Comparing Song Audio Features to Rankings on The Billboard Hot 100

Kevin Carr 501150122
Supervisor: Ceni Babaoglu, PhD
Date: December 3, 2022

Ryerson University

Table of Contents

Abstract

The central problem in this project is to utilise publicly available audio feature data to predict whether or not a song is likely to appear on the Billboard Hot 100 charts. Music streaming services employ data models to characterise audio features for songs. This data is provided and publicly available for multiple streaming services, notably Spotify. The Billboard Hot 100 has been a music industry standard for approximately 70 years. The Billboard Hot 100 contains weekly rankings for songs which are based on sales, plays, and surveys.

Audio feature data and Billboard Hot 100 chart data for this project have been obtained from multiple sources and combined. Billboard Hot 100 charts were obtained for the entire history of the Billboard Hot 100, from 1958 to 2021. Two additional databases were obtained consisting of approximately 10 million songs with audio feature data. Were possible, this audio feature data was merged with songs from the Billboard Hot 100 charts. In cases where audio feature data for Billboard Hot 100 songs were not in available datasets, missing audio features were obtained using the Spotify API, where available. Audio features were obtained for approximately 75% of songs in the Billboard Hot 100 charts, as well as an approximately 10 million additional songs.

For this project, two main techniques have been employed, namely data mining and predictive analytics. First, data mining and knowledge discovery was used to explore the data, cluster audio features, and determine correlations between audio features. Second, predictive analytics was used to attempt to build a predictive model using the data. By utilising knowledge discovered during the data mining phase of the project, predictive analysis has been broken into sets of clustered songs with similar audio features and/or genres.

After employing a number of predictive machine learning models on clustered and un-clustered data, classification success was mixed. Since the dataset is highly unbalanced, achieving high accuracy was trivial. However, optimising for more nuanced metrics such as precision, recall, or F1-score has proven difficult. Based on visual inspection and statistical analysis, the predictions appear to be correct, albeit not as useful anticipated. This is due to the fact that a large number of songs exist with audio features consistent with hit songs, but only a small portion of all songs become hits. Therefore, popular audio feature characteristics predicted using these methods may be necessary for a song to achieve popularity but are not sufficient for commercial success. More research is necessary to confirm this hypothesis.

Introduction

Music streaming services employ data models to characterise audio features for songs, and use this data to recommend songs and playlist to their listeners. This data is provided and publicly available for multiple streaming services, notably Spotify (Spotify, n.d.).

The Billboard Hot 100 has been a music industry standard for approximately 70 years (Wikipedia, 2022). The Billboard Hot 100 contains weekly rankings for songs which are based on sales, plays, and surveys.

For this project, data mining and predictive analytics have been employed. Data mining and knowledge discovery were used to explore the data, cluster audio features, and determine correlations between audio features. Predictive analytics was used to attempt to build a predictive model.

This analysis has the potential to predict future trends in music or the performance of an individual song. These predictions could be useful to musicians or producers attempting to optimise success, or listeners looking for something new.

Literature Review

The primary focus of this literature review was to gather understanding on previous research related to music clustering techniques and the prediction of popularity for songs, especially in cases where song audio features were used for clustering or prediction. Google Scholar (Google Scholar, n.d.), PaperDigest.org (Paper Digest, n.d.), and Elicit.org (Elicit, n.d.) were used to investigate and gather research materials.

Popularity can be defined in numerous ways. In this study, popularity is simply considered to be an appearance on the Billboard Hot 100 charts. The magnitude of this popularity may be further be defined using total weeks on the chart, or top rank on the chart (Lee et. al., 2018). Using appearance on the Billboard Hot 100 as a metric for popularity has been used in other similar studies (Reiman et.al., 2018). Another commonly used popularity metric is 'popularity' as defined in the Spotify API (Kim, 2021; Gao, 2021). Since the Spotify 'popularity' metric does not include historical popularity data and considers only Spotify streaming, this study focusses on the Billboard Hot 100 charts in order to get a longer-term, and wider perspective of music popularity.

Audio features used in music classification have evolved through various stages. MIDI (Musical Instrument Digital Interface) format musical notation has been used to cluster music into categories (Cilibrasi et. al., 2004; Cataltepe et. al., 2007). Low-level audio features such as mel-

frequency cepstral coefficients, spectral flatness, and number of zero crossings have been used to predict steaming popularity (Yang et. al., 2017; Lee et. al., 2018; Araujo et. al., 2019), improve music recommendation systems (Li et. al., 2007; Schedl, 2013), and to classify emotion in music (Jia, 2022). High-level audio features such as danceability, instrumentalness, and speechiness are included in track information available from the Spotify API. These high-level audio features have been used to identify song attributes (Febirautami et. al., 2018), predict popularity (Reiman et.al., 2018; Martín-Gutiérrez et. al., 2020; Kim, 2021; Gao, 2021), to classify music into genres (Setiadi et. al., 2020), and to classify music into moods (Chen et. al., 2021).

Clustering music has been used for recommender systems (Li et. al., 2004; Li et. al., 2007; Huo, 2021), as well as to categorise music (Honingh et. al., 2011). In this study, overall trends, genres, and audio feature clusters have been considered to attempt to improve predictive analytics. It was hypothesised that the accuracy of predictions of song popularity using audio features could be improved by separating predictions by genre (Reiman et.al., 2018). Music genres have been used in combination with high-level audio features to predict popularity (Kim, 2021). It has been noted that audio features within the Billboard Hot 100 within genres are relatively consistent over time (O'Toole et. al., 2022).

In this study, clustering and classification have been used. Similar studies have had success with a variety of techniques and models. Neural networks have been used to predict popularity (Yang et. al., 2017; Gao, 2021), improve music recommendation systems (Li, 2021; Shi, 2021), or classify music (Jia, 2022; Li et. al., 2022). K-Means Clustering has been used in a number of studies to cluster data (Li et. al., 2007; Xu et. al., 2021; Kim et. al., 2021). A variety of classification models have been used to classify music, notably Support Vector Machines

(Laurier et. al., 2009; Lee et. al., 2018; Reiman et.al., 2018; Araujo et. al., 2019; Setiadi et. al., 2020; Wilkes et. al., 2021), K-Nearest Neighbours (Cataltepe et. al., 2007; Reiman et.al., 2018; Kim, 2021), Decision Trees / Random Forests / Boosted Trees (West, 2008; Febirautami et. al., 2018; Chen et. al., 2021; Gao, 2021), and Logistic Regression (Reiman et.al., 2018; Chen et. al., 2021; Gao, 2021). In addition, Principle Component Analysis has been used to reduce dimensionality and improve predictive results (Gao, 2021).

Predicting popularity on the Billboard Hot 100 has been investigated in other studies (Lee et. al., 2018; Reiman et.al., 2018). However, no study has successfully used high-level audio features to predict popularity as defined above. Although high-level audio features were used, Reiman et.al. were not able to accurately predict song popularity. This was potentially due to an overly diverse dataset for non-hit songs. It was hypothesised that the accuracy of predictions of song popularity using audio features could be improved by separating predictions by genre (Reiman et.al., 2018). Additionally, although it has been demonstrated that popularity can be predicted using audio features alone, this was demonstrated using low-level audio features and different statistical descriptions for popularity (Lee et. al., 2018). This study aims to accurately predict song popularity, defined as appearance on the Billboard Hot 100 charts, using high-level audio features and genre data available from the Spotify API. In addition, none of the studies listed above utilised highly-unbalanced data to predict popularity, even though popularity, by definition is highly-unbalanced. This study aims to make predictions using a large and highly-unbalanced dataset.

Data Description

Data Sources

The data gathered for this project have been taken from multiple sources and combined. Data was found using the Google dataset search engine (Google Dataset Search, n.d.).

Three of the relevant sources were found on Kaggle.com, a popular online data science community where users can share datasets (Dhruvil Dave, 2021; Malte Grosse, 2022; Rodolfo Figueroa, 2020). Audio features from the large datasets were matched with the list of songs from the Billboard Hot 100. Missing data were obtained, where available, from the Spotify API.

Data Pre-Processing and Organisation

Data Importing and preprocessing was completed in 3 stages:

1. Import and setup data types
2. Get missing data from Spotify API
3. Merge, clean, and save optimised datasets

Data import and cleaning is included as **Attachment 1**.

The "Billboard 'The Hot 100' Songs" dataset (Dhruvil Dave, 2021) was available in CSV format, and includes date, rank, song title, artist, last-week, peak-rank, and weeks-on-board. This CSV was imported into Python as a Pandas dataframe. This data did not include Spotify song ids, genres, release dates, or audio features, so the Spotify API was used to gather this data.

The SQLite dataset, "8+ M. Spotify Tracks, Genre, Audio Features" (Malte Grosse, 2022) included 9 tables totalling 44 columns. The database was queried to combine song title, artist, Spotify id, release date, and audio features. The queried data were exported to CSV and imported into Python as a Pandas dataframe.

The "Spotify 1.2M+ Songs" dataset (Rodolfo Figueroa, 2020) was available in CSV format. It included combine song title, artist, Spotify id, release date, and audio features. This CSV was imported into Python as a Pandas dataframe.

Based on findings from the literature review portion of this study, genre data was gathered for easily available songs. Since genre data was included in the SQLite database for many of the songs (Malte Grosse, 2022), this data was queried, exported as CSV, and imported into Python. Since multiple genres were often available for a given artist, all genres were populated, then the results were sorted by most common genre. This results in songs being categorised with only one genre. It should be noted that this is the most common genre, and not necessarily the most applicable genre. It should also be noted that not all songs have genre data associated with them. Approximately 69% of songs included genre data (approximately 6.6M entries).

Songs from the Billboard Hot 100 were queried by artist and song name using the Spotify API, gathering Spotify id were available. Songs with Spotify id were queried to gather audio features, genre data, and release dates. Since multiple genres were often available, all genres were obtained from the API, then the result corresponding to the most common genres from the SQLite query data were populated as the song's genre. Similar to above, it should be noted that this is the most common genre, and not necessarily the most applicable genre, and not all songs have genre data associated with them.

Since Get Requests from the Spotify API were time consuming, undefined genres from the "Spotify 1.2M+ Songs" dataset (Rodolfo Figueroa, 2020) were left undefined due to time constraints.

The data from each dataset was combined to form 3 non-distinct working datasets:

- All songs including audio features (approximately 10M entries)
- The Billboard Hot 100 historical charts (approximately 300k entries)
- Songs from The Billboard Hot 100 that include audio features and genre (approximately 20k entries)

Since the dataframes are not distinct, in cases where mutually exclusive groups were necessary, Pandas dataframe query functions and vectorized formula may be implemented to segment data as required.

Once working datasets were compiled, they were exported as Pickle format. Pickle has a number of advantages over CSV format, most notably file size, the retention of data type formats, and the amount of time required to re-load the file into Python Pandas dataframe.

File and Calculation Locations

Datasets and calculations used in this study can be found at the following URL:

https://github.com/KevinCarr42/Billboard-100-Audio-Feature-Analytics

Files too large to upload to GitHub been uploaded to a shared Google Drive folder (shared with Toronto Metropolitan University Google Drive accounts):

https://drive.google.com/drive/folders/10wpORzZURV11VAUPKmCDDKxHFCvwjloR

Descriptive Statistics

Descriptive statistics for the datasets are included in the following tables. More detailed descriptive calculations are included in **Attachment 2**.

Table 1. Descriptive Statistics - All Songs With Audio Features

Audio Feature	Mean	Standard Deviation	Minimum	Quartiles			Maximum
				25%	50%	75%	
acousticness	0.42	0.37	0.00	0.03	0.34	0.82	1.00
danceability	0.53	0.19	0.00	0.40	0.54	0.68	1.00
duration_ms	238,312	156,918	1,000	169,587	216,933	275,400	6,072,187
energy	0.55	0.28	0.00	0.31	0.57	0.79	1.00
instrumentalness	0.26	0.38	0.00	0.00	0.00	0.66	1.00
key	5.2	3.5	0	2	5	8	11
liveness	0.21	0.18	0.00	0.10	0.13	0.26	1.00
loudness	-11.0	6.3	-60.0	-13.7	-9.2	-6.4	7.2
mode	0.66	0.47	0.00	0.00	1.00	1.00	1.00
speechiness	0.10	0.14	0.00	0.04	0.05	0.08	0.97
tempo	118.6	30.9	0	95	119	137	250
time_signature	3.84	0.57	0.00	4.00	4.00	4.00	5.00
valence	0.47	0.28	0.00	0.23	0.47	0.71	1.00

Table 2. Descriptive Statistics - Billboard Hot 100

Audio Feature	Mean	Standard Deviation	Minimum	Quartiles			Maximum
				25%	50%	75%	
rank	50.5	28.9	1	26	51	76	100
last-week	47.6	28.1	1	23	47	72	100
peak-rank	41.0	29.3	1	13	38	65	100
weeks-on-board	9.2	7.6	1	4	7	13	90
acousticness	0.28	0.28	0.00	0.04	0.18	0.47	1.00
danceability	0.60	0.15	0.00	0.51	0.61	0.71	0.99
duration_ms	226,926	65,973	37,013	183,560	221,400	258,533	1,292,293
energy	0.63	0.20	0.02	0.48	0.64	0.79	1.00
instrumentalness	0.03	0.13	0.00	0.00	0.00	0.00	0.99
key	5.2	3.6	0	2	5	8	11
liveness	0.19	0.16	0.02	0.09	0.13	0.24	1.00
loudness	-8.6	3.6	-29.5	-11.0	-8.1	-5.8	2.3
mode	0.73	0.44	0.00	0.00	1.00	1.00	1.00
speechiness	0.06	0.07	0.00	0.03	0.04	0.06	0.94
tempo	120.4	27.9	0	100	119	136	241
time_signature	3.94	0.29	0.00	4.00	4.00	4.00	5.00
valence	0.61	0.24	0.00	0.42	0.63	0.81	0.99

Table 3. Descriptive Statistics - All Songs From Billboard Hot 100 With Audio Features

Audio Feature	Mean	Standard Deviation	Minimum	Quartiles			Maximum
				25%	50%	75%	
acousticness	0.32	0.29	0.00	0.05	0.22	0.56	1.00
danceability	0.59	0.15	0.00	0.49	0.60	0.70	0.99
duration_ms	217,614	67,768	37,013	169,707	210,533	251,260	1,292,293
energy	0.61	0.20	0.02	0.46	0.63	0.78	1.00
instrumentalness	0.04	0.14	0.00	0.00	0.00	0.00	0.99
key	5.2	3.6	0	2	5	8	11
liveness	0.19	0.16	0.02	0.09	0.13	0.25	1.00
loudness	-8.9	3.6	-29.5	-11.3	-8.5	-6.1	2.3
mode	0.74	0.44	0.00	0.00	1.00	1.00	1.00
speechiness	0.07	0.08	0.00	0.03	0.04	0.06	0.94
tempo	120.5	28.2	0	100	119	137	241
time_signature	3.93	0.33	0.00	4.00	4.00	4.00	5.00
valence	0.61	0.24	0.00	0.42	0.64	0.81	0.99

Table 4. Descriptive Statistics - Songs From Billboard Hot 100 With Audio Features And Genre

Audio Feature	Mean	Standard Deviation	Minimum	Quartiles			Maximum
				25%	50%	75%	
acousticness	0.31	0.29	0.00	0.05	0.22	0.54	1.00
danceability	0.59	0.15	0.00	0.49	0.60	0.70	0.99
duration_ms	219,137	67,456	37,013	171,881	212,027	252,237	1,292,293
energy	0.61	0.20	0.02	0.47	0.63	0.78	1.00
instrumentalness	0.03	0.14	0.00	0.00	0.00	0.00	0.99
key	5.2	3.6	0	2	5	8	11
liveness	0.19	0.16	0.02	0.09	0.13	0.25	1.00
loudness	-8.8	3.6	-29.0	-11.3	-8.4	-6.0	-0.4
mode	0.74	0.44	0.00	0.00	1.00	1.00	1.00
speechiness	0.07	0.08	0.00	0.03	0.04	0.06	0.94
tempo	120.6	28.2	0	100	119	137	231
time_signature	3.93	0.33	0.00	4.00	4.00	4.00	5.00
valence	0.61	0.24	0.00	0.42	0.63	0.81	0.99

Audio Feature Descriptions

Audio features available from the Spotify API used in this study are described in detail in the below table (Spotify, n.d.).

Table 5. Description of Audio Features From Spotify API

Audio Feature	Type	Description	Min	Max
acousticness	number <float>	A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.	0	1
danceability	number <float>	Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.	0	1
duration_ms	integer	The duration of the track in milliseconds.	0	N/A
energy	number <float>	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.	0	1
instrumentalness	number <float>	Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.	0	1
key	integer	The key the track is in. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1.	-1	11
liveness	number <float>	Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.	0	1

Audio Feature	Type	Description	Min	Max
loudness	number <float>	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0 db.	-60	0
mode	integer	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.	0	1
speechiness	number <float>	Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.	0	1
tempo	number <float>	The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.	0	N/A
time_signature	integer	An estimated time signature. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure). The time signature ranges from 3 to 7 indicating time signatures of "3/4", to "7/4".	3	7
valence	number <float>	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).	0	1

Histograms

Histograms for each of the audio features are shown in the below figures, comparing songs on the Billboard Hot 100 with all songs in this study.

