Genre Comparison through Music Signal Processing

Jeremy Grifski

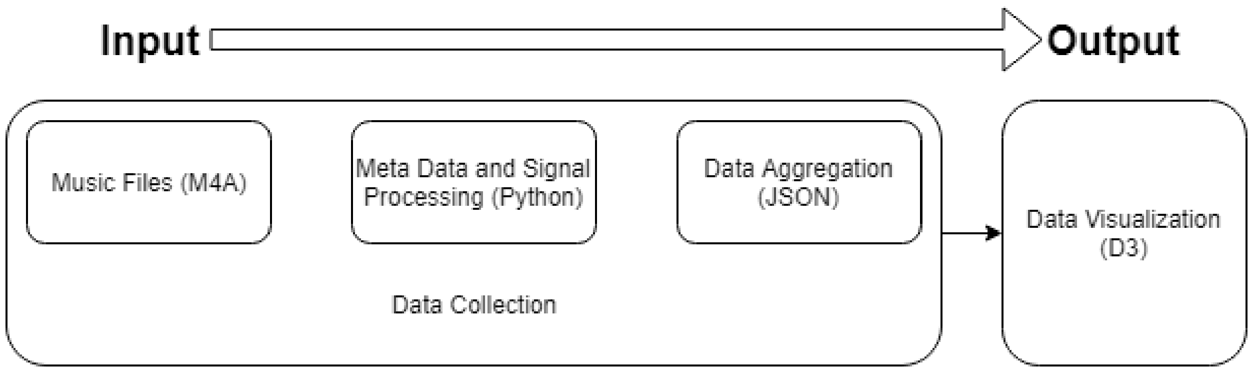

Fig. 1. An overview of the music file to visualization pipeline.

Abstract— Genre is a popular way to classify styles of music, and people may argue there is merit in genre classification. But, are there any tangible differences in music styles that can be discovered through data visualization? In this project, I attempt to answer that question by applying a handful of signal processing methods to a set of music files and visualizing the results relative to genre. All code for the project can be found at https://github.com/jrg94/CSE5539 and the live visualizations can be viewed at https://jrg94.github.io/CSE5539/.

Index Terms—Signal Processing, Python, JavaScript, D3

1 Introduction

As mentioned in the proposal, I am a first-year PhD student with a limited research background. My interests are mainly in music, education, gaming, and data visualization, so it is probably no surprise that I chose to do a project with two of those interests in mind: music and data visualization.

My goal with this project was to explore different genres of music through data visualization while dipping my toes in some basic signal processing. In particular, I chose to analyze a personal collection of songs in two stages: **data collection** and **data visualization**.

In terms of data collection, I spent some time learning how to parse a common file format, M4A, and interpreting that data. As for data visualization, I spent some time trying to address tasks such as:

- How frequent are the various genres in my personal library?
- How does music loudness vary by genre?
- How has music duration varied over time?

In the following sections, I will further explore the processes and results of data collection and visualization.

2 Data Collection

In order to visualize the differences in music genre, I had to first collect some data. Of course, there are a lot of options. For example, music files come in many formats including raw audio formats like MP3, WAV, and M4A as well as descriptive music formats like MIDI. In addition, there are several music mediums like CDs, records, files, sheet music, and live performances.

For the purposes of this project, I chose to use music files from my own personal collection. Unfortunately, that means this project isn't exactly reproducible as the files are not open-source. That said, the software used for processing and visualizing music files are open-source.

In the following subsections, I will cover how exactly I collected my data, what the music file processing software looks like, and how I organized the data for visualization.

2.1 Music Files

Fortunately, I had quite the collection of songs from iTunes purchases over the years. In particular, I had 4,194 songs spread across 983 folders. From there, I had two challenges: choosing which file format to process and selecting what data to collect from those files. Of the 4,194 songs, roughly 1,800 of them were in the M4A format, so I chose to parse that format.

In terms of data collection, I decided to mine several fields from the M4A files such as the fields found in Table 1. Many of these fields were available directly in the M4A file. However, some of the fields had to be calculated using information provided in the file. For example, duration d was not provided directly. It had to be computed from the sample size n, number of channels c, sample rate r, and audio size a, or more formally:

$$d = a/(c * n * r)$$

Here, I am assuming that audio size and sample size are both in bytes. As a result, a song with two channels (stereo), a 44,100 Hz sample rate, a two-byte sample size, and a 176,400-byte audio size would contain just one second of audio.

