.g., `.mp3`) to `.wav` data.

☐ A function to apply a window function to a time series given window centers and related parameters.

☐ A function to compute the local periodograms from a windowed signal.

☐ If you may later support smoothing, a function to smooth a periodogram.

☐ A function to analyze a song and enter it into your database. This should check if the song is already present and either ignore repetitions or update as you see fit.

☐ Functions to manage your database (adding and removing songs, for instance).

☐ Functions to compute signatures.

☐ A function for testing the match of two signatures.

☐ A function for searching the databaser.

In addition, you should stub a function *`slowSearch`* that takes a snippet and searches linearly through the database comparing the local periodograms without a special signature.

Interface

Your program should have a command-line interface of the form

```
freezam subcommand [OPTIONS] [ARGUMENTS]
```

where each "subcommand" is a word indicating the operation to perform. For example,

```
freezam add --title="Song title" --artist="Artist name" song.mp3
freezam identify snippet.mp3
freezam identify snippet.wav
freezam list
```

Your command should support the standard `--help` and `--version` options, and `--help` should give different guidance depending on the sub-command:

```
freezam --help
freezam identify --help
freezam add --help
```

where the first gives general help and the latter two give help tailored to each sub-command.

In this part, you should arrange for your program to accept the basic commands (but new nothing) and to provide appropriate help. Try to define and accept an appropriate set of command-line options for each sub-command, though these can change later.

Testing

Write tests for each core functions that you stubbed. Pure functions such as those for windowing, spectral analysis, signatures, and comparison should be especially well tested. You can keep static data files separately for your tests; for example, to test file conversion with a known outcome.

For the spectral analysis tests, try some simple signals for which the periodogram can be computed by hand easily. Similarly, smoothing can be tested using a signal with non-zero entry, which should reproduce the kernel. You can (and should) add additional tests later, but try to be as comprehensive as possible.

Part II: Basic Implementation

In this part, we focus on the core computational steps and do not worry as much about making the search efficient. You should implement the following.

Input/output:

☐ Read an audio file

☐ Read audio data from a URL

☐ Implement logging functionality that writes a message for each action taken to a designated log file (with a default name). Warnings and error messages should go to the log file but also to standard error. If the `--verbose` option is given, all log entries should be output to standard error as well as the log file.

Managing database:

☐ Adding songs to the library and database

☐ Listing a useful summary of the library contents

☐ Analyzing a single song or the entire library

☐ Output spectrogram images (or data if prefer ed) for a given song

The initial database can have a simple structure (e.g., using a text files and the file system), but keep in mind, you will eventually be putting the database into a better form. In this part, you should also build up a test library of sounds for later work.

Search and Recognition:

☐ Compute local periodograms for snippets

- ☐ Compare song to snippet using whatever signature is being used (initially the entire local periodogram).
- ☐ Start with *slowSearch* which uses the entire local periodogram for comparison and get the basic implementation working.
- ☐ Implement one or two basic signatures and compare the performance with your *slowSearch* results.

Again, in this basic implementation, you can use a simple database and linear search. Be sure to include relevant tests and documentation.

Part III: Advanced Implementation

In this part, we power up the implementation. At the very least, you should refine your code in the following dimensions:

- ☐ Design a database schema and use a modern database system, such as PostgreSQL, to store your database.
- ☐ Add sub-commands to allow removing songs and updating songs' basic information.
- ☐ Implement several signatures and analyze your data set systematically to compare their performance and select the signatures you will use downstream.
- ☐ Insure that your song analyzer can ingest an entire directory to make it easy to analyze a library. (In other words, when the file given is a directory, analyze the audio files within it, or even recursively beneath it.)
- ☐ Structure your database to support more performant search, such as using locality sensitive hashing.

In addition, you can optionally implement an advanced signature like that described in the Wang paper.

Part IV: Integration and Product

In this part, you will bring your code to a shiny polish, producing a useful tool. How you do this is open ended. A few ideas:

- ☐ Implement a `server` sub-command that starts a web server in the background and accepts network requests for adding, analyzing, listing, and recognizing songs.
- ☐ Generate a web-service or desktop application that can record and analyze a snippet.
- ☐ Add a `recommend` sub-command that takes in one or more songs (e.g., those liked by the user) and recommends songs that are similar from within the library.
- ☐ In the `recommend` sub-command, add a service that optionally plays the recommended songs.

Requirements

A **Mastered** submission will:

☐ Include a database schema or organization for your data.

☐ Populate your database with songs using your song analyzer. Add as many songs as you can, though if possible having from 50-100 makes it easier to see how well the algorithm works.

☐ Analyze songs in and out of your library and record the matches. Report the error rates of your matcher.

☐ Write tests for your system.

☐ Well-designed, modular code with effective separation of concerns, and good choice of function and/or object interfaces and that allows easy change of hashing method, window function, and tuning parameters.

☐ Write a command-line interface to your library that handles useful tasks such as: adding new songs to your database, identifying songs from audio snippets, and listing information about the song library.

A **Sophisticated** submission will additionally:

☐ Use excellent variable names, code style, organization, and comments, so the code is clear and readable throughout.

☐ Be efficient, making the best use of its data structures with minimal copying, wasteful searching, or other overhead.

☐ Supply a command-line driver that runs all your tests.

☐ Meet and exceed the requirements in Parts I–IV.

☐ Provide a creative and elegant product in Part IV.

You should also remember that **peer review is an essential part of this assignment.** You will be asked to review another student's submission, and another student will review yours. You will then revise your work based on their comments. You should provide a clear, comprehensive, and helpful review to your classmate.