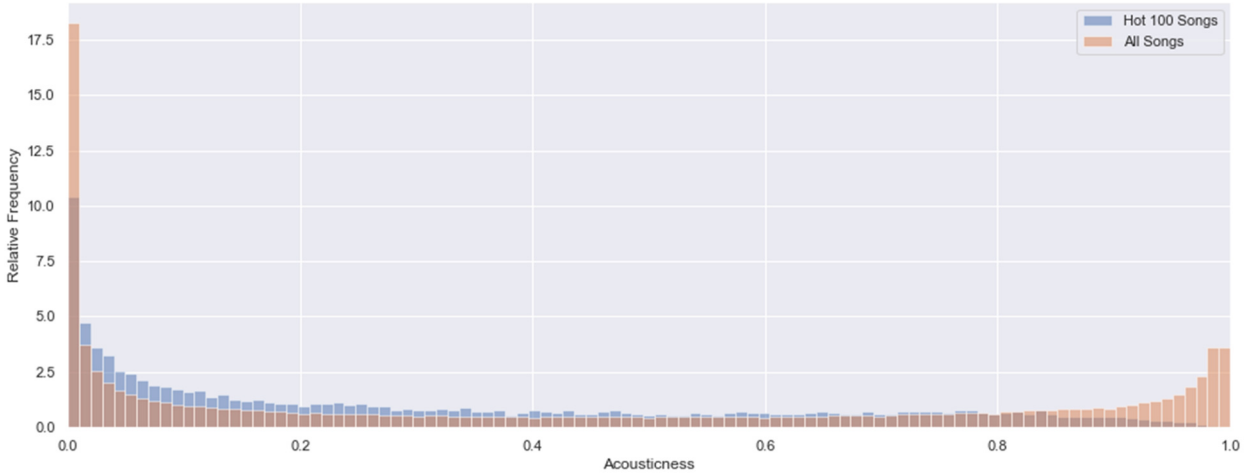


Figure 1. Acousticness Histogram

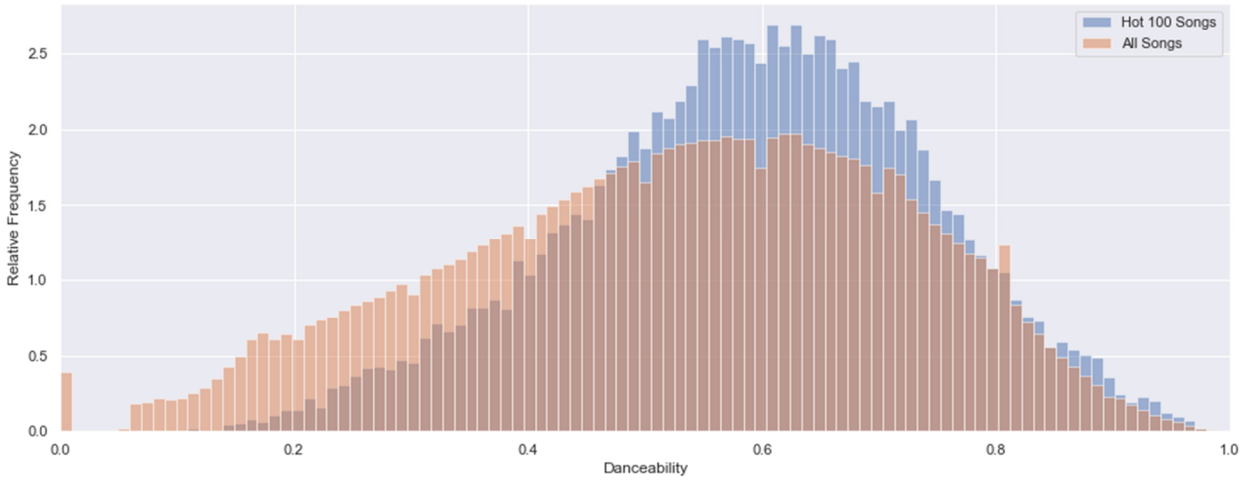


Figure 2. Danceability Histogram

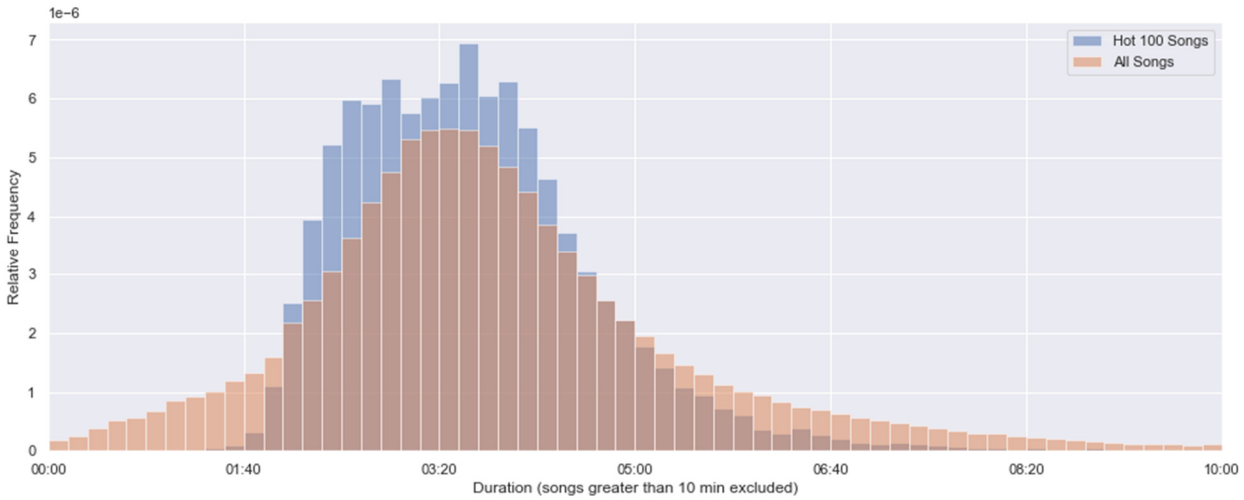


Figure 3. Duration Histogram

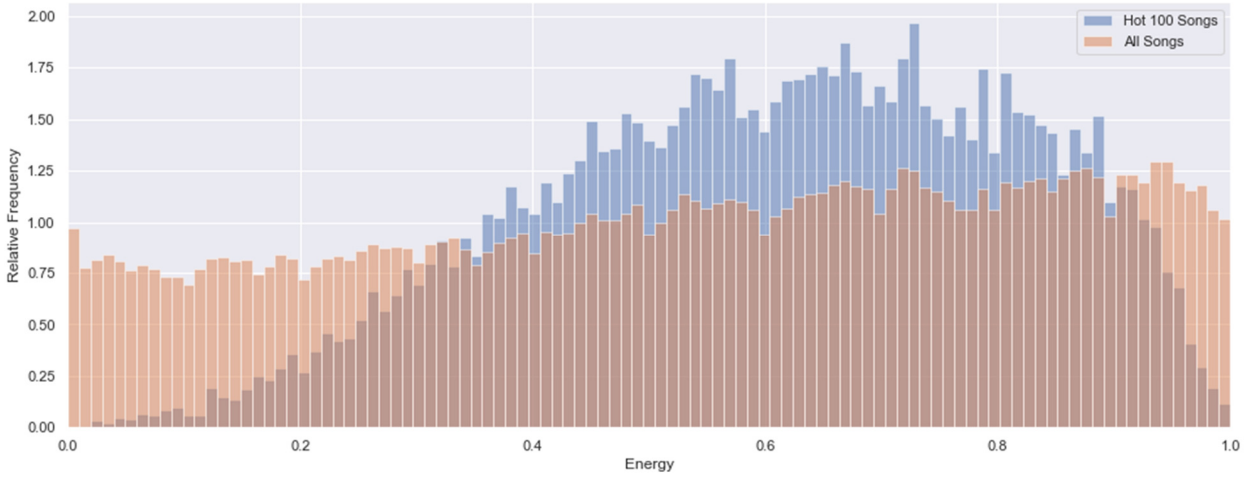


Figure 4. Energy Histogram

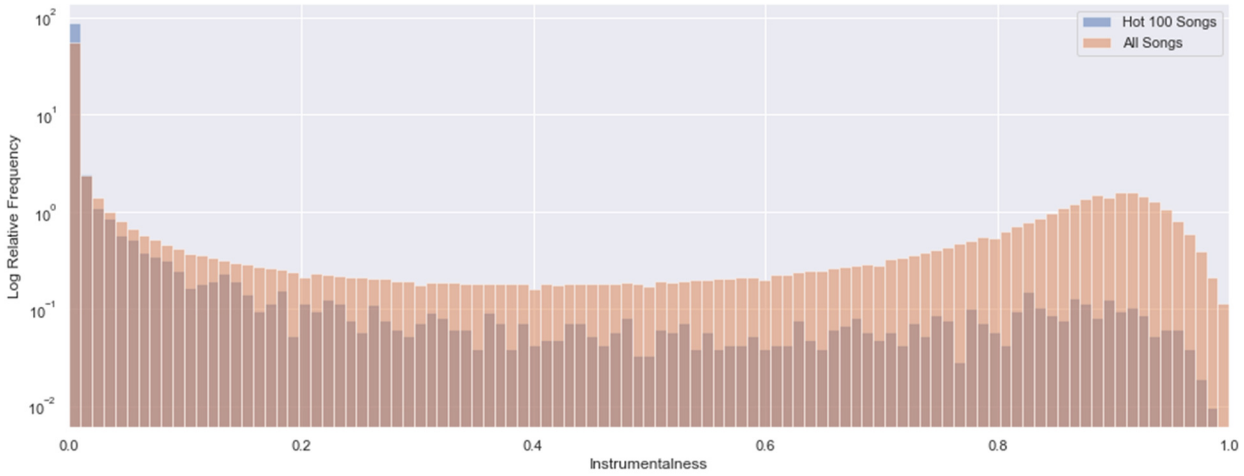


Figure 5. Instrumentalness Histogram

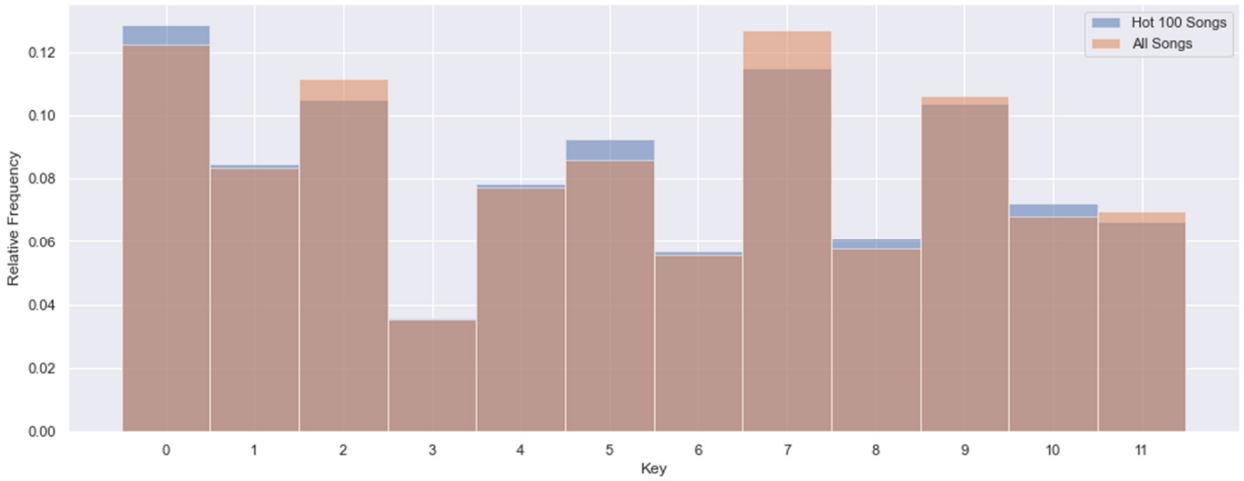


Figure 6. Key Histogram

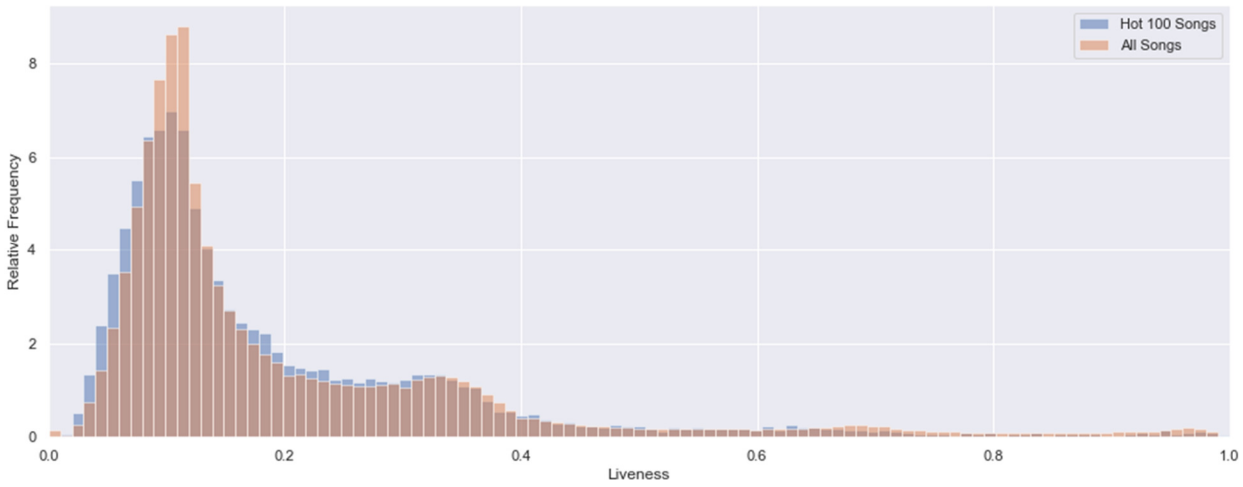


Figure 7. Liveness Histogram

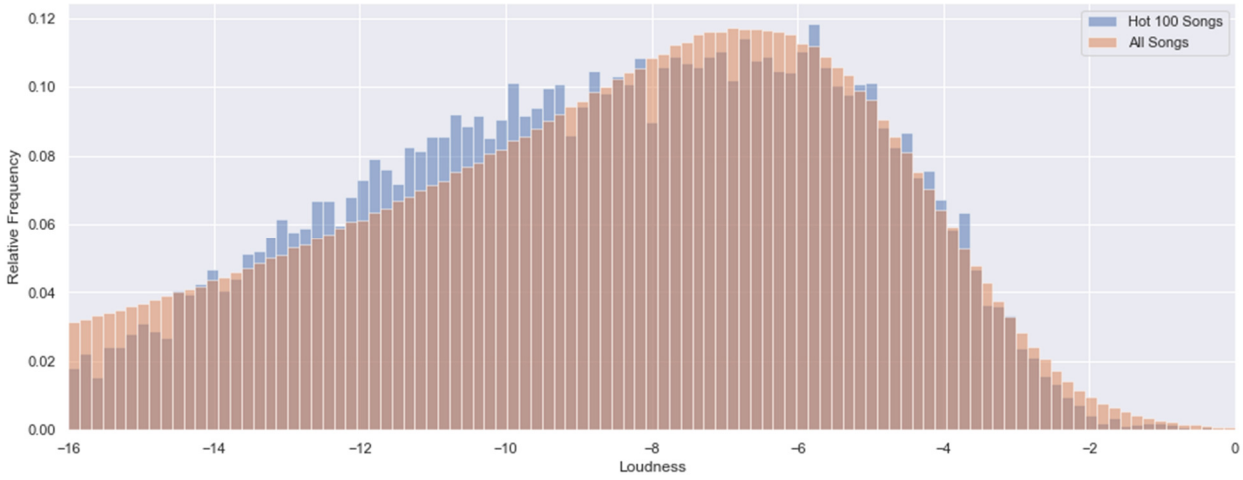


Figure 8. Loudness Histogram

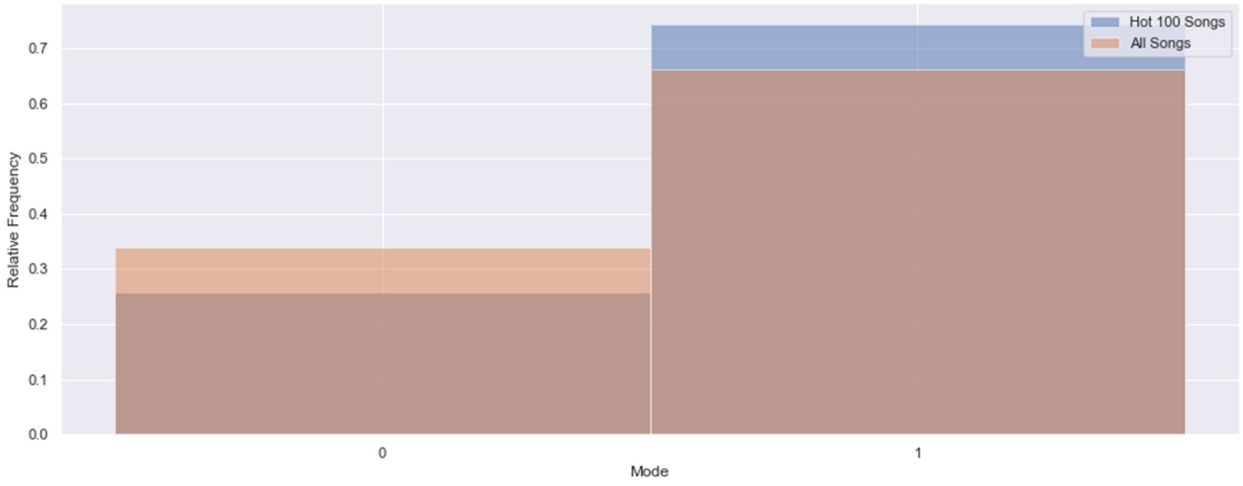


Figure 9. Mode Histogram

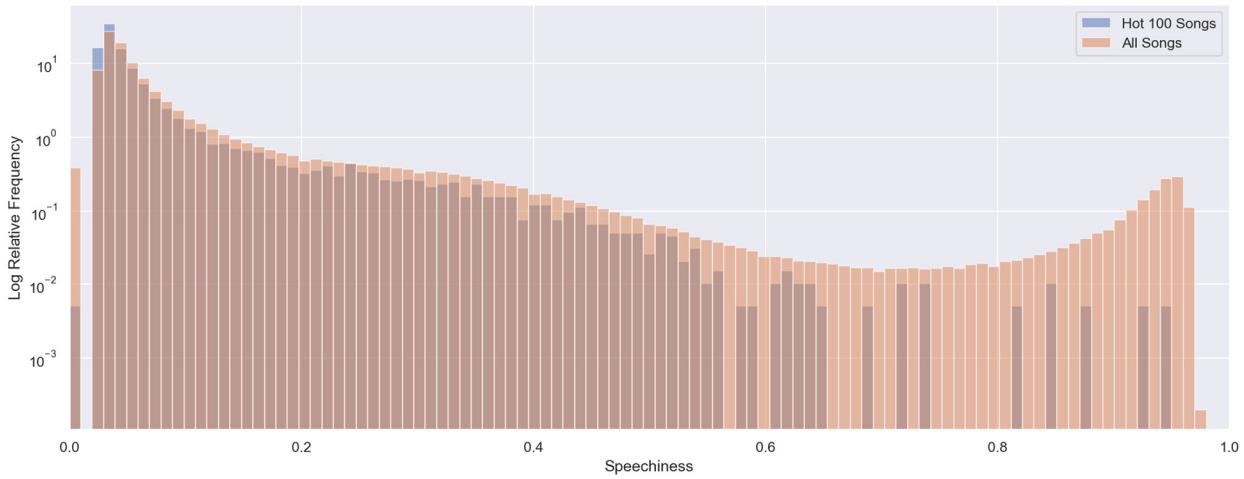


Figure 10. Speechiness Histogram

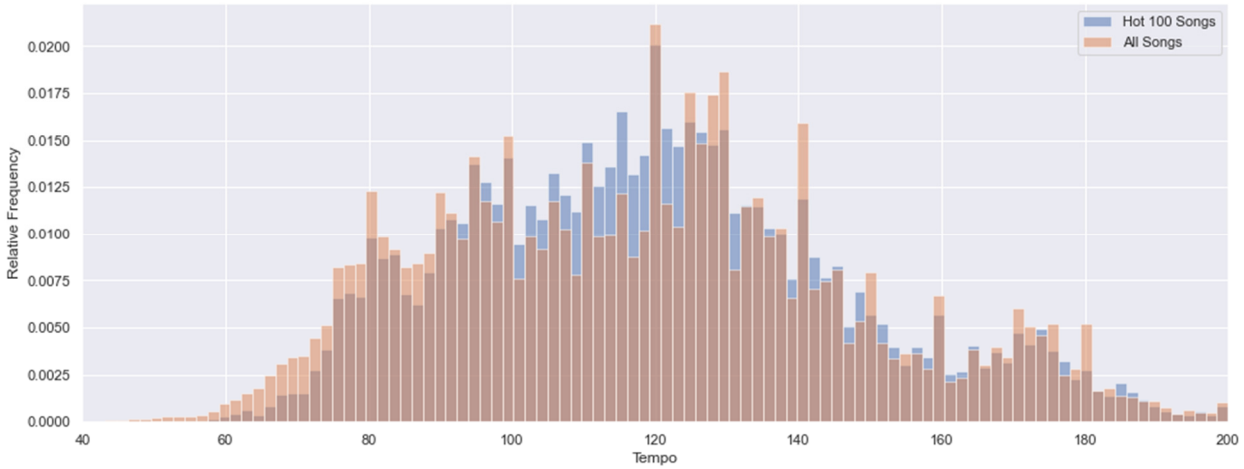


Figure 11. Tempo Histogram

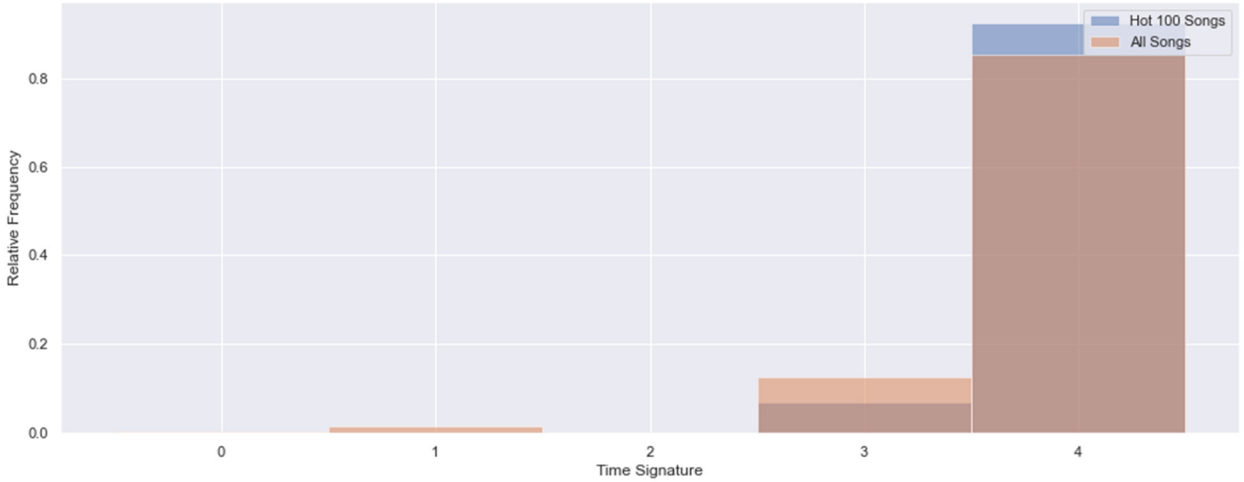


Figure 12. Time Signature Histogram

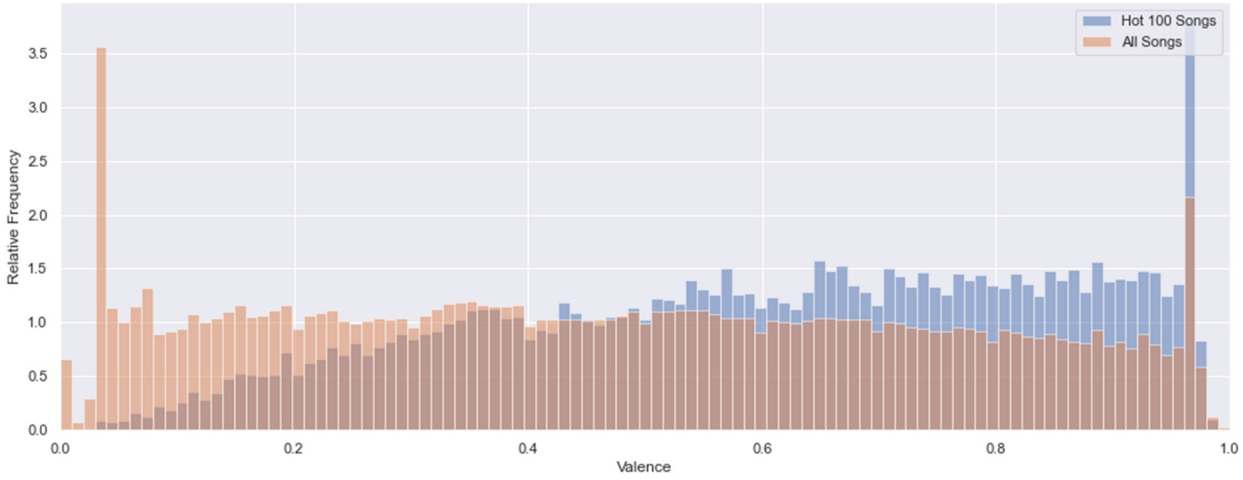


Figure 13. Valence Histogram

Historical Changes in Audio Features

Historical line plots for each of the audio features are shown in the below figures, comparing audio feature averaged by release year for songs on the Billboard Hot 100 compared with all songs in this study.