Likewise, I also had to compute average decibels relative to full scale (dBFS), maximum dBFS, root-mean-square (RMS), and maximum amplitude which are all parameters related to audio loudness [8]. Fortunately, I did not compute these parameters myself as they were

- *Jeremy Grifski is an OSU PhD Student. E-mail: grifski.1@osu.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

Table 1. Data Fields

field	type
Title	String
Genre	String
Artist	String
Album	String
Purchase Date	Date
Release Date	Date
Track Number	Integer
Sample Rate (Hz)	Integer
Sample Size (bits)	Integer
Duration (HH:MM:SS)	String
Number of Channels	Integer
Average dBFS	Float
Max dBFS	Float
RMS	Integer
Max Amplitude	Integer

provided through the Python pydub library [9]. However, for the sake of completeness, the following paragraphs will describe how those values can be obtained.

Of the four parameters, the easiest to compute would be peak amplitude. Based on the pydub documentation, amplitude would be the largest sample across all samples. For example, a song with 16-bit samples would have a maximum possible amplitude of 2^{16} or 32,768.

From there, maximum dBFS can be obtained by computing the dBFS of the peak amplitude relative to the maximum possible amplitude which depends on the sample size. In terms of math, the equation for maximum dBFS might look something like:

$$20 * log_{10}(|sample|/reference)$$

where sample represents the sample value and reference represents the maximum possible amplitude. Based on this equation, it's clear that the closer the sample is to the reference value, the closer the overall equation trends toward zero. As a result, dBFS is typically given as a negative number where values closer to zero indicate louder samples.

Another helpful measurement is root-mean-square, which as the name implies, gives us the square root of the mean square, or more formally [10]:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

Here, x is a sample and n is the number of samples. In other words, we compute the sum of all the samples squared, divide by the number of samples, and calculate the square root of the result. The final number describes the effective value of the waveform.

To compute average dBFS, the same equation for peak dBFS can be used. However, instead of using an individual sample, we compute the root-mean-square of all samples. As a result, we obtain a rough idea of loudness over some time period.

In the following sections, I will discuss how I parsed the M4A files, and how I aggregated that data for visualization.

2.2 Music File Parsing

The first major challenge in this project was finding a way to collect information from the M4A file format. Unfortunately, I was unable to find a tool which could easily glean all the data I wanted from an M4A file. As a result, I decided to write my own file parsing code in Python.

With the aid of the Quicktime documentation, I was able to write a Python script which could roughly interpret the various atoms of an M4A file. In particular, I wrote enough code to parse all of the atoms seen in Table 2.

An atom is a section of an M4A file that stores information based on some accepted standard [1]. Some atoms contain atoms, so parsing can be challenging. Fortunately, Python makes it relatively easy to read and parse byte streams. The challenge was properly mapping each atom based on the standard.

Table 2. All Parsed Atoms

atom	description
moov	the movie atom
trak	the trak atom
mdia	the media atom
minf	the media information atom
stbl	the sample table atom
udta	the user data atom
dinf	the data information atom
pinf	-
schi	-
ftyp	the file type atom
mvhd	the movie header atom
hdlr	the handler atom
mdhd	the media header atom
dref	the data reference atom
smhd	the sound media information header atom
stco	the chunk offset table atom
stsc	the sample-to-chunk table atom
stsd	the standard description atom
esds	the elementary stream descriptor atom
stsz	the sample size table atom
stts	the time-to-sample table atom
tkhd	the track header atom
meta	the meta data atom
ilst	the item list atom
cpil	the compilation atom
gnre	the content genre (i.e. rock) atom
rtng	the content rating (i.e. explicit) atom
stik	the media type (i.e. movie) atom
mdat	the media data atom
frma	the data format atom

In terms of design, the M4A file parsing script, called m4a_parse.py, defines a function for each atom defined in the specification. Then, the script begins by grabbing the first atom which is interpetted by reading the first 8 bytes of the file. The first four bytes give the size of the entire atom including the 8 byte header. Meanwhile, the second four bytes give the name of the atom. From there, the name is used to retrieve the atom-specific parsing function, and parsing of that atom begins. Naturally, the process continues recursively until all atoms are interpetted and stored in a dictionary.

In total, the M4A parsing code is roughly 800 lines of code, yet it is still incomplete. In particular, I was unable to parse the actual audio data as M4A files are in many cases compressed. To account for this setback, I leveraged a Python library called pydub which uses FFmpeg to parse audio files of many formats including M4A [9].

