Muse: Unleashing Music

MuseUM: Offline Music Suggestions & Playlists

Abstract and Motivation

When I tried to leave streaming platforms—in part due to ethics, in part simply because I found the experience unsatisfactory—I ran in to quite a number of problems. Firstly, I needed to accumulate a large digital music library. thankfully Bandcamp came to the rescue (though less legitimate sources are available also), and throughout a few years I accrued a quite large music library, amassing around five-hundred gigabytes of FLAC music files. While this was well and good, it did not really serve its purpose. I already had a large record collection for critically listening to individual albums and artists; I had rather wanted the music library to replace my streaming needs, which were overwhelmingly in the nature of music suggestions. When I streamed music, I usually did it because I simply wanted to click on a pre-generated playlist of a specific mood, or alternatively shuffle-play a specific artist. I streamed, because I wanted a software to intelligently give me background music without me having to interact frequently. I did not want streaming to introduce me to new things—I have record shops and Bandcamp for that—nor did I want it for critical listening. I really needed it for *hands-off* listening. This means I simply wanted to open an app, have a few options to select, and then let the algorithm run. This may have been selecting a generated playlist, or shuffling a mega home-grown playlist, or even just shuffling a specific artist. But clearly I heavily relied on the streaming service's algorithm.

Naturally my next step was to find a similar offline service to do this for me as well, with my five-hundred gigabyte library. But well and behold, I could not find anything satisfactory. Either I needed to set up a server for a mediocre experience (no thank you), or live with literal shuffling (i.e., totally randomized results with no intelligence whatsoever). Faced with this horror, I decided to use my programming skills—however, limited they may be—for the better, and create my *own* tool to achieve this end. Thus *Muse: Unleashing Music* (MuseUM, or simply Muse) is born.

Algorithm and Goals

I had a few things in mind, when it came to how to implement Muse. Firstly, I wanted everything to be offline. One of the big advantages that streaming services have, is that they

Simply Algorithm

This the non-learning one, that is used by default. The more complex algorithm requires a lot of data, especially with large libraries, meaning that there needs to be a solid, basic algorithm to take care of you until then. Additionally, the more complex algorithm is also prone to making mistakes, so this acts a little like a gatekeeper.

The algorithm works as such:

```
let touches: u32; // how often has the song been suggested.
let listens: u32; // how often has the user listened to the _entire song._
let skips: u32; // how often has the song been skipped.
let mut score: f64;

// Omitting type conversion for clarity
score = if touches < 30 { // song is still fairly new
  let (weight_listens, weight_skips) = self.weight();
  // Remember that `listens` does not include `skips`
  return (weight_listens * listens) - (weight_skips * skips)
} else { // song is well known
  self.dampen() * listens - self.dampen() * skips
};

if score < 0.0 {
  score = 0.0;
}
```

You may notice the `weight()` function. This function is exceedingly simple:

```
    fn weight(&self) -> (u8, u8) {
        let low = 1;
        let medium = 2;
        let high = 4;

        let small_threshold = 5;
        let big_threshold = 15;

        if self.touches < small_threshold {
            // Listens are 4x more important than skips
            // This means that early, anecdotal skips are disregarded.
            (high, low)
        } else if self.touches <= big_threshold {
            // Listens are equally important to skips, but the _difference_ between
the two is increased.
            (medium, medium)
        } else {
            // Skips are 4x more important than listens.
            // So skips still take an effect,
            // and the algo learns with stability.
            (low, high)
        }
    }
```

Note that part of the existence of this function, is that I found it more effective, if "new" songs, meaning songs that the user has not listened to that often, are given a strong advantage over old songs. This means that you get much variety, and the algorithm doesn't become repetitive with age.

Another function, for songs listened to more than thirty times, is `dampen()`:

```
    fn dampen(&self) -> f64 {
        // `+1` just in case.
        // `1.2` seems to be ideal.
        let weight = f64::from(self.touches + 1).log(1.2);
        weight
    }
```

This function is as such similar to `weigh()`'s medium weight, but logarithmically growing. Instead of statically staying at two, thus only doubling the difference, the coefficient grows as the user listens to the song more frequently, and thus results in a tendency to weight recent preferences over the sheer frequency of suggestion (i.e., frequency vs. recency).