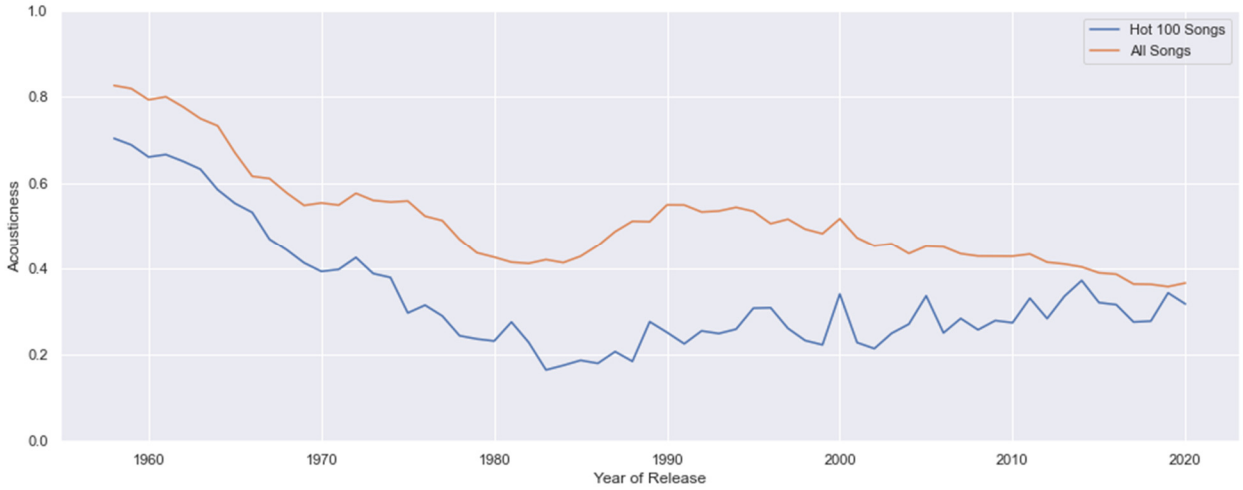


Figure 14. Acousticness History Comparing the Billboard Hot 100 with All Songs

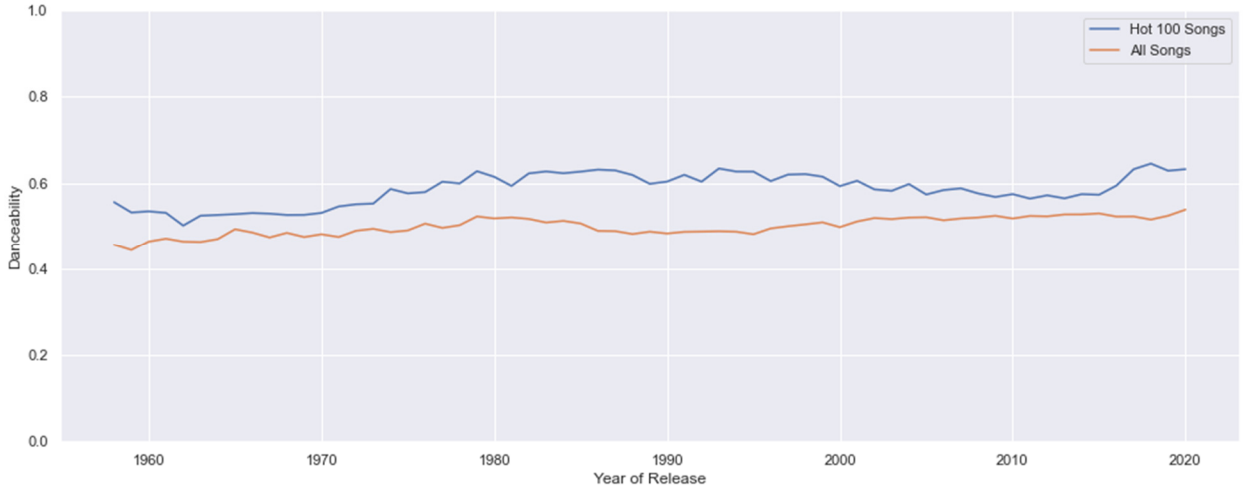


Figure 15. Danceability History Comparing the Billboard Hot 100 with All Songs

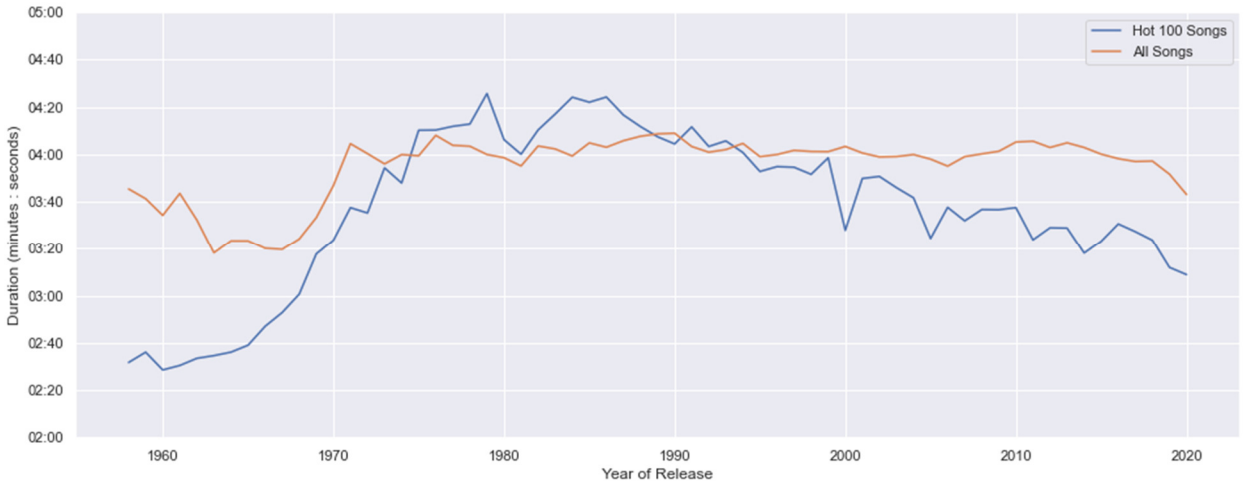


Figure 16. Duration History Comparing the Billboard Hot 100 with All Songs

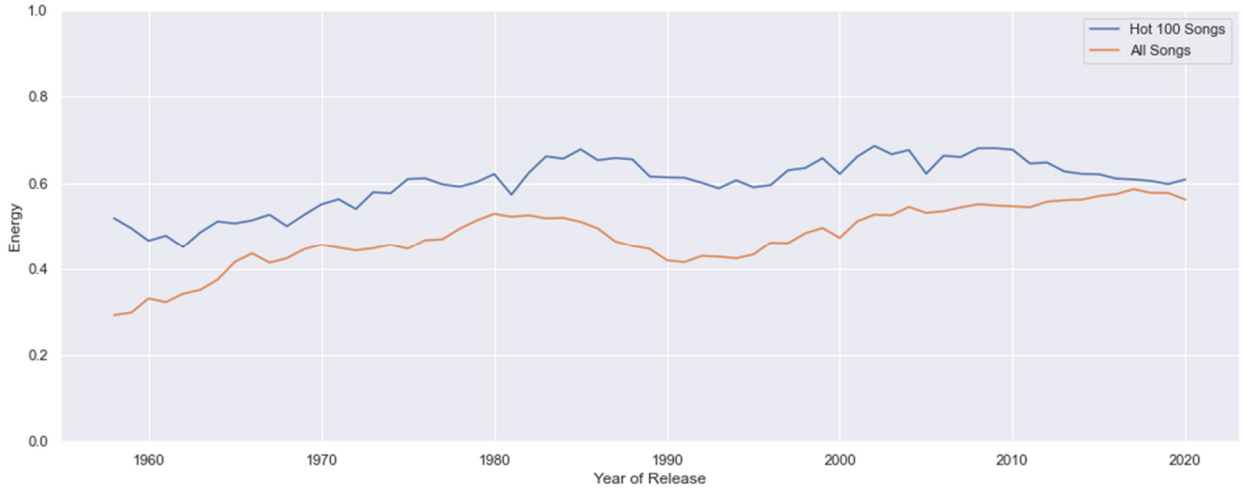


Figure 17. Energy History Comparing the Billboard Hot 100 with All Songs

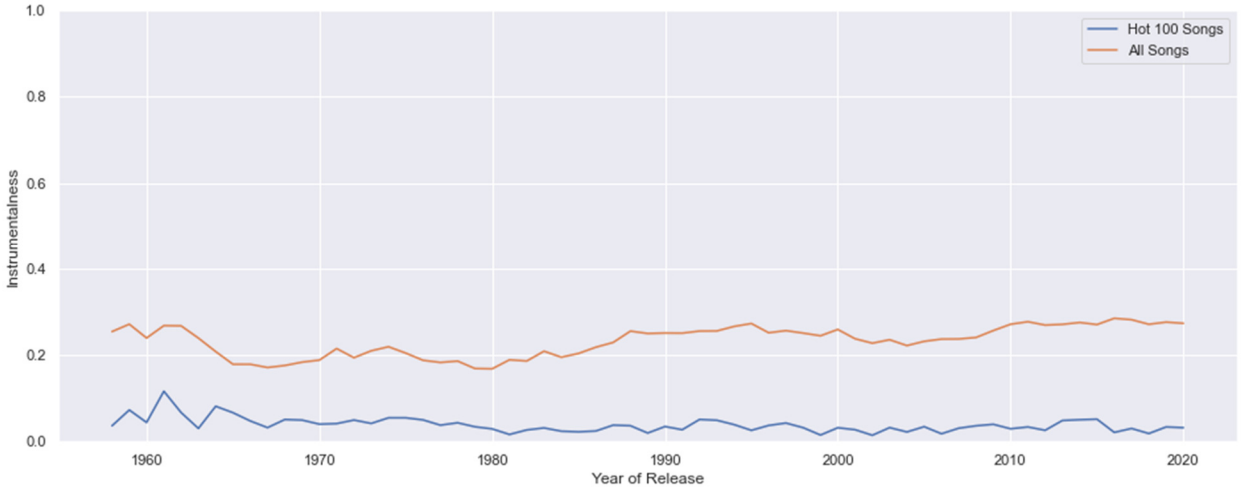


Figure 18. Instrumentalness History Comparing the Billboard Hot 100 with All Songs

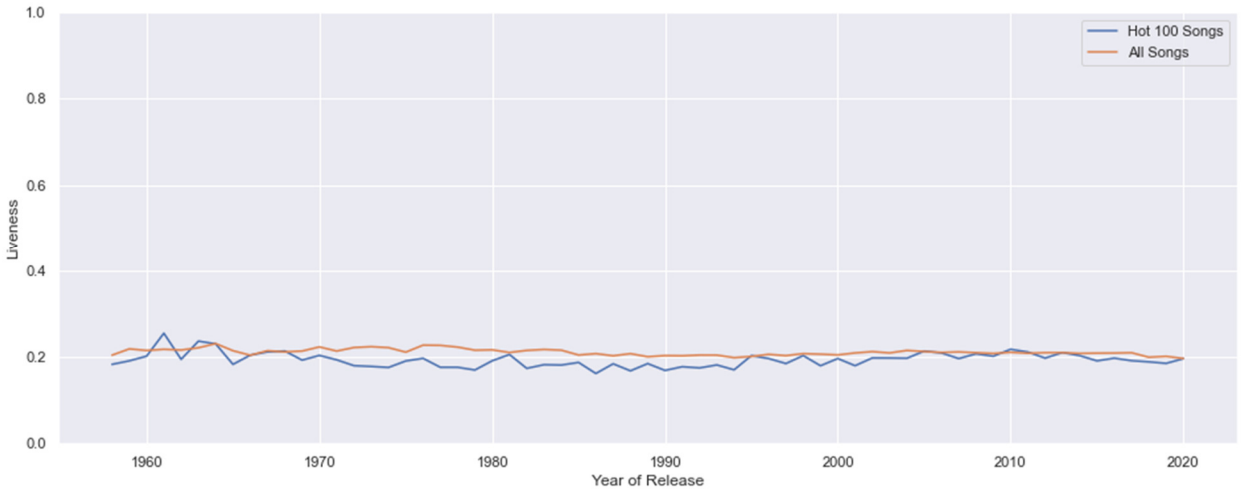


Figure 19. Liveness History Comparing the Billboard Hot 100 with All Songs

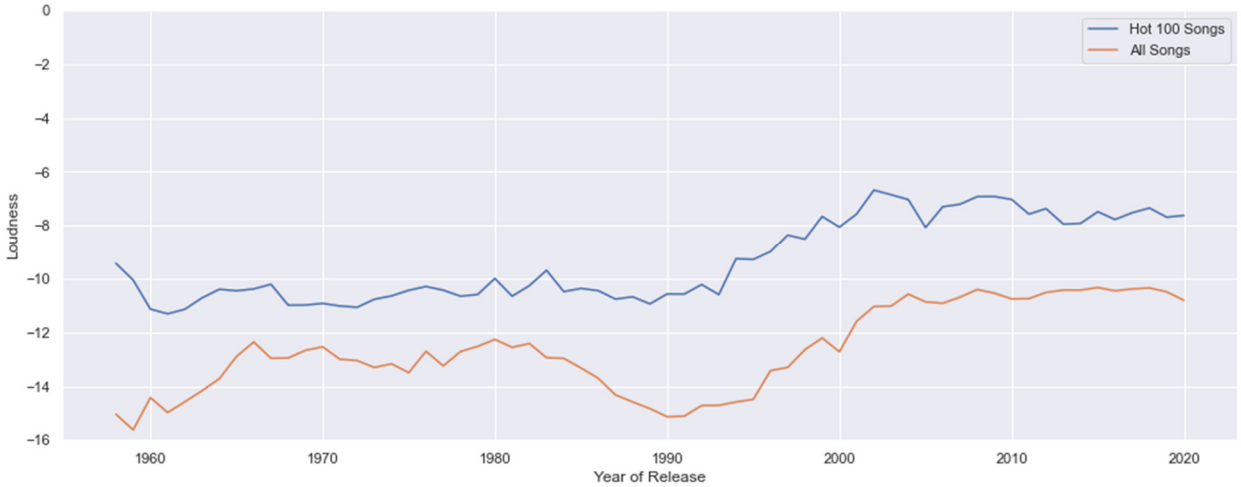


Figure 20. Loudness History Comparing the Billboard Hot 100 with All Songs

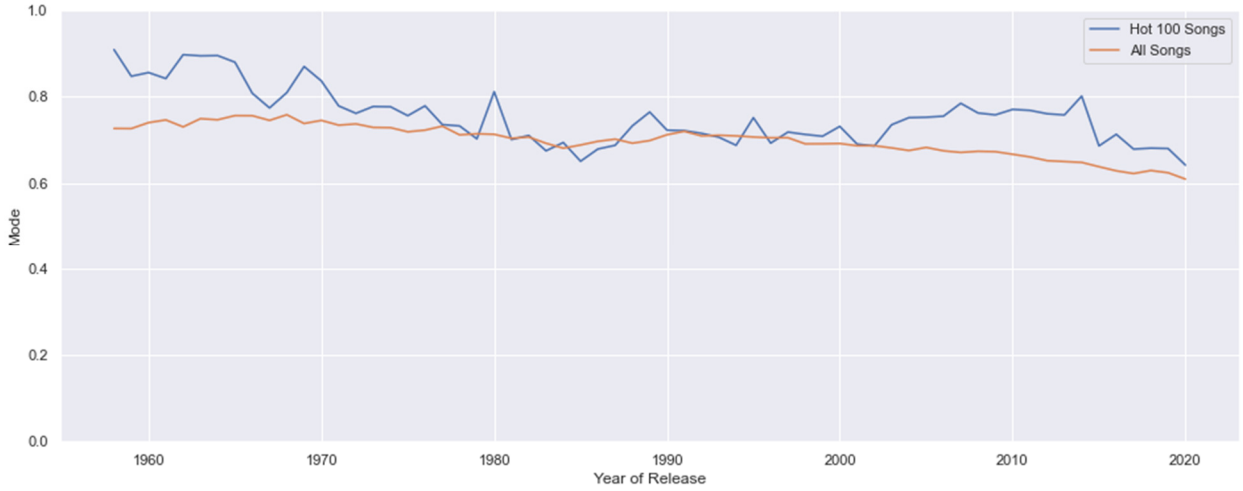


Figure 21. Mode History Comparing the Billboard Hot 100 with All Songs

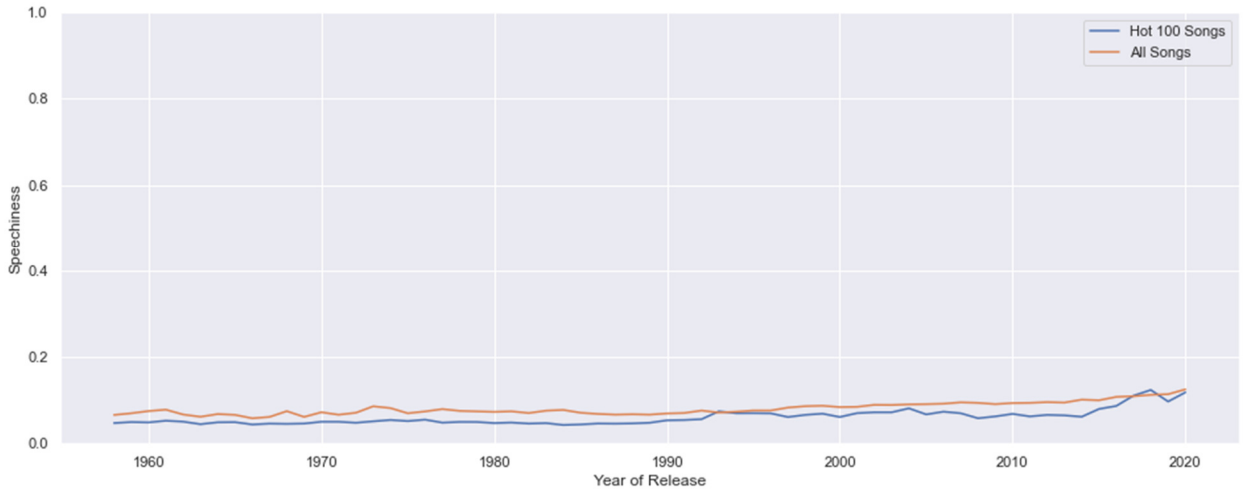


Figure 22. Speechiness History Comparing the Billboard Hot 100 with All Songs

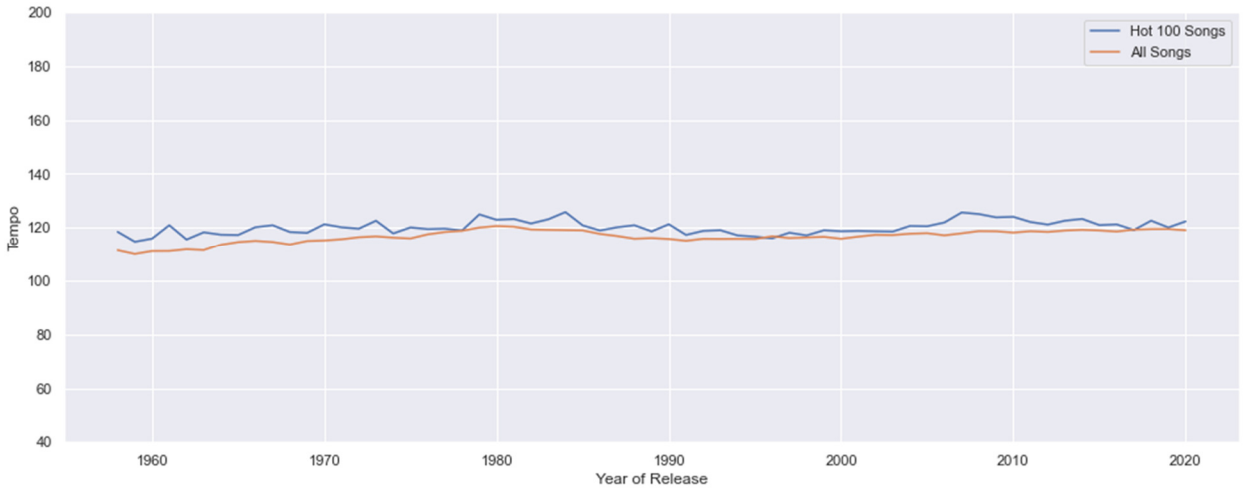


Figure 23. Tempo History Comparing the Billboard Hot 100 with All Songs

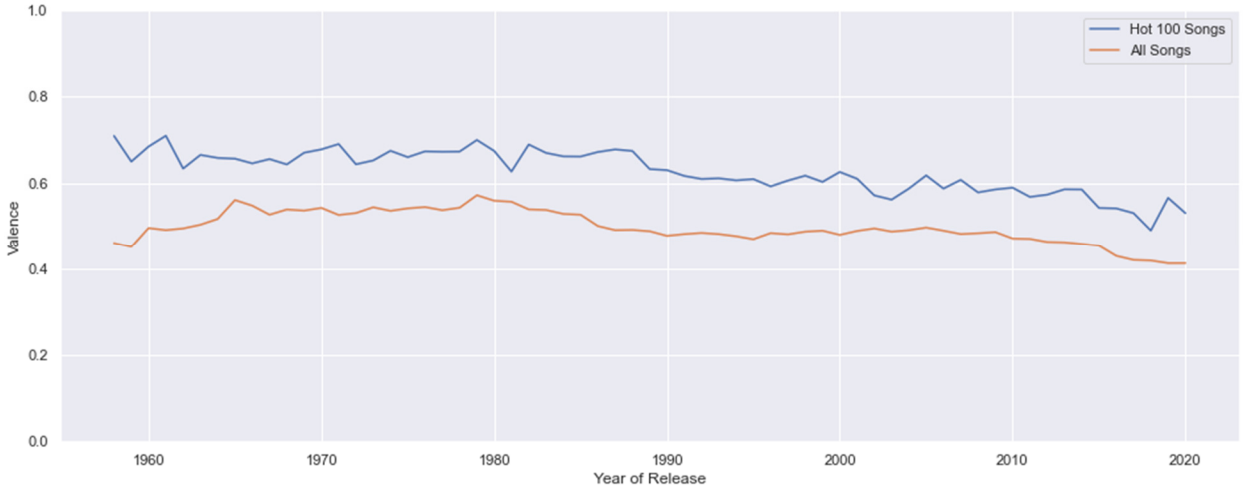


Figure 24. Valence History Comparing the Billboard Hot 100 with All Songs

The following figure shows the Billboard Hot 100 chart yearly audio feature averages over time. In contrast to the above figures, this figure is grouped by appearance on the charts in a given year, and does not necessarily correspond to the release year for any given song. Also of note, all audio features were normalised to range from 0 to 1 in order to create a consistent plot.

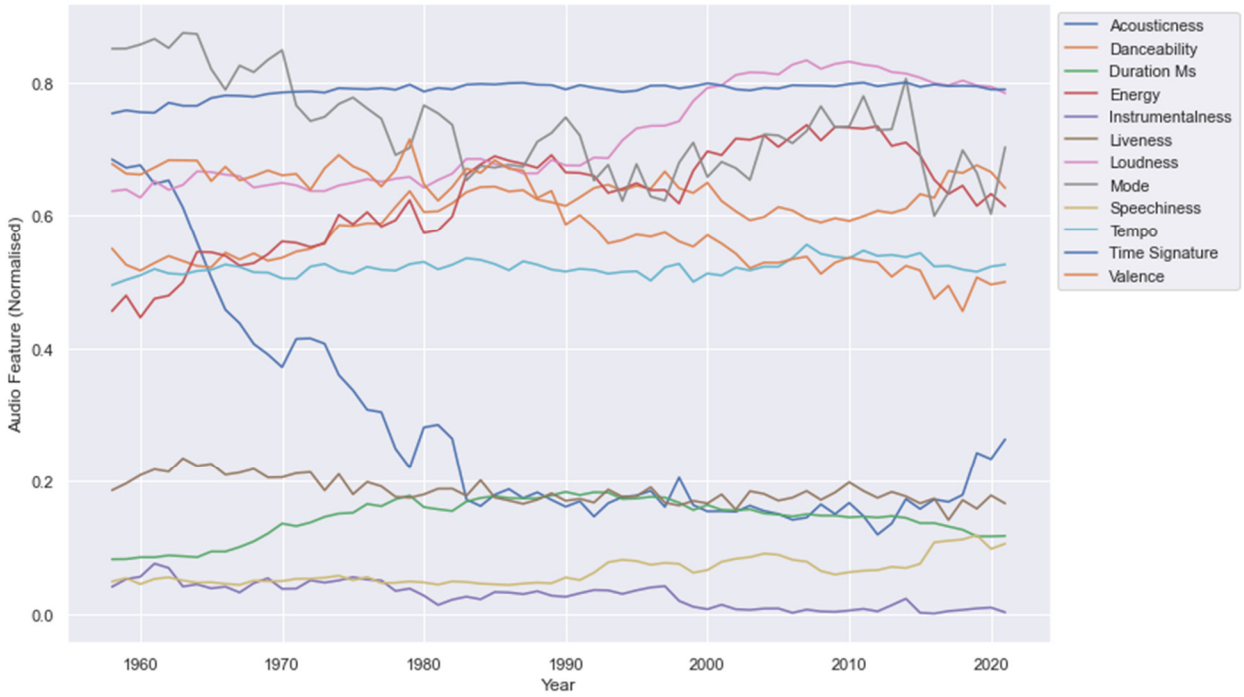


Figure 25. Billboard Hot 100 Historical Charts

A more detailed analysis of each audio feature over time is included in **Attachment 2**.

Correlation Analysis

Correlation analysis is outlined in the below figure. A more detailed correlation analysis in included in **Attachment 2**. The analysis investigated which audio features correlate with each other, differences between Billboard Hot 100 songs and songs not on the list, as well as which features correlate with popularity (as defined above).