2.3 Music Data Aggregation

Being able to parse an M4A file and being able to use that data for visualization are two separate challenges. As a result, I had to find a way to aggregate the parsed data in an easy-to-read format. With the target environment being D3 and JavaScript, I chose to format all the data as JSON files. JSON is easy to load in JavaScript and even easier to change as needed. An example JSON might look something like the following:

```
{
    "path": "E:\\Plex\\Music
    \\A Loss for Words\\The Kids Can't Lose
    \\01 Stamp of Approval.m4a",
    "genre": "Alternative",
    "title": "Stamp of Approval",
    "artist": "A Loss for Words",
    "album": "The Kids Can't Lose",
    "track_number": 1,
        "total_tracks": 11,
        "content_rating": null,
        "owner": "Jeremy Grifski",
        "release_date": "2009-05-12T07:00:00Z",
        "purchase_date": "2014-02-10 16:44:15",
        "sample_rate": 44100,
        "sample_size": 16,
        "length": "00:03:06",
        "number_of_channels": 2,
        "dBFS": -11.080573225629289,
        "max_dBFS": 0.0,
        "rms": 599654648,
        "max_amplitude": 2147483648
}
```

With a JSON like this, it's very easy to retrieve information about a song like its genre or title. In addition, the JSON can be easily expanded to include information like the occurences of certain pitches.

In order to generate the JSON, I created two Python classes: MusicFile and MusicFileSet. A MusicFile provides music file data storage as seen in the JSON above. In addition, the MusicFile class provides extended functionality like the ability to convert an M4A to a WAV file and the ability to persist the parsed atom data to JSON [6].

In order to provide all that functionality, the MusicFile object has to be able to handle the dictionary data provided by the m4a_parse.py script. Since the dictionary is just a more readable format of the raw binary data of the M4A file, the MusicFile object just needs to know the atom hierarchy to obtain the proper data. For example, genre can be obtained from the following atom hiearchy: data $\rightarrow$ moov $\rightarrow$ udta $\rightarrow$ meta $\rightarrow$ ilst $\rightarrow$ gnre. As an added wrinkle, most of the meta data leverages numerical codes. For instance, genres are mapped to numbers, so there's a separate lookup table for genre strings. For the sake of simplicity, I only chose to track all the fields that you can see in the JSON above.

Meanwhile, each MusicFile can then be stored and manipulated in an instance of the MusicFileSet class. The MusicFileSet class is essentially a list wrapper, but it provides several key functionalities such as [7]:

- the ability to load all M4A files from a directory recursively
- the ability to filter a MusicFileSet by genre or size
- the ability to dump the entire set to JSON

Naturally, I leveraged the MusicFileSet class to build a large collection of data files which encompass various sample sizes and genres. In total, I was able to generate 33 data sets.

3 Data Visualization

The goal of this project was to learn some signal processing while trying to find interesting trends in genres of music. To do so, I felt it was appropriate to try to plot some of the data I collected in D3.

D3 is a JavaScript library that allows users to generate interactive web-based visualizations. It works by providing tools for generating Scalable Vector Graphics (SVG). Since SVGs are an image format that is natively supported in the browser, we can expand their functionality with a myriad of open-source libraries. For the sake of this project, I only used utilities provided by D3.

3.1 Genre Histogram

To begin, I felt it was important to get an idea of what kind of data I was using. Since I was planning on working with genre data, I decided to create a histogram of the various genres. Needless to say, the results were skewed toward my favorite genres as seen in Fig. 2.

The visualization in Fig. 2 was rendered from Table 3 which contains 17 distinct genres including a placeholder for songs that have no documented genre. For convenience, the table was sorted in the same order as the plot. As a result, it's clear which genres dominate the collection: alternative and rock.