Complex Algorithm

In addition to this rather simple algorithm, there is a more complicated one at work as well. This one is responsible for *which* songs get suggested to the user.

The complex algorithm remembers which songs were played relative to the next song. This means that each song has data attached to it, that stores all the songs it was before. For example, if you play Song A and then, when it is finished, play Song B fully, the metadata for Song A will be updated to include Song B with a counter. If this scenario repeats itself, the counter in Song A's metadata for Song B will increase by one.

This repeats itself, until nearly every song is "connected" to every other song. What this means, is that you now have very strong data for which songs work well with each other. This essentially lets you identify the user's own sense of genre and mood, and as such means that when you play Song A, you immediately have a list of potential follow-up songs at your fingertips.

The score of the songs, as generated by the previous algorithm, is then multiplied dynamically for this specific situation (i.e., not permanently) by that counter I mentioned earlier. To prevent this from becoming too extreme, however, that counter is first taken the base 1.2 log of. Then these songs, and ten songs selected at random (the algorithm needs to learn somehow, after all), are sorted, and the one with the highest score is played for the user.

Controls and Queue — Playback

The user has a few options. They can listen the the entire thing or skip it (`skip` command). There are also, of course, basic controls for playing, pausing, and stopping, as well as selecting `shuffle` or specific algorithmically generated playlists (further discussed in the following section).

The difficulty of this style of application, however, is that there is no real queue. Songs are suggested dynamically. So to enable the user to get an impression of what will play next, there are also the "next?" and "skip?" controls, which will tell the user what plays next if they listen to the entire song and if they skip it respectively.

Alternatively, the user *can* set a queue if they want it. These are called "trains of though" and three version exist: *currents, threads,* and *streams.* These are always based off a song, and named as such.

Currents

Currents are a mix of two connections generated by the "complex algorithm." Essentially, two alternative pathways, starting from the same song, are generated and mixed.

Here is an example current, “Song A Current”:

```
Starting Song: A

Song A’s Connections:
- Song B
- Song C
- Song D
- Song E

The two top (highest scoring) songs: Song B and Song E

Therefore, two currents start:
- Current Song B
- Current Song E

First, the Song B current is calculated:
- Song B top connection: Song 37
- Song 37 top connection: Song 16
- Song 16 top connection: Song 193
- Song 193 has no connections (with positive score).

Second, the Song E current is calculated:
- Song E top connection: Song 945
- Song 945 top connection: Song 473
- Song 473 top connection: Song 14
- Song 14 further searching is stopped as the Song B current is already finished.

Now, both of these connections are mixed resulting in this current (queue):
- Song A
- Song B
- Song E
- Song 37
- Song 945
- Song 16
- Song 473
- Song 193
- Song 14

As this current is >=9 in length, “Song A Current” is submitted.
```

Thus, a nine-song queue is generated with an interesting mix in genre. Current length is variable, the current will become longer with each connection that is found, but will stop when no connection with a positive (bigger than zero) score can be found. It is always ensured that These is also a minimum of nine and a maximum of 27 songs.

Threads

Threads are identical to currents, but only *one* instead of *two* are generated. As such they are simpler, and don’t have the occasionally odd song combinations. They are the same length.

Streams

Streams are identical to Threads, but always exactly thirty songs long, and with much more randomness. This provides a little more variation. Essentially, every time at least three connections aren’t present, ten songs are taken at random, and the one with the highest score is chosen (this in effect creates a new connection). Additionally, instead always choosing the top connection, one of the *positive score* connections is taken at random, which provides fair competition between the connections, instead of simply strengthening the top one (and snowballing).

Streams are excellent for training the algorithm! If you are new, I recommend either shuffling or using Streams, as these will get the algorithm up and running as fast as possible.