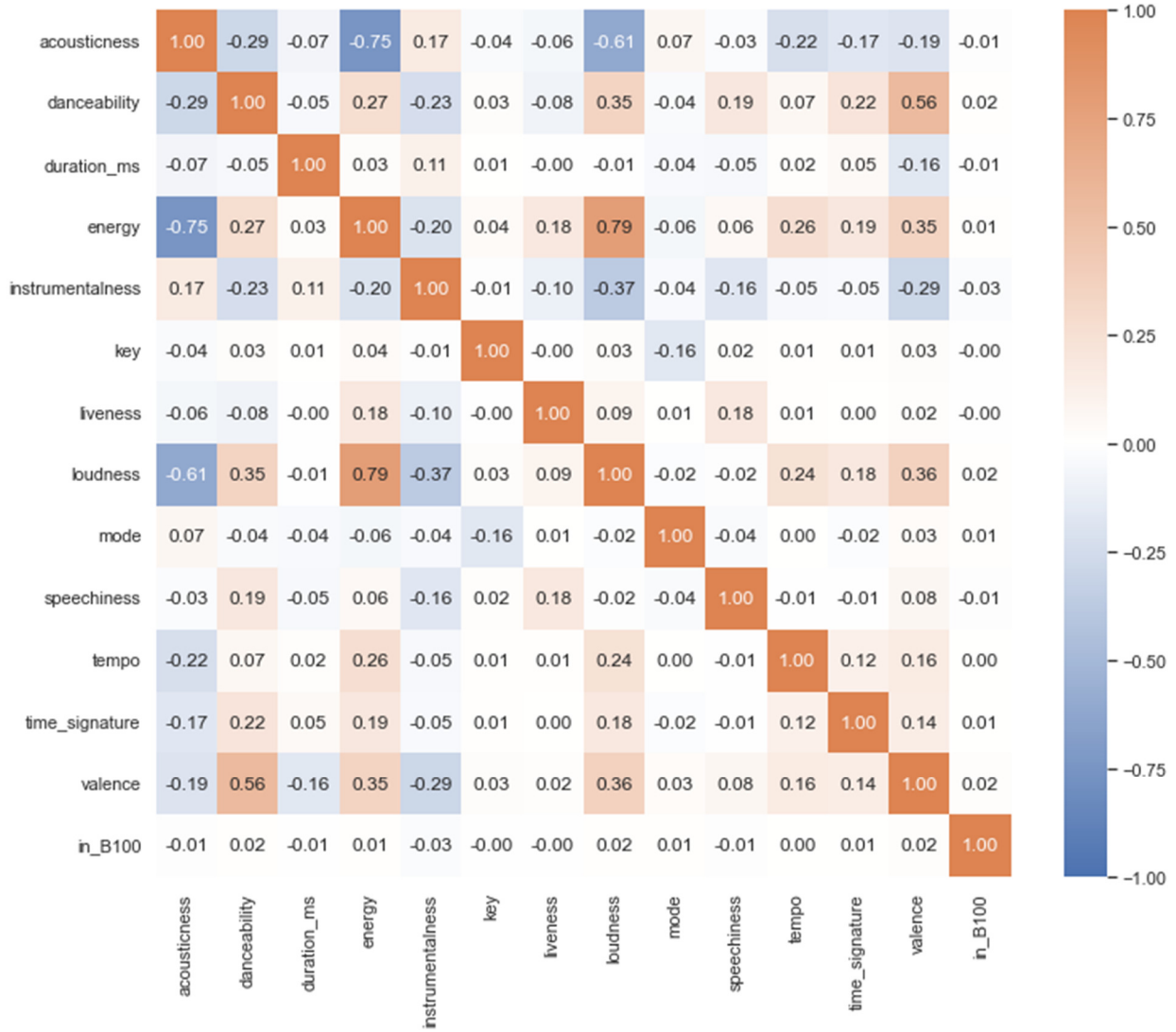


Figure 26. Correlation Analysis Summary

In the above figure, the "in_B100" feature denotes whether or not the song appeared on the Billboard Hot 100 charts.

With a few exceptions, the analysis showed a relatively weak correlation between audio features. This is unsurprising because audio features have been created to represent music simply, accurately, and without redundancies. In later stages of this study, these low correlation values imply that all audio features have the potential to be relevant for predictive analytics, and none of these features can simply be dropped as redundant.

Also of note, every audio feature correlated very weakly with popularity (the "in_B100" feature). As noted in the literature review, this is likely due to large varieties of musical styles all being included in the calculations. Separating analysis into genres yielded stronger correlations for some of the audio features, although correlations were still relatively weak. However, as shown later in this report, clustering into genres prior to classification did not improve classification results. Genres are discussed in the next section, and detailed analysis is included in **Attachment 2**. Classification is discussed later in this report.

Correlation analysis was also conducted strictly on the Billboard Hot 100 dataset. The below figure shows the correlation between audio features and Billboard Hot 100 performance, both peak-rank and weeks-on-board.

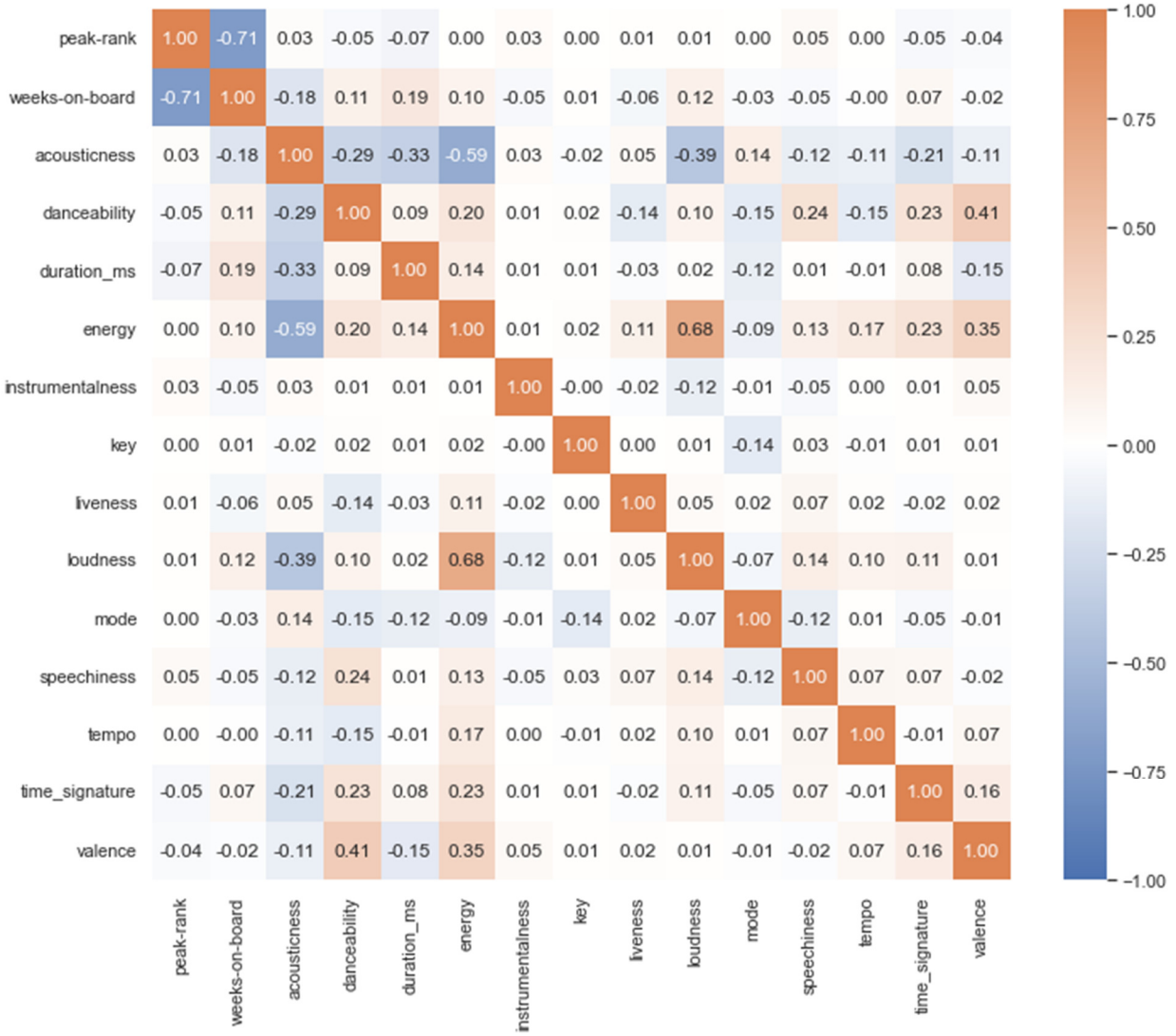


Figure 27. Billboard Hot 100 Correlation Analysis Summary

As shown in the above figure, a number of audio features have a small correlation with performance on the Billboard Hot 100. The following figures show the sorted correlation coefficients corresponding to weeks-on-board and peak-rank, respectively. Note that the order of the charts is reversed because lower peak rank is optimal, whereas higher weeks-on-board is optimal. More detailed analysis is included in **Attachment 2**.

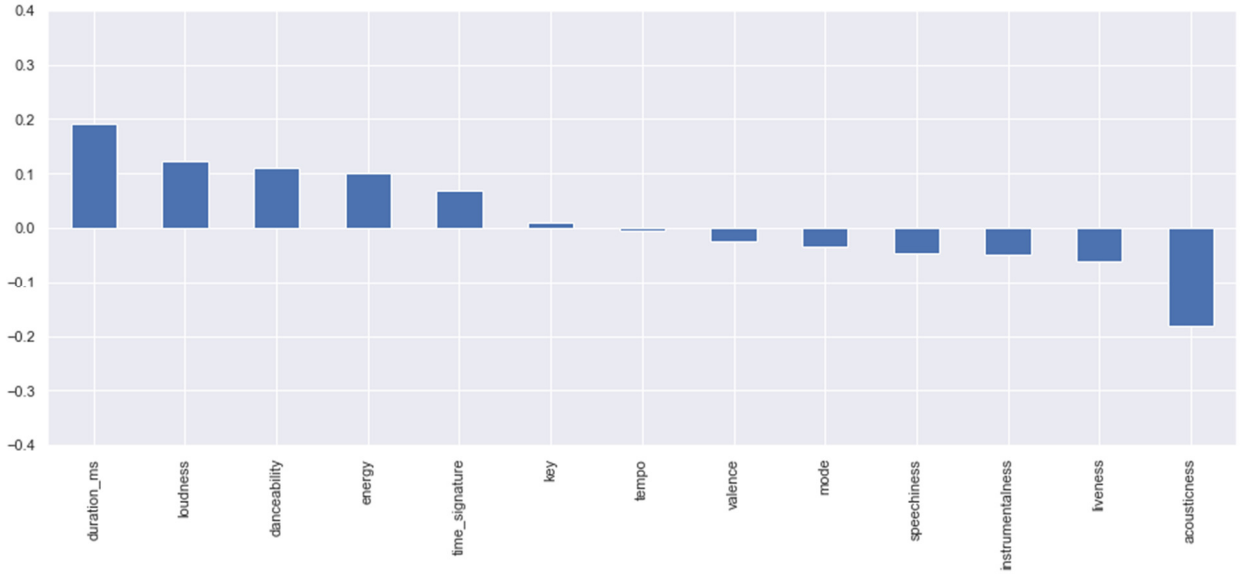


Figure 28. Ranked Audio Feature Correlations With Weeks On Billboard Hot 100 Charts

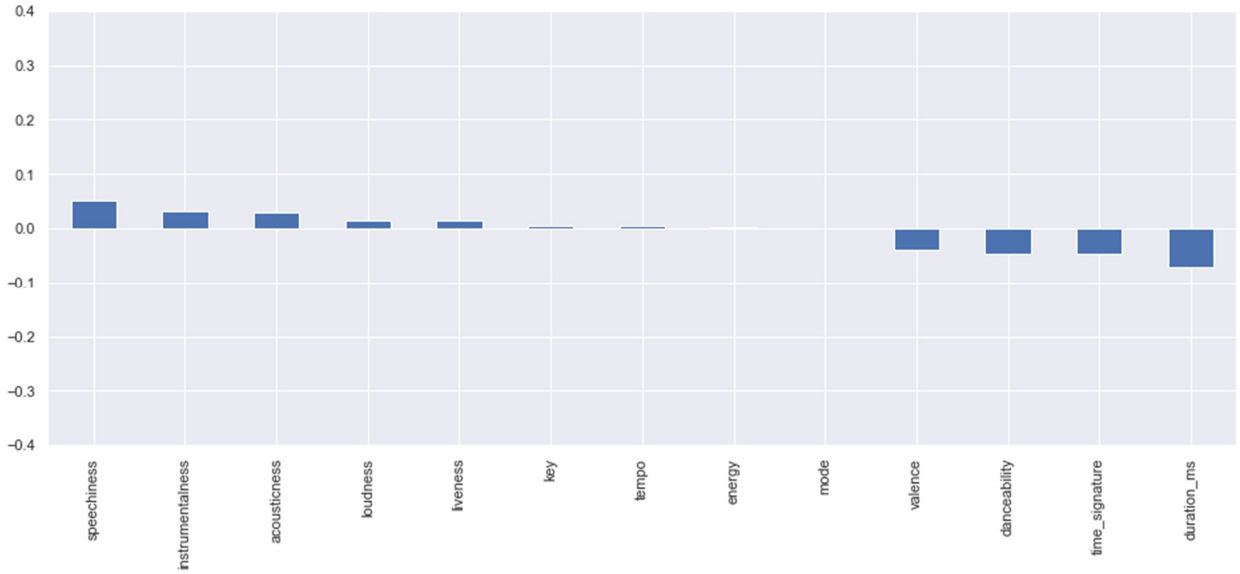


Figure 29. Ranked Audio Feature Correlations With Peak Rank On Billboard Hot 100

Analysis of Genres

In order to improve correlations and future predictions, genre information has been investigated. Correlation analysis showed stronger correlation between audio features and popularity when restricting analysis to specific genres. However, as shown later in this report, clustering into

genres prior to classification did not improve classification results. Detailed genre correlation analysis is included in **Attachment 2**. Classification is discussed later in this report.

Even after excluding all but the most popular genre for each artist, there are still over 5,000 genres in the dataset. Many genres are obscure, and most could more accurately be referred to as sub-genres. For this reason, bins of genres were grouped together to approximate more general genres. Regular Expressions were used to group songs into categories by matching patterns within genre info from the Spotify API. These genre groups were able to reduce variation in audio features to an extent, as shown in the below figure. Details of this genre analysis are included in **Attachment 2**.

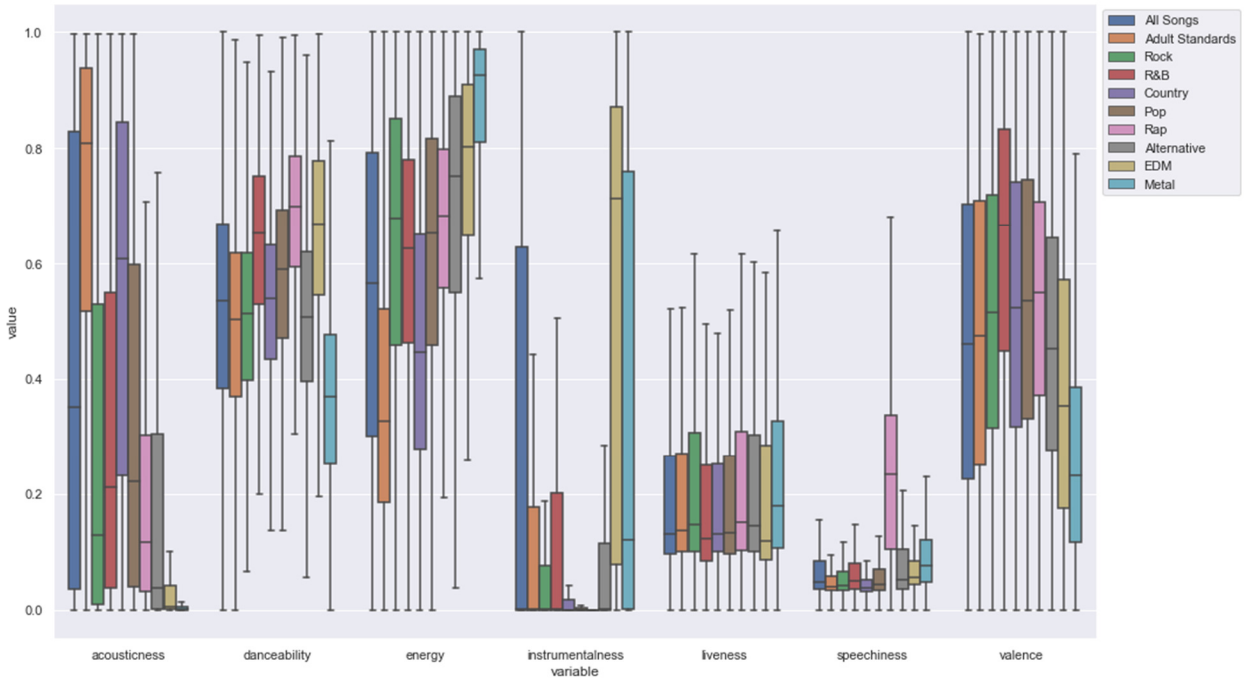


Figure 30. Comparison of Audio Feature Variations Between Genres

It is important to note that these categories are based on assumptions about which genres belong together, and some of these grouping decisions have been made arbitrarily.

Project Approach

The central problem in this project is to utilise publicly available audio feature data to predict the popularity of a song. For this project, two main techniques were employed, namely data mining and predictive analytics.

Outliers

Outliers were investigated using a number of common methods, as well as domain knowledge. The goal of excluding audio features is to exclude all non-musical information from the dataset to attempt to improve the veracity of the analysis. Detailed outlier analysis is included in **Attachment 3**.

Data Mining

First, data mining and knowledge discovery has been used to explore the data and cluster songs by audio features. Genres and audio feature correlation have been explored in detail. Details are included in the **Data Description** section above, and detailed calculations are included in **Attachment 2**.

For this analysis, K-Means Clustering was employed, as it was the most common clustering algorithm noted in the Literature Review. Details of clustering analysis are included in **Attachment 4.**

Predictive Analytics

Secondly, predictive analytics was used to build a number of predictive models. These models attempt to predict whether or not a song will be make it onto the Billboard Hot 100 charts using audio features for that song. Some models incorporated the clusters from the previous phase of the analysis as an attempt to improve predictive accuracy.

This analysis investigated a variety of supervised classification algorithms, namely Logistic Regression, Decision Trees, K-Nearest Neighbours, Random Forests, AdaBoost, and Neural Networks. Statistical significance and predictive power were compared between the utilised algorithms to evaluate the best performance. Details of predictive analytics are included in **Attachment 5.**

Results

Outliers

In order to exclude non-musical tracks from the dataset, a number of methods were investigated, including inter-quartile range (IQR), statistical score, percentiles, and domain knowledge.

None of the standard outlier exclusion methods worked perfectly in isolation. Since musical tracks may have audio features in nearly any range, outliers were still primarily musical in most cases, and thus not excluded. However, a few audio feature ranges were found to consist primarily of non-musical audio tracks. The audio feature ranges excluded from the dataset are as follows:

- Valence = 0

- Speechiness > 0.947 (top 0.5%)
- Temp = 0
- Loudness < -34.7 dB (bottom 0.5%)
- Danceability < 0.0644 (bottom 0.5%)

Additionally, domain knowledge was used to exclude non-musical tracks. In order to determine non-musical tracks, genres were inspected for category names and added to a list of genres to exclude. These excluded genres included categories such as "sleep," "sound effects," and "ringtone." A full list of excluded genres is included in **Attachment 3**.

In addition, music which may not be considered to be "songs" has been excluded (e.g., entire performances, commercials, tv show intros, etc.). Based on domain knowledge, audio tracks less than 1 minute or greater than 10 minutes in length have been excluded.

Detailed outlier analysis is included in **Attachment 3**.

Clustering

The clustering analysis consisted of the following steps:

1. Remove outliers
2. Drop columns not to be included in the clustering
3. Transform data so that all audio features range from 0 to 1
4. Cluster using the K-Means model, optimising K for silhouette score
5. Cluster using the K-Means model, using K equal to the same number of categories as used in the manual genre groupings as defined in **Attachment 2** and **Attachment 4**

6. Cluster data manually using the genre groupings as defined in **Attachment 2** and **Attachment 4**

Clustered data was saved for use in the classification analysis. Details of clustering analysis are included in **Attachment 4.**

Classification

The following machine learning models have been considered in the classification portion of this assessment:

- Logistic Regression
- Decision Trees
- K-Nearest Neighbours
- Random Forests
- AdaBoost

Five-fold cross-validation was used to evaluate the effectiveness of these models. Consistent random seeds were used for the stratified folds and random undersampling in order to have consistency between tests. This enabled a direct matched comparison of out-of-fold predictions between modelling scenarios.

In addition to the above-noted models, neural networks as implemented in TensorFlow were included in the preliminary analysis, but not in the detailed finalised modelling. Details of preliminary classification using neural networks are included in the GitHub repository, but have been excluded from the results and conclusions of this report for simplicity.

Also initially considered but discarded, Support Vector Machine (SVM) models were not feasible do to the extremely large dataset. Some combination of undersampling and PCA may allow for the use of SVM models, but these methods were excluded from this report for simplicity and due to time constraints.

Since the dataset is highly unbalanced, undersampling and oversampling were considered to improve results. Both oversampling and undersampling were shown to improve precision and recall significantly versus the unbalanced calculations. However, undersampling was found to calculate significantly faster without sacrificing precision or recall. Therefore, oversampling was not used for final modelling scenarios. The same random seed was used for all undersampling to assure that all out-of-fold predictions were made on the identical partitions of the dataset, and consistency between modelling scenarios was maintained.

In order to improve results, a large range of hyperparameters was considered for each of the models. A grid search was utilised to find the optimal parameters for each of the models in order to optimise ROC AUC (receiver operating characteristic area under the curve). Due to time constraints, only Logistic Regression and Decision Trees were tuned for the entire dataset. However, limited selections of the dataset were considered for each of the remaining models. Details of this partial dataset tuning are included in the GitHub repository, but have been excluded from the results and conclusions of this report.

Predictions were made with and without clustering the data in order to attempt to improve results. Each of the clusters described above was considered in the classification analysis.

Although a large number of permutations of model, cluster, tuning parameters, and sampling techniques were possible, this assessment was limited to the following scenarios for simplicity and due to time constraints:

1. Logistic Regression – Default Hyperparameters
2. Decision Tree – Default Hyperparameters
3. K-Nearest Neighbours – Default Hyperparameters
4. Random Forest – Default Hyperparameters
5. AdaBoost – Default Hyperparameters
6. Logistic Regression – Tuned Hyperparameters
7. Decision Tree – Tuned Hyperparameters
8. Logistic Regression – Clustered By K-Means Version 1
9. Logistic Regression – Clustered By K-Means Version 2
10. Logistic Regression – Clustered By Genre

For each scenario, the classification analysis consisted of the following steps:

1. Split the data into 5 stratified folds, using a consistent random seed for consistency between tests
2. Train the model using 4 of the folds, balanced using undersampling with consistent random seed for consistency between tests
3. Predict the class for the entire, unbalanced test fold
4. Repeat for each fold, populating out-of-fold predictions into a dataframe including all predictions for cross-validation evaluation

In addition to the analysis outlined above, rough calculations were completed to investigate each of the following topics in greater detail. These extra analysis notebooks are included in the GitHub repository, entitled as follows:

- NOTEBOOK 5A. Choosing Classification Models
- NOTEBOOK 5B. Tuning Classification Models
- NOTEBOOK 5C. Neural Network
- NOTEBOOK 5D. Tuning Classification Models using Spotify popularity

The results of these analyses informed the finalised analysis included in **Attachment 5** ("NOTEBOOK 5. Classification Calculations and Results"), but the results of these rough calculations were not utilised directly in the results and conclusions of this study. They were therefore only included in the repository, and not referenced or included in detail within this report.

Details of predictive analytics and cross-validation results are included in **Attachment 5**, with additional calculations included in the GitHub repository for this project.

Comparison of Results – Effectiveness

Out-of-fold predictions were calculated for each scenario as described above. Each scenario was compared using visualisation and statistical analysis. Since all stratified folds and undersampling utilised consistent random seeds, the results are matched for statistical consistency between tests.