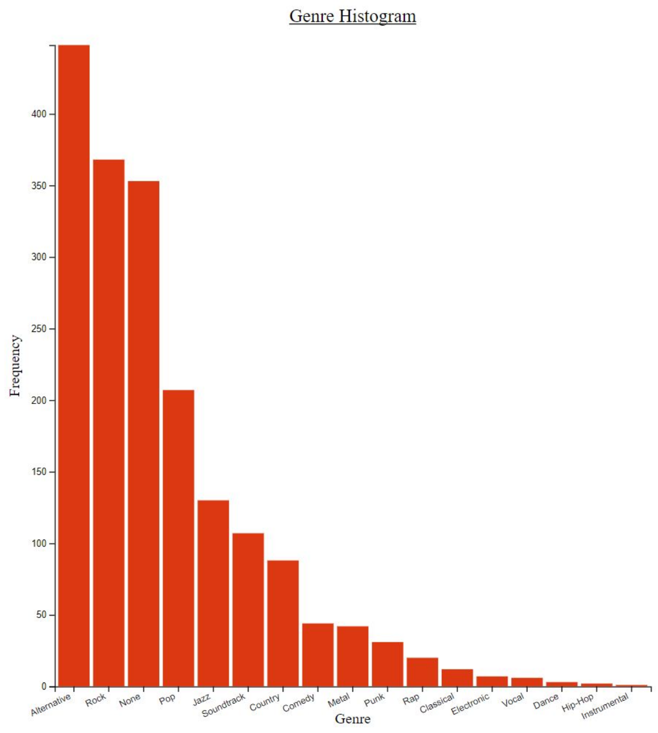

Fig. 2. A genre histogram from [5] of all the files in the dataset.

To encode genres in later plots, I chose to use color. Unfortunately, D3 no longer supports color pallets larger than 12 distinct colors, so I was forced to filter the data set by the 12 most frequent genres in the dataset. As a result, there are about 12 data points missing from Fig. 3 and Fig. 4. In particular, the vocal, dance, hip-hop, and instrumental genres have not been included. Instead, the contain only the genres contained in Table 4.

Since songs lacking genre data provide no useful insight in a genre comparative study, I chose to exclude all files with None tags for genre. In total, I had about 1,500 music files for visualization purposes.

3.2 Duration Versus Release Date

With a proper head count of genres and files, I proceeded to render a visualization of duration by release date which can be seen in Fig. 3. After all, I was interested to see if song durations have changed over time.

Right away, it's clear which genres tend to have the longest durations. For example, Metal and Classical music look like outliers with their durations almost three times as long as what appears to be the average duration. I also find it interesting that duration appears to be decreasing over time. That said, I suppose that makes sense. Three-minute pop songs are a lot more common today than 8-minute jazz songs akin to Maynard Ferguson's rendition of Hey Jude.

In addition, I'm interested in seeing how the genres cluster. Clearly, some genres are more popular today than in the past, and many of them seem to follow the same formula in terms of duration. Apparently, consumers have chosen some sort of duration sweet spot which hovers around 200 seconds.

3.3 dBFS Versus Release Date

Since I was on the subject of data over time, I thought it would be interesting to look at loudness over time using decibels at full scale (dBFS) and release date. The ideas here being that songs may have changed in average loudness over time. Naturally, the dBFS versus release data plot can be seen in Fig. 4.

After looking at Fig. 4, it's tough to make a conclusion on any trends in loudness over time. However, there are a few key features to notice.

Table 3. Genre Totals

Genre	Count
Alternative	448
Rock	368
None	353
Pop	207
Jazz	130
Soundtrack	107
Country	88
Comedy	44
Metal	42
Punk	31
Rap	20
Classical	12
Electronic	7
Vocal	6
Dance	3
Hip-Hop	2
Instrumental	1
Sum	1869

Table 4. Filtered Genre Totals

Genre	Count
Alternative	448
Rock	368
Pop	207
Jazz	130
Soundtrack	107
Country	88
Comedy	44
Metal	42
Punk	31
Rap	20
Classical	12
Electronic	7
Sum	1504

For one, loudness appears to be trending up slightly. However, the loudness is definitely more spread now than it used to be. In general, it seems songs used to hover between -12 and -22 dBFS. Now, songs range anywhere from -8 to -26 dBFS.

In terms of genre, there are a lot of interesting trends. For example, Rock is very clearly trending up in loudness. In the 70s, rock could be seen between -14 and -24 dBFS. Now, rock can be seen at -14 in its quietest. Likewise, genres like alternative and metal appear to be relatively loud on average. Meanwhile, the soundtrack genre tends to be very quiet.

3.4 Average dBFS Versus Genre

Having seen an overview of dBFS over time, I had to take the analysis one step further by asking: what genres are loudest on average? To do that, I ran an analysis which captured the average dBFS of each genre and plotted on a bar graph (Fig. 5) similar to the genre histogram.

In Fig. 5, I noticed right away that dance is the loudest genre by a considerable margin, and I suppose that makes sense. If the target audience of dance music is the club, it make sense for dance songs to be quite loud. However, it's important to notice that dance has only 3 samples, so take that graph with a grain of salt.

Closely behind dance is punk and hip-hop. Again, hip-hop only has two samples representing that average, so that genre can be safely ignored. Meanwhile, punk has around 31 samples which may all be from one artist, so it's tough to really come to any conclusions in that regard.