Detailed calculations and visualisations of results are included in **Attachment 6.**

After employing the classification models on clustered and un-clustered data, classification success was mixed. Since the dataset is highly unbalanced, achieving high accuracy was trivial. However, optimising for more nuanced metrics such as precision, recall, or F1-score has proven difficult. Based on visual inspection and statistical analysis, the predictions appear to be correct, albeit not as useful anticipated. This is due to the fact that a large number of songs exist with audio features consistent with hit songs, but only a small portion of all songs become hits. More details are provided below.

Visualisation of Results Using Principal Component Analysis

In order to visualise the distribution of songs within 11-dimensional audio feature space, the dimensionality of the dataset was reduced using Principal Component Analysis (PCA). In this way, a scatterplots can be used to visualise songs within audio feature space. As a result, it is possible to visually compare songs on the Billboard Hot 100, songs not on the Billboard Hot 100, and songs predicted to be popular.

The figures below show scatterplots of the first two principal components for each prediction scenario outlined above.

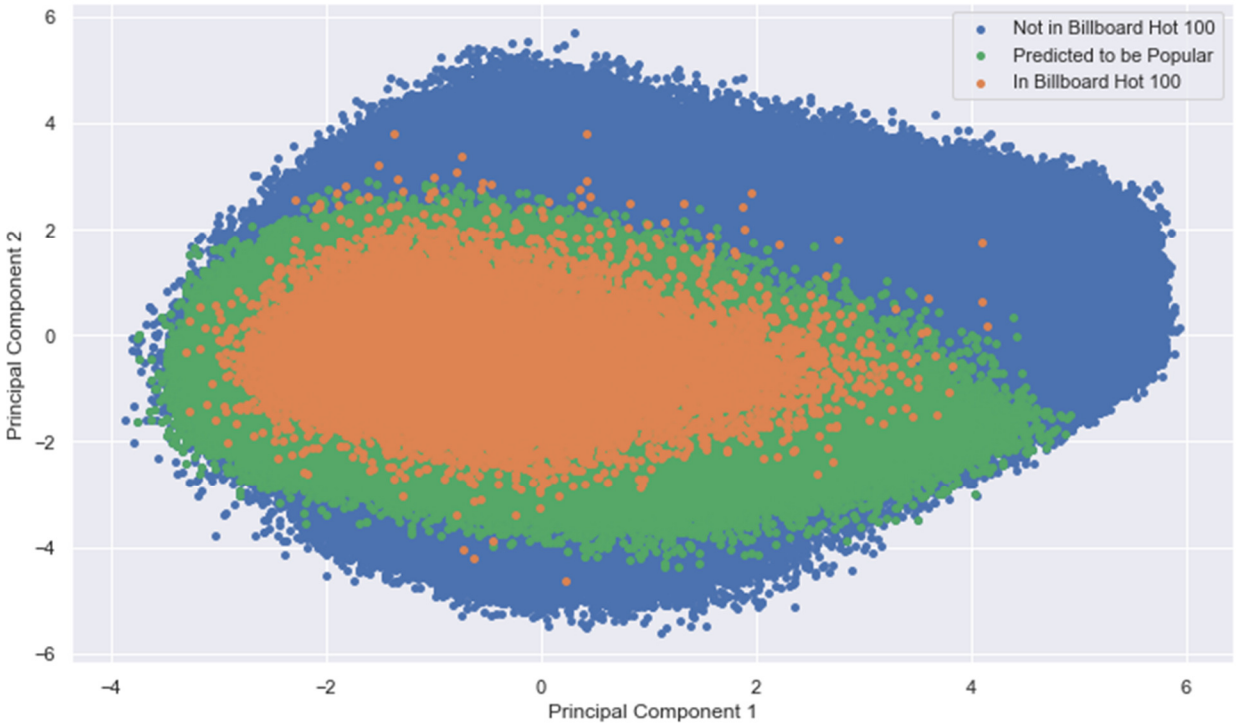


Figure 31. Predicted vs Actual Hits: Logistic Regression - Default Hyperparameters

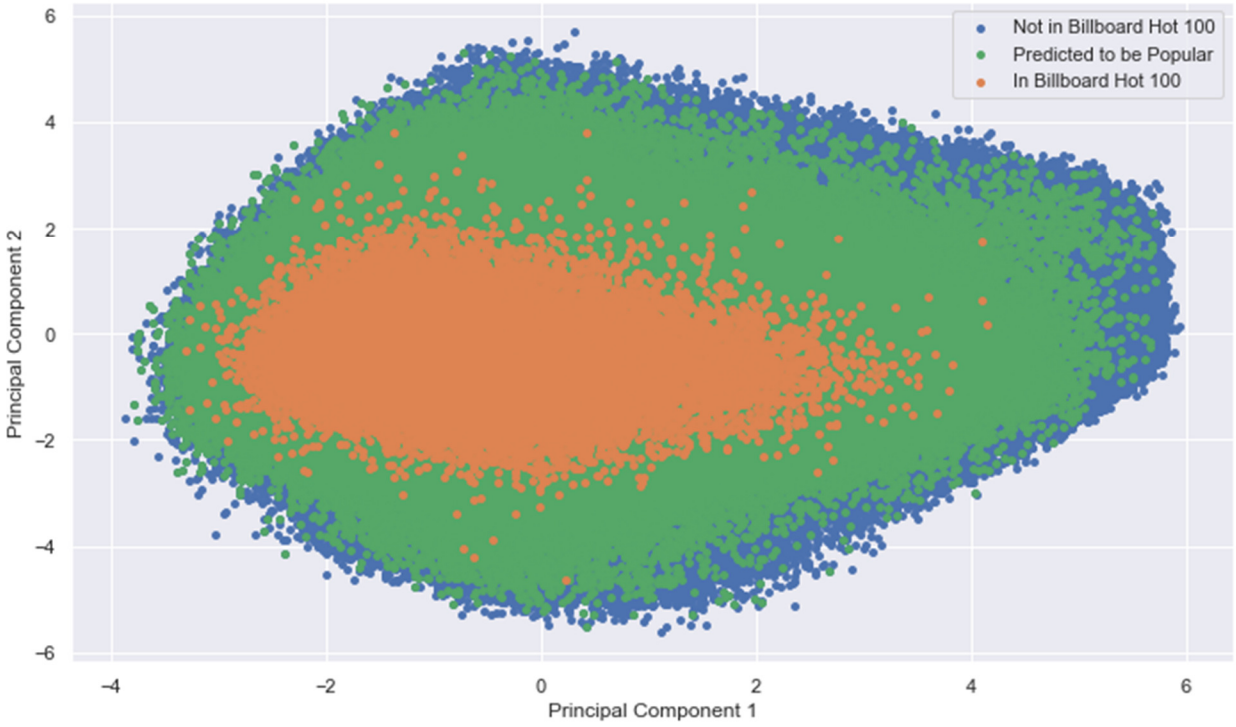


Figure 32. Predicted vs Actual Hits: Decision Tree - Default Hyperparameters

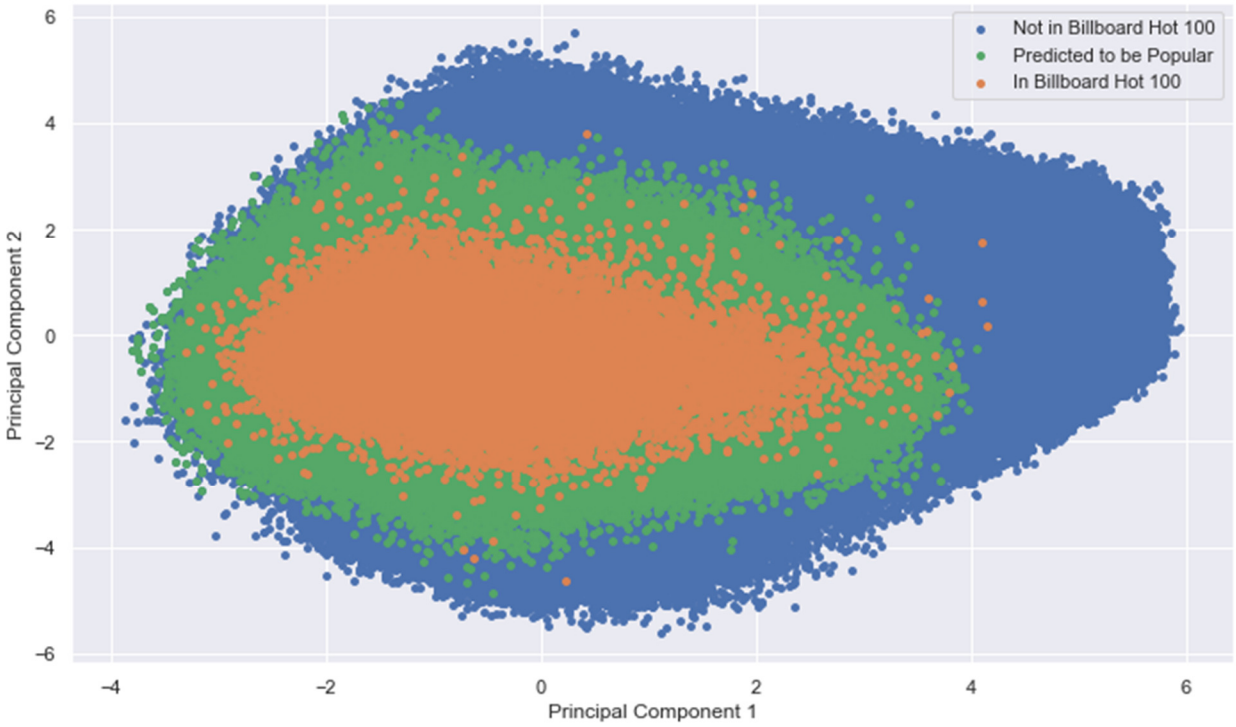


Figure 33. Predicted vs Actual Hits: K-Nearest Neighbours - Default Hyperparameters

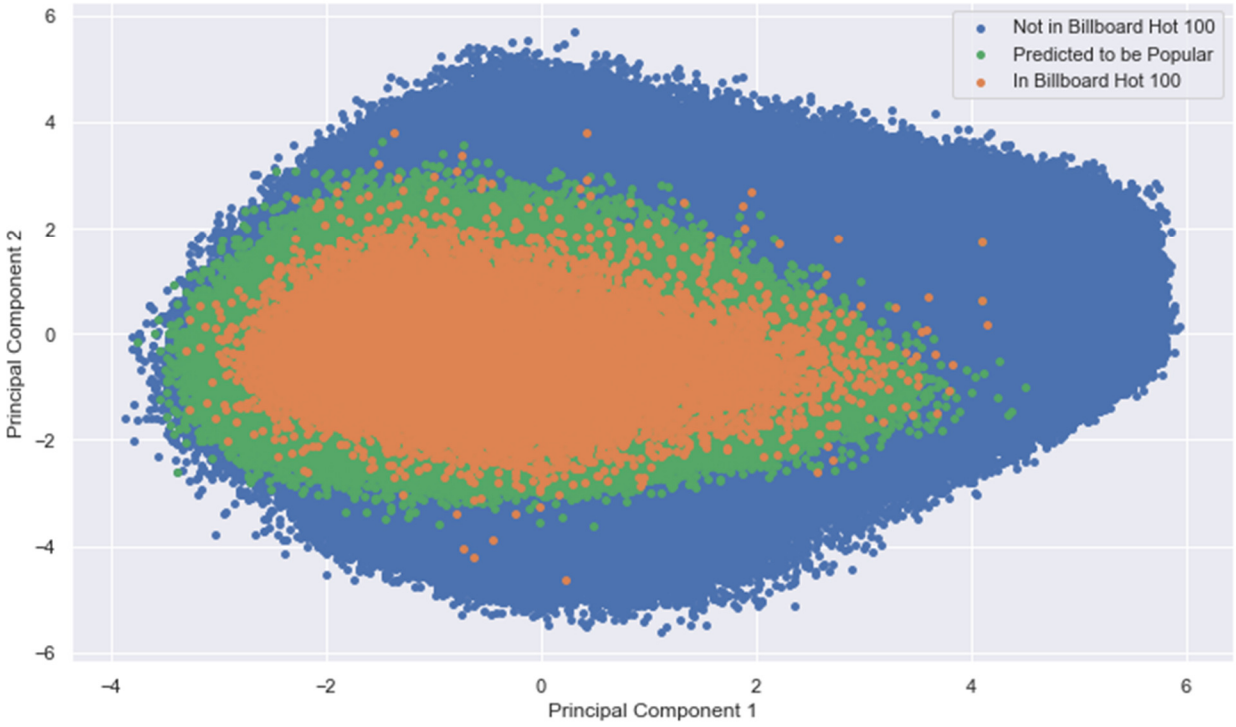


Figure 34. Predicted vs Actual Hits: Random Forest - Default Hyperparameters

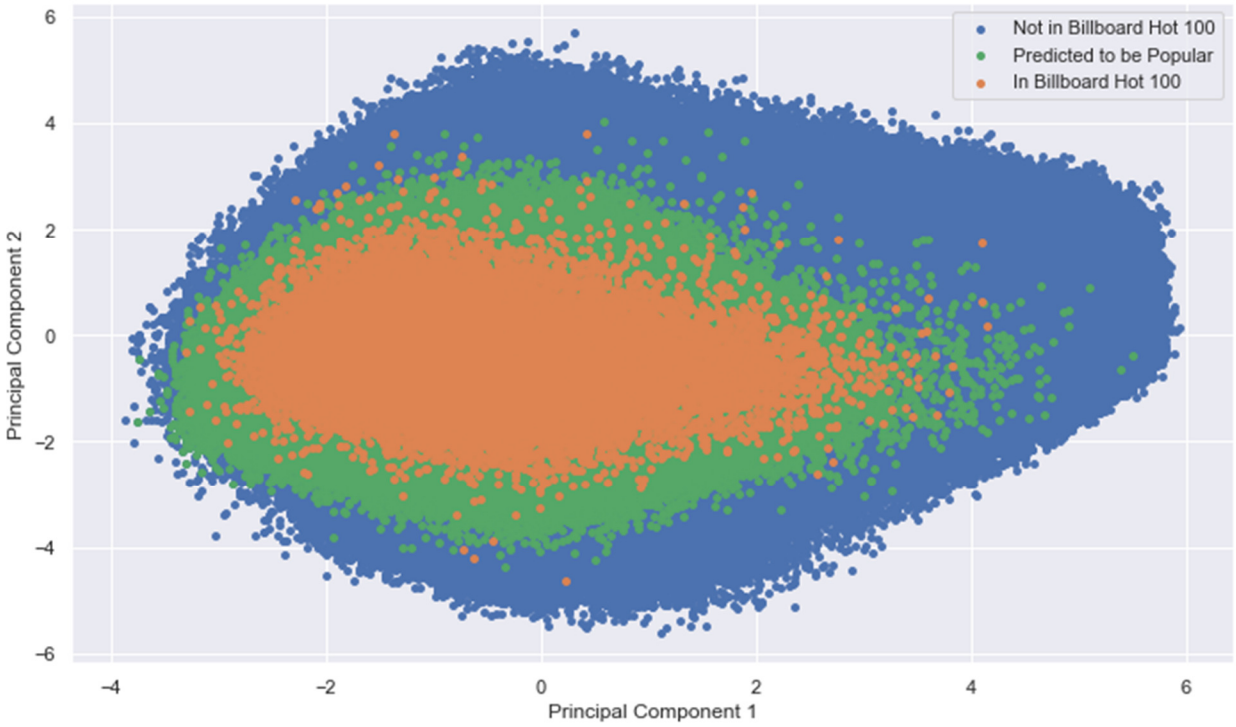


Figure 35. Predicted vs Actual Hits: AdaBoost - Default Hyperparameters

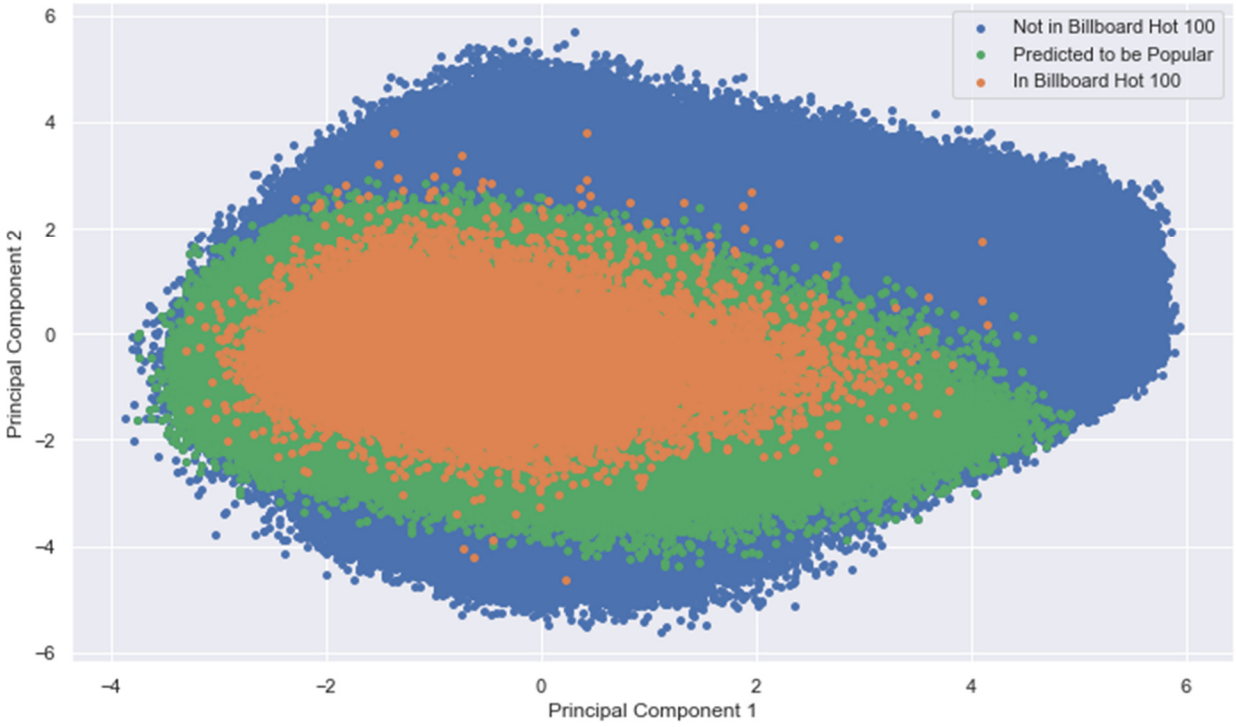


Figure 36. Predicted vs Actual Hits: Logistic Regression - Tuned Hyperparameters

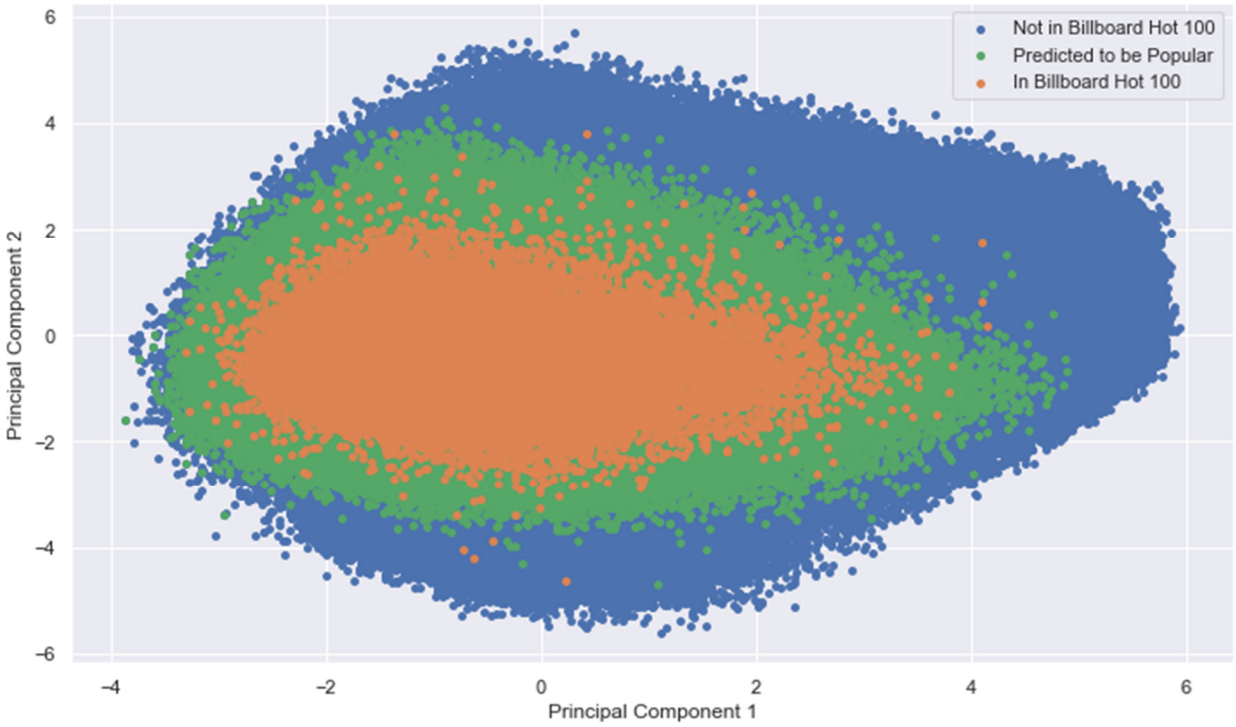


Figure 37. Predicted vs Actual Hits: Decision Tree - Tuned Hyperparameters

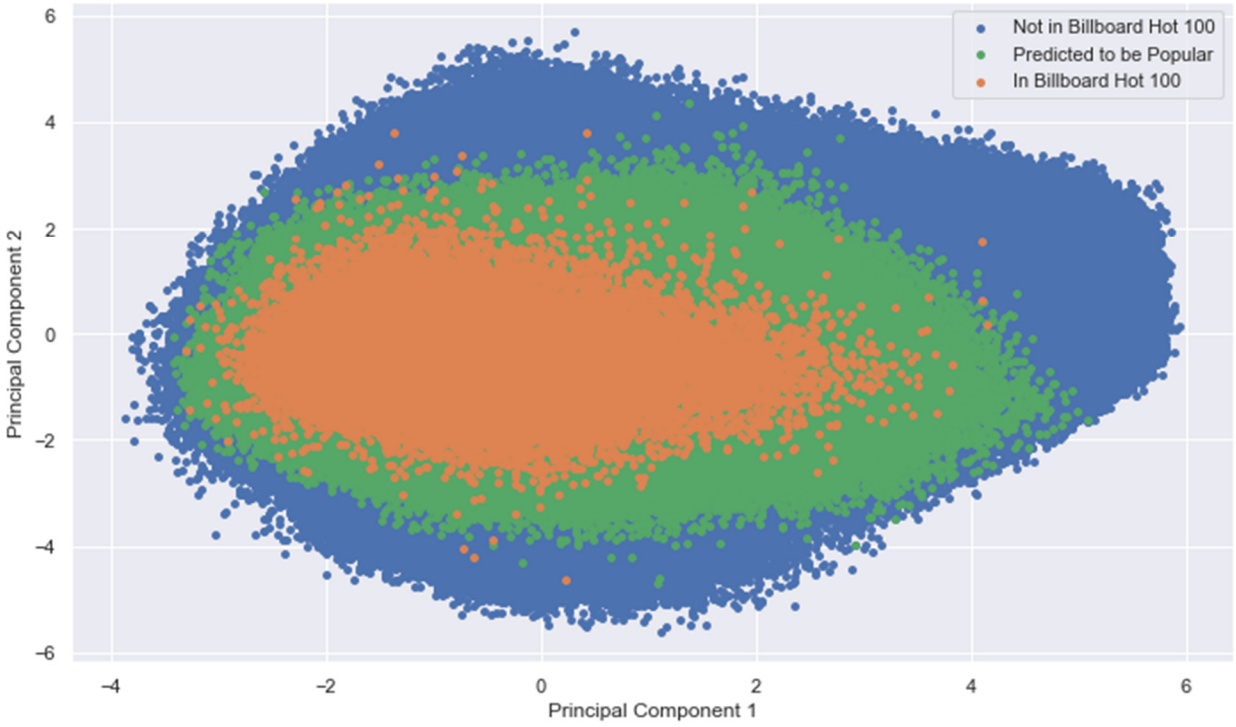


Figure 38. Predicted vs Actual Hits: Logistic Regression - Clustered By K-Means Version 1

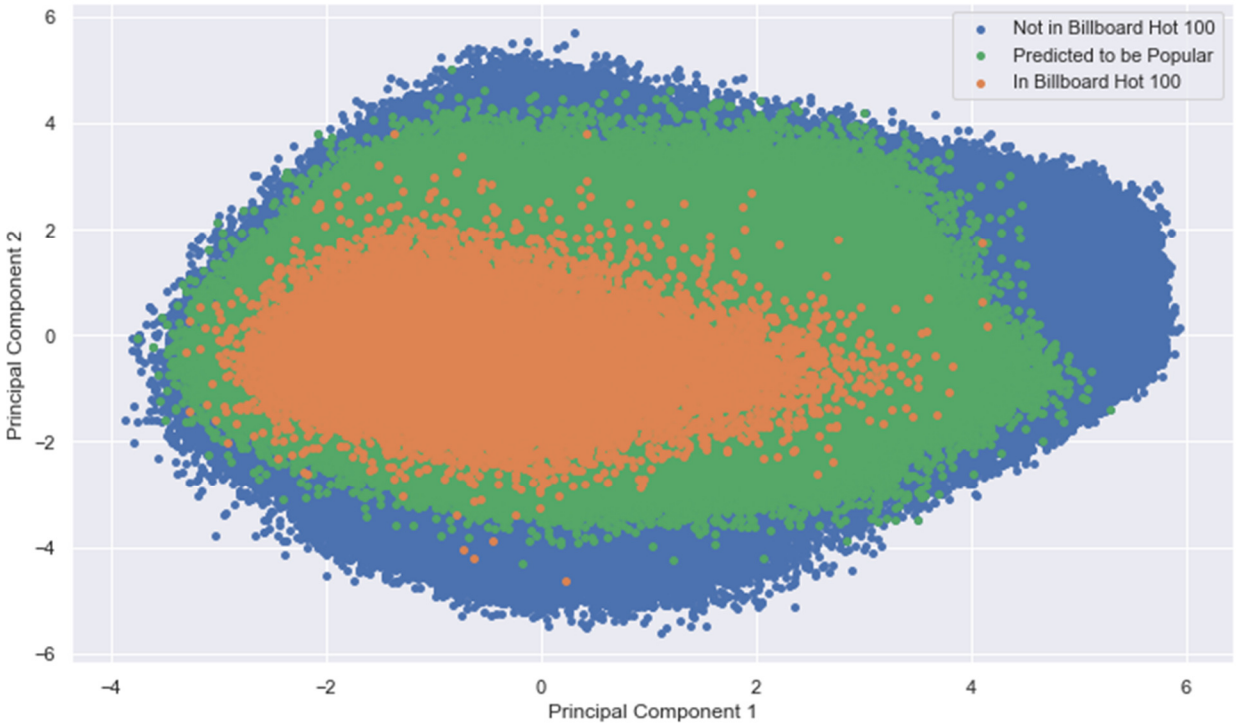


Figure 39. Predicted vs Actual Hits: Logistic Regression - Clustered By KMeans Version 2

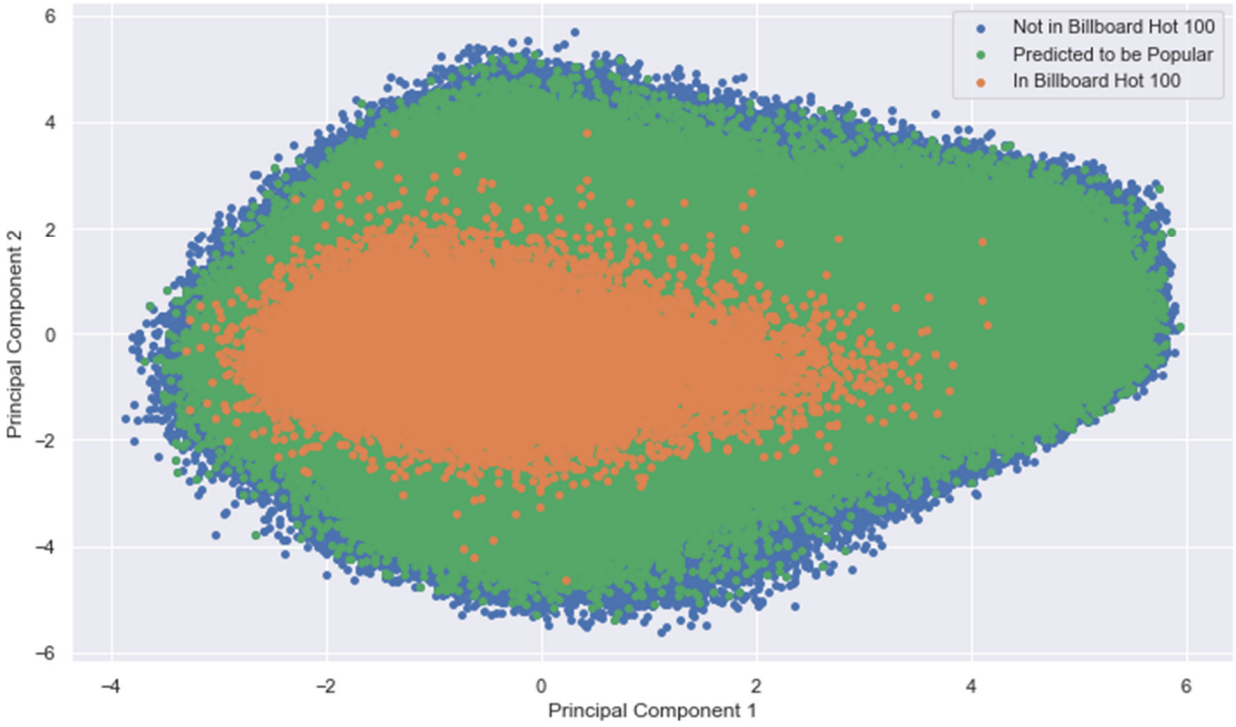


Figure 40. Predicted vs Actual Hits: Logistic Regression - Clustered By Genre

As shown in the figures above, most models performed fairly well at classifying songs into audio feature ranges consistent with Billboard Hot 100 hits. However, some of these modelling scenarios included audio features that deviated more than one would expect based on visual inspection of the PCA scatterplots. Notably, clustering by genre appears to have created results significantly worse than the un-clustered scenarios.

In addition to PCA scatterplots, histograms have been generated for each audio feature, as well as the first 2 principal components to visually inspect predicted results. For each of the histograms, the "Predicted Popular" columns have been subdivided into modelling scenarios ordered as outlined above. To avoid clutter, labels for these series have been excluded from the figures.

The figures below show histograms for the first two principal components comparing songs on the Billboard Hot 100, songs not on the Billboard Hot 100, and songs predicted to be popular.

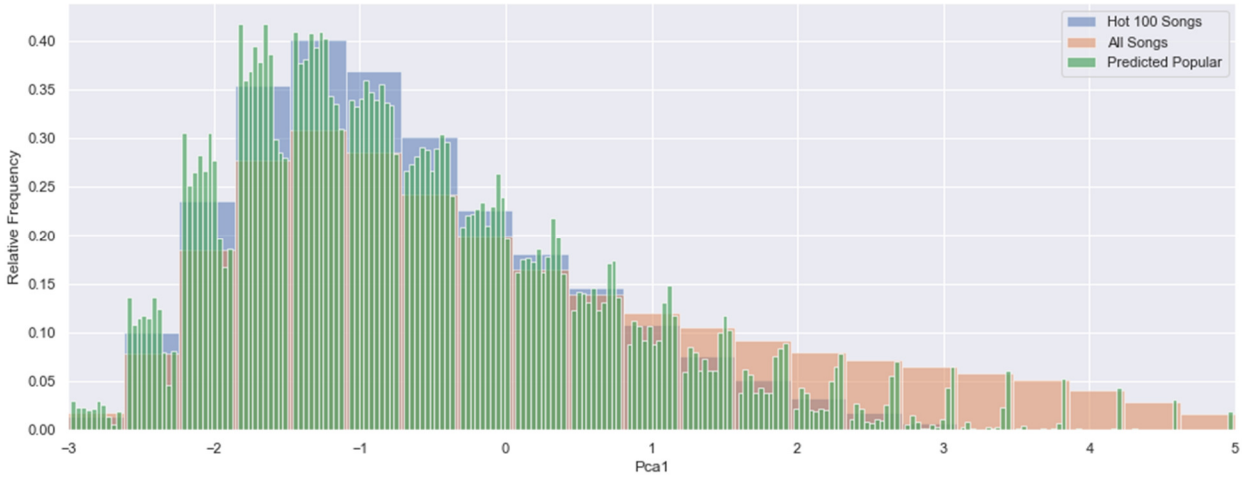


Figure 41. Comparing Hot 100 Songs with Predicted Popularity – First Principal Component

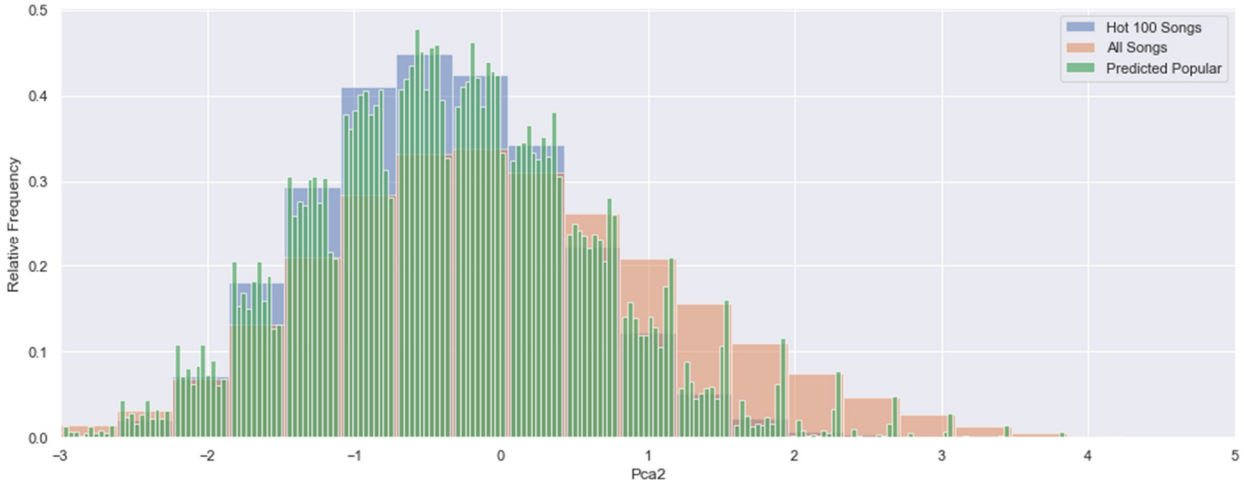


Figure 42. Comparing Hot 100 Songs with Predicted Popularity – Second Principal Component

Similar to the PCA scatterplots, these histograms show predictions relatively consistent with audio features as distributed within the Billboard Hot 100. Also similar to the scatterplots, some predictions deviate more than others, most notably the scenario clustered by genre.

In addition to the above histograms, similar figures for each audio feature are included in **Attachment 6**.

Statistical Analysis

In addition to the visual inspection outlined above, statistical analysis was performed to evaluate the performance of the models and to assess which model performed the best.

Since results for each scenario have similar recall and precision measures, with and without hyperparameter tuning or clustering, it was unknown whether these models were statistically distinct. To assess whether all model were statistically equivalent, a Friedman Test was performed on all matched predictions. The null hypothesis, that all models are statistically

equivalent, was rejected to any arbitrary degree of certainty; the p-value rounds to exactly zero, meaning it is many orders of magnitude below zero.

Since the null hypothesis for the Friedman Test was rejected, at least one of the models is statistically different from the others, and a more detailed statistical test was required. The Wilcoxon Signed Rank Test was performed to compare each model with each other model. Similar to above, the results of these tests indicate that all models are statistically distinct to any arbitrary degree of certainty. Therefore the null hypothesis was rejected for every test. Each of the modelling scenarios is statistically distinct.

Details of statistical analysis are included in **Attachment 6.**

Ranking of Models

Based on the results of the modelling, the models were evaluated for performance. Multiple metrics were considered when evaluating the performance of the models. The most applicable metric for this study is likely the F1-Score, because it incorporates both Recall and Precision. Scoring metrics for each modelling scenario are presented in the below table.

Table 6. Model Performance By Scenario - Scores

Modelling Scenario	Scoring Metric			
	Precision	**Recall**	**F1 Score**	**Accuracy**
Logistic Regression - Default Hyperparameters	0.004	0.802	0.009	0.569
Decision Tree - Default Hyperparameters	0.004	0.647	0.009	0.653
K-Nearest Neighbours - Default Hyperparameters	0.005	0.762	0.009	0.612
Random Forest - Default Hyperparameters	0.006	0.794	0.012	0.678
AdaBoost - Default Hyperparameters	0.005	0.797	0.01	0.61
Logistic Regression - Tuned Hyperparameters	0.004	0.802	0.009	0.57
Decision Tree - Tuned Hyperparameters	0.005	0.757	0.01	0.65
Logistic Regression - Clustered By K-Means Version 1	0.004	0.681	0.009	0.627
Logistic Regression - Clustered By K-Means Version 2	0.004	0.694	0.009	0.618
Logistic Regression - Clustered By Genre	0.002	0.125	0.005	0.872

Based on the scores in the above table, the models are ranked as shown in the below table.

Table 7. Model Performance By Scenario - Ranks

Modelling Scenario	Rank			
	Precision	**Recall**	**F1 Score**	**Accuracy**
Logistic Regression - Default Hyperparameters	5	1	4	10
Decision Tree - Default Hyperparameters	5	9	4	3
K-Nearest Neighbours - Default Hyperparameters	2	5	4	7
Random Forest - Default Hyperparameters	1	4	1	2
AdaBoost - Default Hyperparameters	2	3	2	8
Logistic Regression - Tuned Hyperparameters	5	1	4	9
Decision Tree - Tuned Hyperparameters	2	6	2	4
Logistic Regression - Clustered By K-Means Version 1	5	8	4	5
Logistic Regression - Clustered By K-Means Version 2	5	7	4	6
Logistic Regression - Clustered By Genre	10	10	10	1

As shown in the above tables, the Random Forest generally outperformed the other models. This is consistent with the visual inspection of the results shown previously.

Interesting to note, the clustering did not improve any of the models, and clustering by genre lead to the worst results. This is an unexpected result based on the Literature Review.

Comparison of Results – Efficiency

Unfortunately, the Random Forest model has many tuning parameters and calculates relatively slowly, so could take on the order of days or weeks to completely tune the hyperparameters, assuming ideal conditions. Therefore, an optimised Random Forest model was not tested. It is anticipated that this model would perform better than the untuned model. This optimised Random Forest model is outside the scope of this project, but is discussed in the Future Work section below.

Another time limitation involved oversampling versus undersampling. Oversampling was significantly more time consuming due to the large size of the dataset. Incorporating multiple model runs to perform hyperparameter tuning further increased time to calculate, since the model needs to be re-tested for each set of hyperparameters. The following table outlines the

approximate times required to calculate each model using the oversampled and undersampled data, with and without hyperparameter tuning.

Table 8. Efficiency By Machine Learning Model - Training

Machine Learning Model	Approximate Time to Complete Training			
	Single Scenario Undersampled	Single Scenario Oversampled	Tune Hyperparameters Undersampled	Tune Hyperparameters Oversampled
Logistic Regression	28 seconds	50 minutes [1]	61 minutes	4 days [1]
Decision Trees	27 seconds	50 minutes [1]	67 minutes	5 days [1]
K-Nearest Neighbours	18 minutes	30 hours [1]	60 hours [1]	300 days [1]
Random Forest	3.1 minutes	5 hours [1]	90 hours [1]	400 days [1]
AdaBoost	1.3 minutes	2 hours [1]	> 15 hours [2]	80 days [1]

Notes: [1] Estimated based on known factors, likely to be significantly longer based on Note [2], which was estimated to be significantly faster. Models may vary significantly based on time complexity, these estimates should be considered to be very approximate.
[2] Run did not complete, but was aborted after approximately 15 hours of calculation.

As shown in the above table, Logistic Regression and Decision Trees are the fastest models. The remaining models were found to be too time intensive to be fully evaluated in this study.

In order to assess the efficiency of the models, prediction times for fully trained models was also assessed. The below table shows the amount of time each model took to perform

Table 9. Efficiency By Machine Learning Model - Predictions

Machine Learning Model	Time to Prediction Entire Dataset	Approximate Predictions Per Millisecond
Logistic Regression	0.81 seconds	11,000
Decision Trees	1.6 seconds	5,400
K-Nearest Neighbours	18 minutes	8
Random Forest	2.2 minutes	68
AdaBoost	0.83 minutes	180

As shown in the above table, Logistic Regression is the fastest model. However, for singular predictions or small batches of data, any of the models may be adequate.

Due to limitations discussed above, fully tuned models were not considered for this evaluation. It is possible that fully tuned models could perform more or less quickly than default hyperparameter models.

Comparison of Results – Stability

As shown in the above comparison of results, predictions are relatively stable, even using different machine learning algorithms. Although there exists room for improvement, all models provide predictions consistent with the known class data, with the possible exception of clustered data. Statistical analysis and visual inspection confirm these observations.

Although an optimised Random Forest or Neural Network may perform best, a Logistic Regression model performs nearly as well in most cases. More testing would be required to determine an optimal model. Additionally, methods outlined in the Future Work section below should be considered before choosing an optimal model.

Discussion and Conclusions

Discussion and Limitations

Although results are not as precise as desired, the models appear to be functioning correctly. Based on the nature of the problem, it appears that there may be no way to avoid a large percentage of false positives. There is likely too much variance in each of the audio features to predict popularity with any precision, even after clustering or grouping by genre.

Looking at the PCA scatterplots and histograms of predictions, it appears that predictions are lined up well with actual popular songs. Although there is noticeable room for improvement, the main issue with precision involves popular songs taking up a large portion of 2-dimensional audio feature space (and presumably also the full dimensional audio feature space). Therefore,

even with a perfectly optimised model, we would still expect low precision due to the highly unbalanced data, which is inherent to the nature of popularity.

Conclusions

This study had the initial goal of predicting music popularity. Although predicting popularity has proven difficult, it is possible to quantify features that are characteristic of most popular songs. These features may not be sufficient to get a song onto the Billboard Hot 100, but in most cases, a range of audio features does appear to be necessary in order to have a chance of achieving popularity.

Future Work

Based on the findings of this study, a number of potential future areas of investigation are possible. A few interesting options are listed below.

Due to time constraints, an optimised Random Forest model was not assessed. It would be interesting to assess the effectiveness of this model, potentially in comparison to or conjunction with the other future work outlined in this section.

In terms of prediction, rather than predicting popularity, future models may be used to predict whether or not commercial success is possible. By focussing on whether or not the song's audio features lie within the optimal range for popular music, more accurate and useful predictions could be possible.

The use of PCA was utilised in this study for visualisation of higher dimensional data. However, some sources have noted improvements in predictions using PCA as part of the classification

pipeline. It would be interesting to evaluate the performance of these models with and without PCA. PCA could also be used to reduce complexity and speed up slow models like the Random Forest and AdaBoost models, and may allow for the use of SVM models.

In addition, PCA has the potential to be used analytically by fitting a system of inequalities to the lower-dimension PCA space in order to back-calculate potential audio feature ranges for popular songs. This study could involve using the 2-dimensional PCA plots to fit an ellipse or two curves, within/between which popularity is more likely. Using the back-calculated inequalities, a system of equations could be used to describe the potential audio features which are most likely to achieve popularity. This method has the potential to be quicker and more intuitive than machine learning methods, and could potentially lead to higher precision and recall.

Once a useful and streamlined model is available for predicting a song's popularity potential, it would be useful to develop a plugin for musical software. If this was achieved, it would not be necessary to upload music to Spotify before checking the popularity potential of a song. This would require extra steps, most notably the calculation of audio features from raw audio data. It is unknown whether this is possible, due to the proprietary nature of the Spotify API and algorithms. More refined versions of this plugin could even offer advice to help achieve audio feature ranges with improved potential for popularity.

References

Araujo, C. V. S., Cristo, M. A. P., & Giusti, R. (2007). Predicting Music Popularity on Streaming Platforms. ANAIS DO SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO MUSICAL (SBCM 2019).

Billboard Hot 100. (2022, September 6). Wikipedia. https://en.wikipedia.org/w/index.php?title=Billboard_Hot_100&oldid=1108834581

Cataltepe, Z., Yaslan, Y., & Sonmez, A. (2007). Music Genre Classification Using MIDI and Audio Features. EURASIP JOURNAL ON ADVANCES IN SIGNAL PROCESSING.

Chen, Y. C., Chen, Z. C., & Hsia, C. H. (2021). Music Mood Classification System for Streaming Platform Analysis Via Deep Learning Based Feature Extraction. 2021 IEEE INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS-TAIWAN (ICCE-TW).

Cilibrasi, R. L., Vitányi, P., & Wolf, R. D. (2004). Algorithmic clustering of music. Proceedings of the Fourth International Conference onWeb Delivering of Music, 2004. EDELMUSIC 2004..

Dhruvil Dave. (2021, November 9). Billboard "The Hot 100" Songs [Data set]. Kaggle. https://doi.org/10.34740/KAGGLE/DS/1211465

Elicit. (n.d.). https://elicit.org/

Febirautami, L. R., Surjandari, I., & Laoh, E. (2018). Determining Characteristics of Popular Local Songs in Indonesia's Music Market. 2018 5TH INTERNATIONAL CONFERENCE ON INFORMATION SCIENCE AND CONTROL ENGINEERING (ICISCE).

Gao, A. (2021). Catching the Earworm: Understanding Streaming Music Popularity Using Machine Learning Models. E3S Web of Conferences 253, 03024

Google Dataset Search. (n.d.). https://datasetsearch.research.google.com/

Google Scholar. (n.d.). https://scholar.google.com/

Honingh, A. K., & Bod, R. (2011). Clustering and Classification of Music by Interval Categories. MCM.

Huo, Y. (2021). Music Personalized Label Clustering and Recommendation Visualization. Complex..

Jia, X. (2022). A Music Emotion Classification Model Based on The Improved Convolutional Neural Network. COMPUTATIONAL INTELLIGENCE AND NEUROSCIENCE.

Kim, J. H. (2021). Music Popularity Prediction Through Data analysis of Music's Characteristics. International Journal of Science, Technology and Society.