That said, from a biased point-of-view, I think this curve is fairly accurate. I would expect rougher genres like metal, punk, and dance to be louder on average than genres that have more ebb and flow like jazz and instrumental. Also, I'd expect genres with less simultaneous sounds like vocal to be quiter on average.

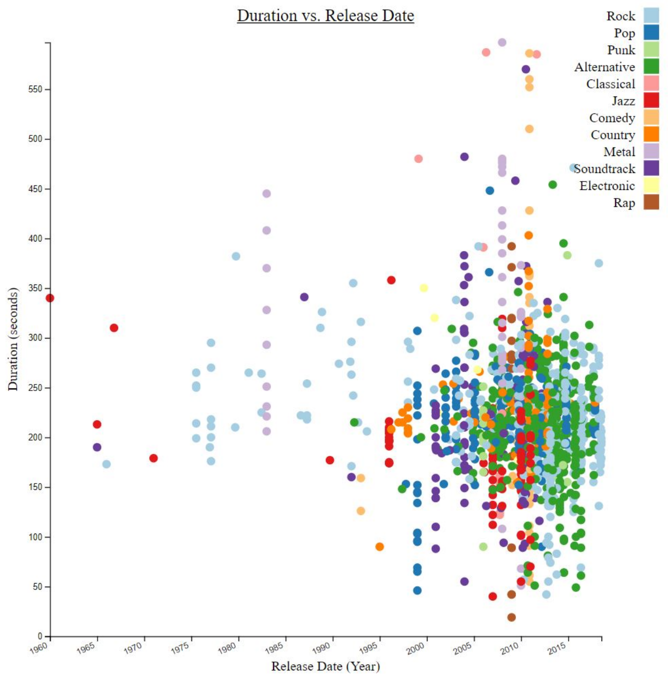

Fig. 3. A plot of duration versus release date from [4] where colors correspond to genres.

Perhaps most surprisingly, classical comes in at the quietest genre from its twelve sample average. My experience with live classical music is that it can be very quiet at times and very loud at other times. Perhaps that contrast is skewed toward the lower end for more effect. I would be interested in digging into this more in the future.

4 Challenges

As it turns out, the scope of the initial proposal was much too broad. Many of the signal processing approaches proposed were outside of my capabilities within a semester. Given more time, I would have liked to explore audio features beyond loudness such as pitch and onset.

In addition, it turns out that reading and parsing existing audio formats is really challenging. Not only are the formats themselves not well documented, but the files are general fairly large. As a result, processing can take a lot of time and resources. Retrieving data for visualization took many hours of periodic persistence to ensure nothing was lost.

Finally, overall software design was challenging. From parsing and data collection to data aggregation and visualization, there were a lot of steps in the pipeline. In particular, I ended up leveraging two separate scripting languages, Python and JavaScript, as well as an intermediate data respresentation, JSON. In total, I have written over 1600 lines of code for this project not including the report.

5 Conclusion

After exploring M4A file parsing, audio signal processing, and genre data visualization, I have concluded that analyzing music is a very difficult task. While I had plenty of data, parsing common file formats, analyzing audio levels, and generating compelling visualizations are not easy tasks.

That said, I learned a lot about the relationships between different genres of music. For example, it's hardly surprising that certain genres have longer songs on average (i.e. Jazz, Classical, and Metal). However, I was suprised to see that songs have gotten slightly shorter over time.

In the future, I'd like to explore music genres with an even larger data set as to avoid introducing any personal biases. In addition, I would