Kim, S., Park, J., Seong, K., Cho, N., Min, J., & Hong, H. (2021). Music-Circles: Can Music Be Represented With Numbers?. ARXIV.

Laurier, C. , Lartillot, O. , Eerola, T. , & Toiviainen, P. (2009). Exploring relationships between audio features and emotion in music.

Lee, J. , & Lee, J. S. (2018). Music Popularity: Metrics, Characteristics, and Audio-Based Prediction. IEEE Transactions on Multimedia.

Li, L. (2021). Learning Recommendation Algorithm Based on Improved BP Neural Network in Music Marketing Strategy. COMPUTATIONAL INTELLIGENCE AND NEUROSCIENCE.

Li, Q., Kim, B. M., Guan, D. H., & Oh, D. W. (2004). A Music Recommender Based On Audio Features. SIGIR.

Li, Q., Myaeng, S. H., & Kim, B. M. (2007). A Probabilistic Music Recommender Considering User Opinions and Audio Features. INF. PROCESS. MANAG..

Li, X., & Li, J (2022). Music Classification Method Using Big Data Feature Extraction and Neural Networks. JOURNAL OF ENVIRONMENTAL AND PUBLIC HEALTH.

Malte Grosse. (2022, March 23). 8+ M. Spotify Tracks, Genre, Audio Features [Data set]. Kaggle. https://www.kaggle.com/datasets/maltegrosse/8-m-spotify-tracks-genre-audio-features/

Martín-Gutiérrez, D. , Peñaloza, G. H., Belmonte-Hernández, A. , & García, F Á. (2020). A Multimodal End-to-End Deep Learning Architecture for Music Popularity Prediction. IEEE ACCESS.

O'Toole, K., & Horvát, E. Á. (2022). Novelty and Cultural Evolution in Modern Popular Music. ARXIV.

Paper Digest. (n.d.). https://www.paperdigest.org/

Reiman, M., & Örnell, P. (2018). Predicting Hit Songs with Machine Learning. EXAMENSARBETE INOM TEKNIK, GRUNDNIVÅ, 15 HP.

Rodolfo Figueroa. (2020, December 22). Spotify 1.2M+ Songs [Data set]. Kaggle. https://www.kaggle.com/datasets/rodolfofigueroa/spotify-12m-songs

Schedl, M. (2013). Ameliorating Music Recommendation: Integrating Music Content, Music Context, and User Context for Improved Music Retrieval and Recommendation.

Setiadi, D. R. I. M., Rahardwika, D. S. , Rachmawanto, E. H. , Sari, C. A. , Susanto, A., Mulyono, I. U. W. , Astuti, E. Z. , & Fahmi, A. (2020). Effect of Feature Selection on The Accuracy of Music Genre Classification Using SVM Classifier. 2020 INTERNATIONAL SEMINAR ON APPLICATION FOR TECHNOLOGY OF INFORMATION AND COMMUNICATION (ISEMANTIC).

Shi, J. (2021). Music Recommendation Algorithm Based on Multidimensional Time-Series Model Analysis. COMPLEX..

Spotify for Developers. (n.d.). https://developer.spotify.com/documentation/web-api/

West, K. (2008). Novel Techniques for Audio Music Classification and Search. ACM SIGMULTIMEDIA RECORDS.

Wilkes, B., Vatolkin, I., & Müller, H. (2021). Statistical and Visual Analysis of Audio, Text, and Image Features for Multi-Modal Music Genre Recognition. ENTROPY (BASEL, SWITZERLAND).

Xu, Y., & Xu, S. (2021). A Clustering Analysis Method for Massive Music Data.

Yang, L. C., Chou, S. Y., Liu, J. Y., Yang, Y. H., & Chen, Y. (2017). Revisiting the problem of audio-based hit song prediction using convolutional neural networks. 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).

Attachments

Attachment 1

Imports

In [274…

```
# import modules
import pandas as pd
import numpy as np
import re
from ast import literal_eval
import spotipy

# jupyter notebook full-width display
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

# pandas formatting
pd.set_option('display.float_format', '{:_.2f}'.format)
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 200)
```

In [4]:

```
# helper functions

def find_all_tracks(track_title, artist_name):
    """ returns list of lists ['id', 'song', 'artist'] """
    track_info = spotify.search(q='artist:' + artist_name + ' track:' + track_title, type='track')

    if track_info['tracks']['items'] == []:  # if track doesn't exist on Spotify
        return 'MISSING'
    else:
        all_tracks = []
        number_of_results = len(track_info['tracks']['items'])

        # check if there is a better match
        for i in range(number_of_results):
            track_id = track_info['tracks']['items'][i]['id']
            artist_name = track_info['tracks']['items'][i]['artists'][0]['name']
            song_name = track_info['tracks']['items'][i]['name']
            all_tracks.append([track_id, song_name, artist_name])

        # if we made it through the loop without returning, note 'MISSING' and return the 0th id
        return all_tracks


def remove_punctuation(text_input):
    text_input = str(text_input)  # avoid float errors in applymap()
    text_input = re.sub(r'&', 'and', text_input)  # replaces & with 'and'
    text_input = re.compile(r'[^a-zA-Z 0-9]').sub('', text_input)
    return text_input.lower().strip()


def clean_text(text_input):
    text_input = str(text_input)  # avoid float errors in applymap()
    text_input = text_input.strip().lower()
    text_input = re.sub(r'&', 'and', text_input)  # replaces & with 'and'
    text_input = re.sub(r'and.+', '', text_input)  # removes text after the 'and'
    text_input = re.compile(r'the').sub('', text_input)  # remove all 'the' (maybe just need the 1st word?)
    text_input = re.sub(r',.+', '', text_input)  # removes all misc artists, after comma
    text_input = re.sub(r'(?:feat).+', '', text_input)  # removes all misc artists, after 'feat'
    text_input = re.sub(r'\(.+', '', text_input)  # removes text after first bracket
    text_input = re.sub(r'\-.+', '', text_input)  # removes text after first dash
    text_input = re.compile(r'[^a-zA-Z 0-9]').sub('', text_input)  # remove punctuation
    text_input = re.sub(' +', ' ', text_input)  # remove multiple spaces
    return text_input.strip()
```

STEP 1: Import and Setup Datatypes

In [27]:

```
formatting = [
    'id', 'song', 'artist', 'genre', 'release_date',
    'acousticness', 'danceability', 'duration_ms',
    'energy', 'instrumentalness', 'key', 'liveness', 'loudness',
```

```
    'mode', 'speechiness', 'tempo','time_signature', 'valence'
]

dtypes = {
    'key': 'Int16', 'mode': 'Int16', 'time_signature': 'Int16', 'tempo': 'float32',
    'acousticness': 'float32', 'danceability': 'float32', 'duration_ms': 'Int64',
    'energy': 'float32', 'instrumentalness': 'float32', 'liveness': 'float32',
    'loudness': 'float32', 'speechiness': 'float32', 'valence': 'float32'
}
```

In [28]:

```
%%time
##### Billboard Top 100 Historical Data
url_B100 = r'D:\RYERSON\820\Datasets\Billboard The Hot 100 Songs\charts.csv'
dtypes_timeseries = {
    'rank': 'Int16', 'last-week': 'Int16', 'peak-rank': 'Int16', 'weeks-on-board': 'Int16'
}
df_B100 = pd.read_csv(url_B100, dtype=dtypes_timeseries)
df_B100['date'] = pd.to_datetime(df_B100['date'])

# Unique Songs from The Billboard 100 Dataset
df_B100_songs = df_B100[['song', 'artist']].drop_duplicates().sort_values(['artist', 'song']).reset_index(drop=True)

# Add Columns
df_TEMP = pd.DataFrame()
df_TEMP['release_date'] = pd.to_datetime(np.nan)
df_TEMP['genre'] = pd.NA
df_TEMP[formatting[5:]] = np.nan
df_TEMP = df_TEMP.astype(dtypes)
df_B100_songs = pd.concat([df_B100_songs, df_TEMP], axis=1)
df_B100_songs['id'] = pd.NA
df_B100_songs = df_B100_songs[formatting]

# save files as pickle
df_B100.to_pickle('init_df_B100.pickle')
df_B100_songs.to_pickle('init_df_B100_songs.pickle')
```

```
Wall time: 812 ms
```

In [86]:

```
%%time
##### SQL 8+M
"""
    SELECT * FROM tracks
    JOIN r_track_artist ON tracks.id = r_track_artist.track_id
    JOIN artists ON r_track_artist.artist_id = artists.id
    JOIN audio_features ON audio_features.id = tracks.audio_feature_id
"""
df_SQL = pd.read_csv('all_audio_features_sql.csv', dtype=dtypes)

# import genre and release date data
"""
    SELECT tracks.id AS id, release_date, genre_id as genre FROM tracks
    JOIN r_albums_tracks ON tracks.id = r_albums_tracks.track_id
    JOIN albums ON r_albums_tracks.album_id = albums.id
    JOIN r_track_artist ON tracks.id = r_track_artist.track_id
    JOIN r_artist_genre ON r_track_artist.artist_id = r_artist_genre.artist_id
"""
df_genre = pd.read_csv('SQL_track_release_date_and_genre.csv')
df_genre['genre_count'] = df_genre.groupby('genre')['genre'].transform('count')  # add a count column
df_TEMP = df_genre.copy()  # create temp df, sort by most common genre, merge with SQL data
df_TEMP = df_TEMP.sort_values('genre_count', ascending=False).drop_duplicates(['id']).reset_index(drop=True)
df_SQL = df_SQL.merge(df_TEMP, on='id', how='left')

# format and save df_genre
df_genre = (
    df_genre[['genre', 'genre_count']]
    .sort_values('genre_count', ascending=False)
    .drop_duplicates(['genre'])
    .reset_index(drop=True)
)

# formatting
df_SQL['release_date'] = pd.to_datetime(df_SQL['release_date'], unit='ms', origin='unix', errors = 'coerce')
df_SQL = df_SQL.rename(
    {'name:1': 'artist', 'name': 'song','duration:1': 'duration_ms'},
    axis=1
)[formatting].reset_index(drop=True)
```

```
# save files as pickle
df_genre.to_pickle('df_genre.pickle')
df_SQL.to_pickle('df_SQL.pickle')
```

```
Wall time: 3min 35s
```

In [87]:
```
len(set(df_SQL.id))
```

Out[87]:
```
8741110
```

In [79]:
```
%%time
##### Spotify 1.2M+ Songs
url_1M_songs = r'D:\RYERSON\820\Datasets\Spotify 1.2M+ Songs\tracks_features.csv'
dtypes = {
    'key': 'Int16', 'mode': 'Int16', 'time_signature': 'Int16', 'tempo': 'float32',
    'acousticness': 'float32', 'danceability': 'float32', 'duration_ms': 'Int64',
    'energy': 'float32', 'instrumentalness': 'float32', 'liveness': 'float32',
    'loudness': 'float32', 'speechiness': 'float32', 'valence': 'float32'
}
df_1M_songs = pd.read_csv(url_1M_songs, dtype=dtypes)
print(df_1M_songs.shape[0])

# doesn't have genre (would take 1000+ hrs to get using API, will stay NA)
df_1M_songs['genre'] = pd.NA

# explode artists
df_1M_songs['artists'] = df_1M_songs['artists'].apply(literal_eval) #convert to list type
# print(df_1M_songs.explode('artists', ignore_index=True).shape[0]) # correct
df_1M_songs = df_1M_songs.explode('artists').reset_index(drop=True)
print(df_1M_songs.shape[0])

# formatting
df_1M_songs['release_date'] = pd.to_datetime(df_1M_songs['release_date'], errors = 'coerce')
df_1M_songs = df_1M_songs.rename({
    'name': 'song',
    'artists': 'artist'
}, axis=1)[[
    'id', 'song', 'artist', 'genre', 'release_date',
    'acousticness', 'danceability', 'duration_ms',
    'energy', 'instrumentalness', 'key', 'liveness', 'loudness',
    'mode', 'speechiness', 'tempo','time_signature', 'valence'
]].reset_index(drop=True)

# NOTE: DO NOT DROP DUPLICATES YET, NEED FOR LOOKUP WITH B100

# save files as pickle
df_1M_songs.to_pickle('df_1M_songs.pickle')
```

```
1204025
1798207
Wall time: 19.7 s
```

In [91]:
```
%%time
# merge SQL and CSV AF to get df_10M
ids_SQL = set(df_SQL['id'].to_list())
df_TEMP = df_1M_songs.copy()
df_TEMP = df_TEMP[~df_TEMP.id.isin(ids_SQL)]  # drop songs already in SQL before merge
df_10M = pd.concat([df_SQL, df_TEMP]).reset_index(drop=True)

# save as pickle
df_10M.to_pickle('init_df_10M.pickle')
```

```
Wall time: 25.8 s
```

In [92]:
```
len(set(df_10M.id))
```

Out[92]:
```
9592419
```

STEP 2: Spotify API - GET id, then audio features

In [93]:
```
# reload data if required
df_B100_songs = pd.read_pickle('init_df_B100_songs.pickle')

# number missing ids
df_B100_songs[df_B100_songs.id.isnull()].shape[0]
```

Out[93]:
```
29681
```

get a temporary authorization token from: https://developer.spotify.com/console/get-search-item

In [109…
```
# input the temporary token
TEMP_TOKEN = input('Enter token: ')

# create a spotify object
spotify = spotipy.Spotify(auth=TEMP_TOKEN)
```

```
Enter token: BQC7g6bIGQdnc-SGz2llxofZdjOcE6-TD9T4ft9AYVjBUEGyxDetGEojqsp98avFRwnLKCtOa07iFlvvqm_ZZCHtfOnqVj6JgjBNvVyE9nWH8oyhqw
6O4QPSbBDnTTZ1wfSSuHM3-W5hrRGn92Jzb2JfoKmUMJ4F_QDX7qGZoo2H
```

In [100…
```
%%time
# loop to GET id

counter = 0
start_over_at = 17600
if start_over_at == 0:
    id_from_API = set()

for i, row in df_B100_songs.iterrows():

    if counter % 100 == 0:
        print(counter, end=' ')
    if counter % 1000 == 0:
        print()

    counter += 1

    if i < start_over_at:  # where we timed out last time
        continue

    # save temp file
    if counter % 1000 == 0:
        df_B100_songs.to_pickle('df_B100_songs_ID_TEMP.pickle')

    if not df_B100_songs.iloc[[i]].isnull()['id'].values[0]:
        continue

    # these are the actual song and artist from the Billboard Hot 100
    song = df_B100_songs.loc[df_B100_songs.index[i], 'song']
    artist = df_B100_songs.loc[df_B100_songs.index[i], 'artist']

    # get all track info from Spotify API matching 'song' and 'artist'
    all_track_info = find_all_tracks(song, artist)

    # restart loop if there are no results
    if all_track_info == 'MISSING':
        continue

    # first subloop - check for direct matched
    is_exact_match = False
    id_found = False

    for track_info in all_track_info:
        temp_id = track_info[0]
        temp_song = track_info[1]
        temp_artist = track_info[2]

        # if there is an exact text match
        is_exact_match = remove_punctuation(temp_song) == remove_punctuation(song) and \
                         remove_punctuation(temp_artist) == remove_punctuation(artist)

        # continue subloop
        if is_exact_match:
            df_B100_songs.loc[df_B100_songs.index[i], 'id'] = temp_id
            id_from_API.add(temp_id)
```

```
            id_found = True
            break

    # if we found the id, go to the next row item, else check for approx matches
    if not id_found:

        is_probable_match = False

        # second subloop - check for indirect matches (shouldn't run if direct match occurs)
        for track_info in all_track_info:
            temp_id = track_info[0]
            temp_song = track_info[1]
            temp_artist = track_info[2]

            # if there is a probable text match
            is_probable_match = clean_text(temp_song) == clean_text(song) and \
                                clean_text(temp_artist) == clean_text(artist)

            if is_probable_match:
                df_B100_songs.loc[df_B100_songs.index[i], 'id'] = temp_id
                id_from_API.add(temp_id)
                break


df_B100_songs.to_pickle('df_B100_songs_ID_COMPLETE.pickle')
```

```
0
100 200 300 400 500 600 700 800 900 1000
1100 1200 1300 1400 1500 1600 1700 1800 1900 2000
2100 2200 2300 2400 2500 2600 2700 2800 2900 3000
3100 3200 3300 3400 3500 3600 3700 3800 3900 4000
4100 4200 4300 4400 4500 4600 4700 4800 4900 5000
5100 5200 5300 5400 5500 5600 5700 5800 5900 6000
6100 6200 6300 6400 6500 6600 6700 6800 6900 7000
7100 7200 7300 7400 7500 7600 7700 7800 7900 8000
8100 8200 8300 8400 8500 8600 8700 8800 8900 9000
9100 9200 9300 9400 9500 9600 9700 9800 9900 10000
10100 10200 10300 10400 10500 10600 10700 10800 10900 11000
11100 11200 11300 11400 11500 11600 11700 11800 11900 12000
12100 12200 12300 12400 12500 12600 12700 12800 12900 13000
13100 13200 13300 13400 13500 13600 13700 13800 13900 14000
14100 14200 14300 14400 14500 14600 14700 14800 14900 15000
15100 15200 15300 15400 15500 15600 15700 15800 15900 16000
16100 16200 16300 16400 16500 16600 16700 16800 16900 17000
17100 17200 17300 17400 17500 17600 17700 17800 17900 18000
18100 18200 18300 18400 18500 18600 18700 18800 18900 19000
19100 19200 19300 19400 19500 19600 19700 19800 19900 20000
20100 20200 20300 20400 20500 20600 20700 20800 20900 21000
21100 21200 21300 21400 21500 21600 21700 21800 21900 22000
22100 22200 22300 22400 22500 22600 22700 22800 22900 23000
23100 23200 23300 23400 23500 23600 23700 23800 23900 24000
24100 24200 24300 24400 24500 24600 24700 24800 24900 25000
25100 25200 25300 25400 25500 25600 25700 25800 25900 26000
26100 26200 26300 26400 26500 26600 26700 26800 26900 27000
27100 27200 27300 27400 27500 27600 27700 27800 27900 28000
28100 28200 28300 28400 28500 28600 28700 28800 28900 29000
29100 29200 29300 29400 29500 29600 Wall time: 42min 49s
```

In [101…

```
# new ids
RUN2_id_from_API = id_from_API.copy()
len(id_from_API)
```

Out[101…

```
21277
```

In [103…

```
df_id_from_API = pd.DataFrame(list(id_from_API))
df_id_from_API.to_pickle('df_id_from_API.pickle')
```

In [105…

```
# 3 duplicates, remove later
# number of songs in new ids (should match len(id_from_API))
df_B100_songs[df_B100_songs.id.isin(id_from_API)].id.count()
```

Out[105…

```
21280
```

In [107…

```
# was 4140 with sloppier (faster) method
# still missing ids
df_B100_songs[df_B100_songs.id.isnull()].shape[0]
```

Out[107…

```
8401
```

get a temporary authorization token from: https://developer.spotify.com/console/get-search-item

In [188…

```
# input the temporary token
TEMP_TOKEN = input('Enter token: ')

# create a spotify object
spotify = spotipy.Spotify(auth=TEMP_TOKEN)
```

```
Enter token: BQDDf0krvUaAgJcHwIpBspgI4OQvVewwi1L8MW90b72DNhEHDQqVXCH0WTh20m8pgBewyDOXnkrB-fBc4okZUFoLXOCMNPxrUtWFd6dQ2iXtArUgGwL-IH1O17KpWUMZbm-xjwb5PC7Beem8EFXMAvHAkFd-fRwOl3dt15fVWGke
```

In [189…

```
%%time
# loop to get genre, release_date, and audio features

# ordered list of genres for choosing the best genre
list_of_ordered_genres = list(df_genre.genre)

start_over_at = 25894
counter = 0
if start_over_at == 0:
    how_many_passes = 0  # for QA, how many genre should be missing
    how_many_fails = 0

for i, row in df_B100_songs.iterrows():

    if counter % 100 == 0:
        print(counter, end=' ')
    if counter % 1000 == 0:
        print()

    counter += 1

    if i < start_over_at:  # where we timed out last time
        continue

    # save temp file
    if counter % 1000 == 0:
        df_B100_songs.to_pickle('df_B100_songs_AF_TEMP.pickle')

    # get the current track id
    track_id = df_B100_songs['id'].iloc[i]

    # if we don't have a new ID, skip the entry (it's not on Spotify)
    if track_id not in id_from_API:
        continue

    # Get Audio Features - 1st GET request
    list_of_features = [
        'acousticness', 'danceability', 'duration_ms', 'energy',
        'instrumentalness', 'key', 'liveness', 'loudness', 'mode',
        'speechiness', 'tempo', 'time_signature', 'valence'
    ]

    try:
        temp_audio_features = spotify.audio_features(track_id)
        for key in list_of_features:
            df_B100_songs.loc[i, key] = temp_audio_features[0][key]
    except:  # madonna has none AF, maybe a remaster????
        how_many_fails += 1
        if how_many_fails < 100:
            print('fail')
        pass

    # Get Release Date - 2nd GET request
    track_info = spotify.track(track_id)
    df_B100_songs.loc[i, 'release_date'] = track_info['album']['release_date']

    try:
```

```
        # Get Release Date Genre - 3rd GET request
        artist_id = track_info['artists'][0]['id']
        artist_info = spotify.artist(artist_id)
        list_of_artist_genres = artist_info['genres']

        most_common_genre = list_of_artist_genres[0] # default to first genre
        if len(list_of_artist_genres) == 1:
            pass
        else:
            for genre in list_of_ordered_genres:
                if genre in list_of_artist_genres:
                    most_common_genre = genre
                    break
        df_B100_songs.loc[i, 'genre'] = most_common_genre
    except:
        how_many_passes += 1
        pass  # didn't have any genres (or other error), move on


df_B100_songs.to_pickle('df_B100_songs_AF_COMPLETE.pickle')
```

```
0
100 200 300 400 500 600 700 800 900 1000
1100 1200 1300 1400 1500 1600 1700 1800 1900 2000
2100 2200 2300 2400 2500 2600 2700 2800 2900 3000
3100 3200 3300 3400 3500 3600 3700 3800 3900 4000
4100 4200 4300 4400 4500 4600 4700 4800 4900 5000
5100 5200 5300 5400 5500 5600 5700 5800 5900 6000
6100 6200 6300 6400 6500 6600 6700 6800 6900 7000
7100 7200 7300 7400 7500 7600 7700 7800 7900 8000
8100 8200 8300 8400 8500 8600 8700 8800 8900 9000
9100 9200 9300 9400 9500 9600 9700 9800 9900 10000
10100 10200 10300 10400 10500 10600 10700 10800 10900 11000
11100 11200 11300 11400 11500 11600 11700 11800 11900 12000
12100 12200 12300 12400 12500 12600 12700 12800 12900 13000
13100 13200 13300 13400 13500 13600 13700 13800 13900 14000
14100 14200 14300 14400 14500 14600 14700 14800 14900 15000
15100 15200 15300 15400 15500 15600 15700 15800 15900 16000
16100 16200 16300 16400 16500 16600 16700 16800 16900 17000
17100 17200 17300 17400 17500 17600 17700 17800 17900 18000
18100 18200 18300 18400 18500 18600 18700 18800 18900 19000
19100 19200 19300 19400 19500 19600 19700 19800 19900 20000
20100 20200 20300 20400 20500 20600 20700 20800 20900 21000
21100 21200 21300 21400 21500 21600 21700 21800 21900 22000
22100 22200 22300 22400 22500 22600 22700 22800 22900 23000
23100 23200 23300 23400 23500 23600 23700 23800 23900 24000
24100 24200 24300 24400 24500 24600 24700 24800 24900 25000
25100 25200 25300 25400 25500 25600 25700 25800 25900 26000
26100 26200 26300 26400 26500 26600 26700 26800 26900 27000
27100 27200 27300 27400 27500 27600 27700 27800 27900 28000
28100 28200 28300 28400 28500 28600 28700 28800 28900 29000
29100 29200 29300 29400 29500 29600 Wall time: 17min 11s
```