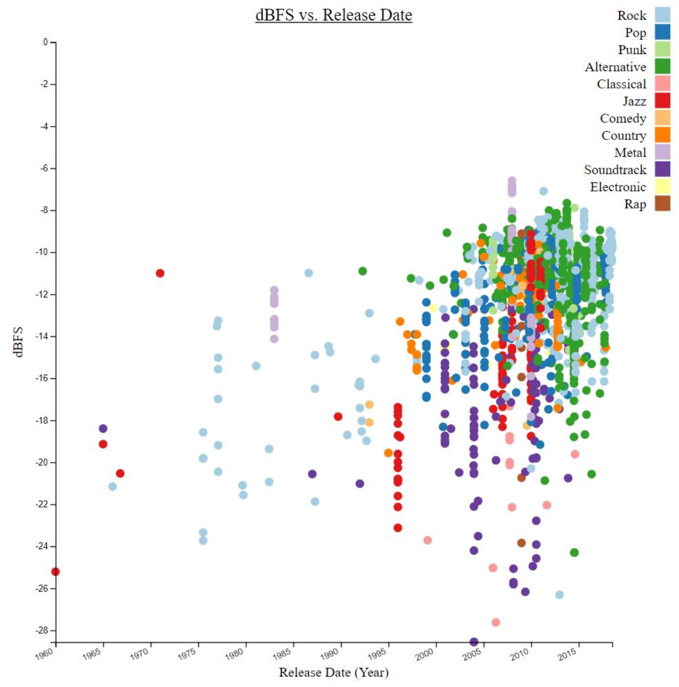

Fig. 4. A dBFS versus release date from [2] where colors correspond to genres.

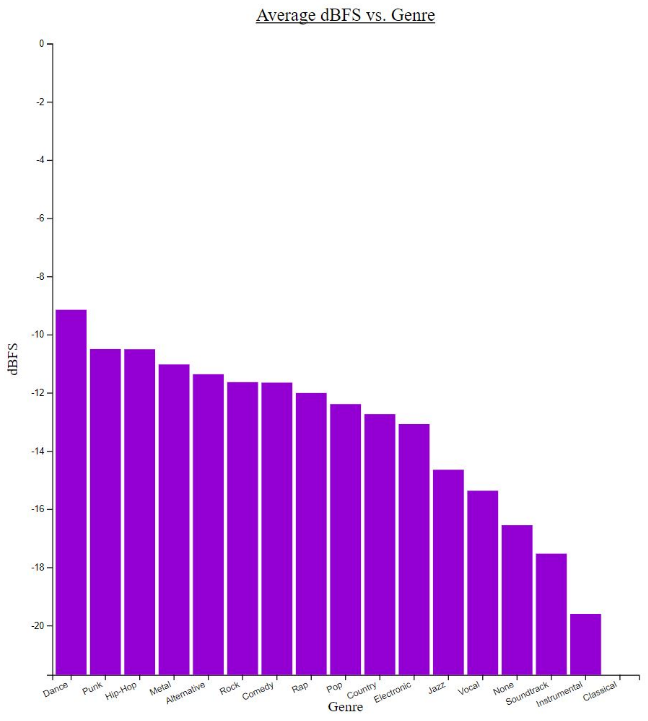

Fig. 5. A average dBFS histogram from [3] of all the files in the dataset.

like to expand the visualization tool to include interactivity elements like filtering. It would be nice to be able to hide or gray out some of the points in the scatter plots. Also, I would like to be able to hover over each point to see meta information like song title.

References

[1] Apple. Movie Atoms. https://developer.apple.com/library/archive/documentation/QuickTime/QTFF/QTFFChap2/qtff2.html, 2016 (accessed 09-April-2019).

[2] J. Grifski. dBFS Vs. Release Date Plot - CSE 5539. https://jrg94.github.io/CSE5539/term-project/musiviz/dbfs-vs-release-date.html, 2019 (accessed 09-April-2019).

[3] J. Grifski. dBFS Vs. Release Date Plot - CSE 5539. https://jrg94.github.io/CSE5539/term-project/musiviz/average-dbfs-vs-genre.html, 2019 (accessed 09-April-2019).

[4] J. Grifski. Duration Vs. Release Date Plot - CSE 5539. https://jrg94.github.io/CSE5539/term-project/musiviz/duration-vs-release-date.html, 2019 (accessed 09-April-2019).

[5] J. Grifski. Genre Histogram - CSE 5539. https://jrg94.github.io/CSE5539/term-project/musiviz/genre-histogram.html, 2019 (accessed 09-April-2019).

[6] J. Grifski. MusicFile. https://github.com/jrg94/CSE5539/blob/master/term-project/musiparse/music_file.py, 2019 (accessed 09-April-2019).

[7] J. Grifski. MusicFileSet. https://github.com/jrg94/CSE5539/blob/master/term-project/musiparse/music_file_set.py, 2019 (accessed 09-April-2019).

[8] J. Price. Understanding dB. http://www.jimprice.com/prosound/db.htm, 2007 (accessed 09-April-2019).

[9] J. Robert, V. Srinivasan, and T. Fiers. API Documentation. https://github.com/jiaaro/pydub/blob/master/API.markdown, 2018 (accessed 09-April-2019).

[10] E. Weisstein. Root-Mean-Square. http://mathworld.wolfram.com/Root-Mean-Square.html, 2011 (accessed 09-April-2019).