In [190…

```
# started 5:20
df_B100_songs.to_pickle('df_B100_songs_AF_TEMP.pickle')

# how far did we get - Run 4
passes_counter = (how_many_passes, counter)
passes_counter
```

Out[190…

```
(1524, 29681)
```

In [192…

```
(
    df_B100_songs.shape[0],
    df_B100_songs.shape[0] - df_B100_songs.key.isnull().sum(),
    df_B100_songs.shape[0] - df_B100_songs.genre.isnull().sum()
)
# all, with AF, with genre
```

Out[192…

```
(29681, 21277, 19756)
```

In [193…

```
df_B100_songs.to_pickle('df_B100_songs_POST_API.pickle')
```

STEP 3: Merge, Clean, and Save Datasets

In [221…

```
%%time
# reimport data
df_10M = pd.read_pickle('init_df_10M.pickle')
df_B100 = pd.read_pickle('init_df_B100.pickle')
df_B100_songs = pd.read_pickle('df_B100_songs_POST_API.pickle')
```

```
Wall time: 6.63 s
```

In [222…

```
dtypes = {
    'key': 'Int16', 'mode': 'Int16', 'time_signature': 'Int16', 'tempo': 'float32',
    'acousticness': 'float32', 'danceability': 'float32', 'duration_ms': 'Int64',
    'energy': 'float32', 'instrumentalness': 'float32', 'liveness': 'float32',
    'loudness': 'float32', 'speechiness': 'float32', 'valence': 'float32'
}
df_B100_songs = df_B100_songs.astype(dtypes)
```

In [223…

```
%%time
# drop rows in df_10M matching B100 songs
B100_song_id = set(df_B100_songs[~df_B100_songs.id.isna()].id)
df_10M = (
    df_10M[~df_10M.id.isin(B100_song_id)]
    .sort_values('release_date')
    .drop_duplicates(subset=['id'])
    .sort_values('artist')
    .reset_index(drop=True)
)
```

```
Wall time: 45.9 s
```

In [224…

```
df_10M.shape
```

Out[224…

```
(9577395, 18)
```

In [225…

```
%%time
# also remove partial matches (based on cleaned text, alt versions, etc)

# add approx song and artist columns for approx match
df_10M['approx_song'] = df_10M[['song']].applymap(clean_text)
df_10M['approx_artist'] = df_10M[['artist']].applymap(clean_text)

# add approx song and artist columns to B100 copy() for merging
df_TEMP = df_B100_songs.copy()
df_TEMP['approx_song'] = df_TEMP[['song']].applymap(clean_text)
df_TEMP['approx_artist'] = df_TEMP[['artist']].applymap(clean_text)
df_TEMP['REMOVE'] = 1
df_TEMP = df_TEMP[['approx_song', 'approx_artist', 'REMOVE']]

# remove partial matches
df_10M = df_10M.merge(df_TEMP, on=['approx_song', 'approx_artist'], how='left')
df_10M = df_10M[df_10M.REMOVE != 1].reset_index(drop=True)
```

```
Wall time: 2min 36s
```

In [226…

```
df_10M.shape
```

Out[226…

```
(9410481, 21)
```

In [227…

```
# fix formatting
formatting = [
    'id', 'song', 'artist', 'genre', 'release_date',
    'acousticness', 'danceability', 'duration_ms',
    'energy', 'instrumentalness', 'key', 'liveness', 'loudness',
    'mode', 'speechiness', 'tempo','time_signature', 'valence'
]

df_B100_songs = df_B100_songs[formatting]
```

In [277…

```
# drop duplicate ID and null values from df_B100_songs
# NOTE: formerly this would be df_B100_songs_AF, redundant data
df_B100_songs = df_B100_songs[~df_B100_songs.key.isna()]
df_B100_songs = df_B100_songs.drop_duplicates(subset = ['id']).dropna(subset = ['id']).reset_index(drop=True)
```

In [229…

```
# add in a flag from whether or not the song is in the B100
df_B100_songs['in_B100'] = True
df_10M['in_B100'] = False

# concat B100 songs at end of df_10M (all songs w data)
df_10M = pd.concat([df_10M, df_B100_songs]).sort_values('artist').reset_index(drop=True)
```

In [252…

```
# FINAL DATASET: df_10M pickle file
df_10M = df_10M.drop(['approx_song', 'approx_artist', 'REMOVE'], axis=1)
df_10M.to_pickle('df_10M.pickle')
```

In [231…

```
# FINAL DATASET: df_B100_songs pickle file
df_B100_songs = df_B100_songs[formatting]  # get rid of the in_B100 columns
df_B100_songs.to_pickle('df_B100_songs.pickle')
```

In [283…

```
# merge B100 songs into B100
# EXACT match only
df_B100 = df_B100.merge(df_B100_songs, on=['song', 'artist'], how='left')
```

In [284…

```
# confirm no songs in B100 songs, not in B100
len(set(df_B100_songs.id)), len(set(df_B100.id))
# looks good (only NA in B100)
```

Out[284…

```
(21274, 21275)
```

In [285…

```
# FINAL DATASET: df_B100 pickle file
df_B100.to_pickle('df_B100.pickle')
```

check datasets

In [253…

```
# data types
pd.concat(
    [df_10M.dtypes, df_B100.dtypes, df_B100_songs.dtypes],
    keys=['df_10M.dtypes', 'df_B100.dtypes', 'df_B100_songs.dtypes'],
    axis=1
)
```

Out[253…

	df_10M.dtypes	df_B100.dtypes	df_B100_songs.dtypes
id	object	object	object
song	object	object	object
artist	object	object	object
genre	object	object	object
release_date	datetime64[ns]	datetime64[ns]	datetime64[ns]
acousticness	float32	float32	float32
danceability	float32	float32	float32
duration_ms	Int64	Int64	Int64
energy	float32	float32	float32
instrumentalness	float32	float32	float32
key	Int16	Int16	Int16
liveness	float32	float32	float32

	df_10M.dtypes	df_B100.dtypes	df_B100_songs.dtypes
loudness	float32	float32	float32
mode	Int16	Int16	Int16
speechiness	float32	float32	float32
tempo	float32	float32	float32
time_signature	Int16	Int16	Int16
valence	float32	float32	float32
in_B100	bool	NaN	NaN
date	NaN	datetime64[ns]	NaN
rank	NaN	Int16	NaN
last-week	NaN	Int16	NaN
peak-rank	NaN	Int16	NaN
weeks-on-board	NaN	Int16	NaN

In [254…

```
df_10M.describe().loc['mean':'max'].T
```

Out[254…

	mean	std	min	25%	50%	75%	max
acousticness	0.42	0.37	0.00	0.03	0.34	0.82	1.00
danceability	0.53	0.19	0.00	0.40	0.54	0.68	1.00
duration_ms	238_311.97	156_917.57	1_000.00	169_587.00	216_933.00	275_400.00	6_072_187.00
energy	0.55	0.28	0.00	0.31	0.57	0.79	1.00
instrumentalness	0.26	0.37	0.00	0.00	0.00	0.66	1.00
key	5.24	3.54	0.00	2.00	5.00	8.00	11.00
liveness	0.21	0.18	0.00	0.10	0.13	0.26	1.00
loudness	-11.00	6.34	-60.00	-13.72	-9.21	-6.40	7.23
mode	0.66	0.47	0.00	0.00	1.00	1.00	1.00
speechiness	0.10	0.14	0.00	0.04	0.05	0.08	0.97
tempo	118.62	30.87	0.00	95.04	118.91	137.49	249.99
time_signature	3.84	0.57	0.00	4.00	4.00	4.00	5.00
valence	0.47	0.28	0.00	0.23	0.47	0.71	1.00

In [255…

```
df_B100.describe().loc['mean':'max'].T
```

Out[255…

	mean	std	min	25%	50%	75%	max
rank	50.50	28.87	1.00	26.00	51.00	76.00	100.00
last-week	47.59	28.05	1.00	23.00	47.00	72.00	100.00
peak-rank	40.97	29.35	1.00	13.00	38.00	65.00	100.00
weeks-on-board	9.16	7.62	1.00	4.00	7.00	13.00	90.00
acousticness	0.28	0.27	0.00	0.04	0.18	0.47	1.00
danceability	0.60	0.15	0.00	0.50	0.61	0.71	0.99
duration_ms	226_926.40	65_973.00	37_013.00	183_560.00	221_400.00	258_533.00	1_292_293.00
energy	0.63	0.20	0.02	0.48	0.64	0.79	1.00
instrumentalness	0.03	0.13	0.00	0.00	0.00	0.00	0.99
key	5.22	3.56	0.00	2.00	5.00	8.00	11.00
liveness	0.19	0.16	0.01	0.09	0.13	0.24	1.00

	mean	std	min	25%	50%	75%	max
loudness	-8.59	3.59	-29.52	-10.98	-8.11	-5.78	2.29
mode	0.73	0.44	0.00	0.00	1.00	1.00	1.00
speechiness	0.06	0.07	0.00	0.03	0.04	0.06	0.94
tempo	120.41	27.94	0.00	99.79	118.96	136.17	241.01
time_signature	3.94	0.29	0.00	4.00	4.00	4.00	5.00
valence	0.61	0.24	0.00	0.42	0.63	0.81	0.99

In [256…
```
df_B100_songs.describe().loc['mean':'max'].T
```

Out[256…

	mean	std	min	25%	50%	75%	max
acousticness	0.32	0.29	0.00	0.05	0.22	0.56	1.00
danceability	0.59	0.15	0.00	0.49	0.60	0.70	0.99
duration_ms	217_614.29	67_767.96	37_013.00	169_707.00	210_533.00	251_260.25	1_292_293.00
energy	0.61	0.20	0.02	0.46	0.62	0.77	1.00
instrumentalness	0.04	0.14	0.00	0.00	0.00	0.00	0.99
key	5.20	3.55	0.00	2.00	5.00	8.00	11.00
liveness	0.19	0.16	0.01	0.09	0.13	0.25	1.00
loudness	-8.90	3.62	-29.52	-11.32	-8.52	-6.08	2.29
mode	0.74	0.44	0.00	0.00	1.00	1.00	1.00
speechiness	0.07	0.07	0.00	0.03	0.04	0.06	0.94
tempo	120.50	28.19	0.00	99.64	119.00	136.73	241.01
time_signature	3.93	0.33	0.00	4.00	4.00	4.00	5.00
valence	0.61	0.24	0.00	0.42	0.63	0.81	0.99

In [257…
```
# Date Range for Billboard Hot 100
df_B100.date.min(), df_B100.date.max()
```

Out[257…
```
(Timestamp('1958-08-04 00:00:00'), Timestamp('2021-11-06 00:00:00'))
```

In [286…
```
# Genre Counts, AF counts
df_B100_songs.genre.count(), df_B100_songs.shape[0]
```

Out[286…
```
(19750, 21274)
```

In [287…
```
# percentage of songs with audio features that also have genre data
# NOTE: not all songs on Spotify have an associated genre
df_B100_songs.genre.count() / df_B100_songs.valence.count()
```

Out[287…
```
0.9283632603177587
```

In [291…
```
# number of total songs that include audio features with and without genre
df_10M.genre.count(), df_10M.shape[0], df_10M.genre.count() / df_10M.shape[0]
```

Out[291…
```
(6320843, 9431755, 0.6701661567757008)
```

In [292…
```
# All Billboard 100 lists
# number not null, total, proportion not null
(
    df_B100[df_B100.id.notnull()].shape[0],
    df_B100.shape[0],
    df_B100[df_B100.id.notnull()].shape[0] / df_B100.shape[0]
)
```

Out[292...
```
(243051, 330087, 0.7363240600205401)
```

DISCUSSION: Outliers

In [293...
```
# yep, I checked, that song is actually 37 seconds...
df_B100_songs[df_B100_songs.duration_ms == df_B100_songs.duration_ms.min()]
```

Out[293...

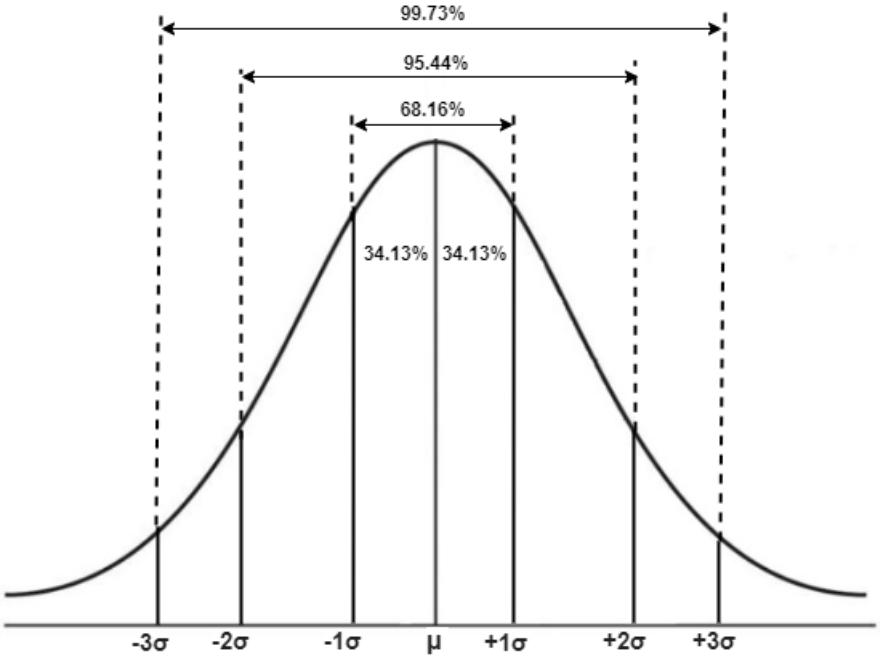
	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	l
10556	4IluCotvqijraSdnVLaFnM	Beautiful Trip	Kid Cudi	hip hop	2020-12-11	0.97	0.33	37013	0.51	0.95	11	0.88	

In [294...
```
# yep, I checked, that song is actually 21 minutes...
df_B100_songs[df_B100_songs.duration_ms == df_B100_songs.duration_ms.max()]
```

Out[294...

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness
14649	6pN3ra1mEPtjFsdCvDDHW3	Get Ready	Rare Earth	classic rock	1969-09-30	0.00	0.41	1292293	0.87	0.53	0	0.83

not exactly zscore, but a guess



In [263...
```
outside = 1 - .9973
outside/2, 1 - outside/2
```

Out[263...
```
(0.0013500000000000179, 0.99865)
```

In [295...
```
# IQR doesn't work for outliers (it excludes almost none)
# but using Z=3, this "song" would be excluded
df_B100_songs.describe(percentiles=[0.00135, 0.999]).loc['mean':'max'].T

# OUTLIERS SHOULD BE INVESTIGATED IN MORE DETAIL, BEFORE CLUSTERING
```

Out[295...

	mean	std	min	0.14%	50%	99.9%	max
acousticness	0.32	0.29	0.00	0.00	0.22	0.98	1.00

	mean	std	min	0.14%	50%	99.9%	max
danceability	0.59	0.15	0.00	0.16	0.60	0.96	0.99
duration_ms	217_614.29	67_767.96	37_013.00	95_309.39	210_533.00	677_275.35	1_292_293.00
energy	0.61	0.20	0.02	0.06	0.62	0.99	1.00
instrumentalness	0.04	0.14	0.00	0.00	0.00	0.95	0.99
key	5.20	3.55	0.00	0.00	5.00	11.00	11.00
liveness	0.19	0.16	0.01	0.02	0.13	0.98	1.00
loudness	-8.90	3.62	-29.52	-23.03	-8.52	-1.16	2.29
mode	0.74	0.44	0.00	0.00	1.00	1.00	1.00
speechiness	0.07	0.07	0.00	0.02	0.04	0.59	0.94
tempo	120.50	28.19	0.00	60.85	119.00	207.99	241.01
time_signature	3.93	0.33	0.00	1.00	4.00	5.00	5.00
valence	0.61	0.24	0.00	0.04	0.63	0.98	0.99

In [296…
```
df_10M.query('in_B100 == True').sample(5)
```

Out[296…

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	ke
3976753	5pxI50SA8WciKebbTFfAAA	Livingston Saturday Night	Jimmy Buffett	classic rock	1978-01-01	0.23	0.61	192867	0.88	0.78	
8464516	5BE7v9I2FUjUnObRAcopIs	You Know What I Mean	The Turtles	folk rock	1967-01-01	0.74	0.30	122400	0.64	0.00	1
2028020	5yGTQzYbEdY6B9RFZJypgt	Rhythm Of The Night	Debarge	soul	1985-01-01	0.08	0.71	229107	0.76	0.00	1
5673182	7bdUKJBcNob37UCRAs1wC6	Out Of Mind Out Of Sight	Models	australian rock	2017-07-05	0.01	0.63	217467	0.77	0.00	
830455	3SdTKo2uVsxFblQjpScoHy	Stand By Me	Ben E. King	soul	1962-08-20	0.57	0.65	180056	0.31	0.00	

In [297…
```
df_10M.query('in_B100 == False').sample(5)
```

Out[297…

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalne
5404601	7v1ChixYWsmjprghoKlsgM	Through My Eyes	Matt Perrone	<NA>	2012-01-31	0.08	0.68	246600	0.79	0.(
852059	6Q6Jw7R3CzPIWfEiu1KWzn	Sing, Sing, Sing	Benny Goodman	adult standards	2010-10-10	0.91	0.58	505993	0.66	0.(
545857	2xeT0rKPFa9YV3MWnz2Blp	Decaying Development Rave	Aria Bare	<NA>	2020-05-01	0.02	0.72	469500	0.90	0.(
1715922	4F48BXHLvuxiB3CvOnHXPH	Wandering	Craig Taubman	judaica	2001-01-01	0.03	0.61	207533	0.81	0.(
6316550	2i7SoOM7HSWOyh89mXmnv1	All Or Nothing - Paul Birken Remix	Paul Birken	acid techno	2016-08-15	0.00	0.85	367054	0.69	0.(

In [ ]:
```
```

Attachment 2

DATA

Import

Import Modules

In [1]:
```
# import modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import time
import seaborn as sns
sns.set_theme()

# jupyter notebook full-width display
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

# pandas formatting
pd.set_option('display.float_format', '{:.3f}'.format)
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 200)
```

Import Data

In [2]:
```
# all songs with audio features (combined from 3 sources)
df_10M = pd.read_pickle('df_10M.pickle')

# all Billboard 100 lists, audio features included where possible
df_B100 = pd.read_pickle('df_B100.pickle')

# unique songs from the Billboard 100 lists, only songs with audio features included
df_B100_songs = pd.read_pickle('df_B100_songs.pickle')

# all unique songs from the Billboard 100 lists, only songs with genre included
df_B100_songs_genre = df_B100_songs.dropna().copy().reset_index(drop=True)
```

In [3]:
```
# Popularity (as defined as "has been on the Billboard Hot 100" list)
# now combine, categorise, and correlate datasets

set_B100_id = set(df_B100_songs.id)

temp_not_popular = df_10M[~df_10M.id.isin(set_B100_id)].copy().reset_index(drop=True)
temp_not_popular['POPULAR'] = False

temp_popular = df_B100_songs.copy()
temp_popular['POPULAR'] = True

df_popularity = pd.concat([temp_not_popular, temp_popular]).reset_index(drop=True)

del temp_not_popular
del temp_popular
```

In [4]:
```
# import genre count data
df_genre_counts = pd.read_pickle('df_genre.pickle').set_index('genre').rename({'genre_count': 'count'}, axis=1)
```

Data Description

Data Sources

The Billboard 100

https://en.wikipedia.org/wiki/Billboard_Hot_100

https://www.kaggle.com/datasets/dhruvildave/billboard-the-hot-100-songs

1.2M Songs with Metadata (csv)

https://www.kaggle.com/datasets/rodolfofigueroa/spotify-12m-songs

8+ M. Spotify Tracks, Genre, Audio Features (SQL)

https://www.kaggle.com/datasets/maltegrosse/8-m-spotify-tracks-genre-audio-features

Spotify API

https://developer.spotify.com/documentation/web-api/

https://developer.spotify.com/console/get-search-item

https://developer.spotify.com/console/get-audio-features-track/

https://developer.spotify.com/documentation/web-api/reference/#/operations/get-audio-features

Spotipy Library: https://spotipy.readthedocs.io/en/master/

Data Description and Discussion

- The Billboard 100 data did not include audio features. It was combined with audio features from the following sources:
  - 1.2M Songs with Metadata (csv format)
  - 8+ M. Spotify Tracks, Genre, Audio Features (SQLite format)
  - Spotify API data gathered via the library Spotipy
- Overall, audio features was gathered for approximately 75% of songs from the Billboard 100.
  - Some songs were excluded based on data repetition issues
    - Typically this was only hard to find songs with very similar names
    - For example searching for 'Metallica The Unforgiven' and 'Metallica The Unforgiven Part 2' yielded the same Spotify id
      - It was determined that excluding these songs was less error-prone than manually fixing the issues
      - Alternatively, we could have kept 1 song. In this case, there is up to a 50% chance that the song is mislabelled, so this option appeared less favourable than dropping both repeat instances.
- A Quality Assurance (QA) check was performed on the final dataset.
  - Audio features from 100 songs were gathered from the Spotify API and compared to the datasets listed above.
  - There were 3 non-trivial issues noted in 2 of the 100 songs:
    - Madonna Live To Tell
      - A significant increase in loudness (~7 dB)
      - Approximately 1 second difference in length
      - All other audio features consistent between data sources
      - Both of these changes appear to result from remastering and re-uploading the track
        - https://artists.spotify.com/help/article/re-uploading-music
    - Lil Wayne Let It All Work Out
      - The key signature was not consistent between the 2 sources
      - The newer source (the API request from Sept 11, 2022) was correct (B major)
      - The SQL database was also different
      - My supposition is that these errors are due to the characteristics of the song:
        - atonal (most notably the singing)
        - detuned (bass pitch automation, and low-fi detuning effects)
  - Overall, there is a large degree of consistency between datasets. Furthermore, inconsistencies are all explainable with reasonable suppositions.

Spotify API Audio Feature Descriptions

from: https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-audio-features

acousticness

number \

A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.

```
>= 0
<= 1
```

analysis_url

string

A URL to access the full audio analysis of this track. An access token is required to access this data.

danceability

number \

Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.

duration_ms

integer

The duration of the track in milliseconds.

energy

number \

Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.

id

string

The Spotify ID for the track.

instrumentalness

number \

Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.

key

integer The key the track is in. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1.

```
>= -1
<= 11
```

liveness

number \

Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.

loudness

number \

The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0 db.

mode

integer

Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.

speechiness

number \

Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.

tempo

number \

The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.

time_signature

integer

An estimated time signature. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure). The time signature ranges from 3 to 7 indicating time signatures of "3/4", to "7/4".

```
>= 3
<= 7
```

track_href

string

A link to the Web API endpoint providing full details of the track.

type

string

The object type.

Allowed value: "audio_features"

uri

string

The Spotify URI for the track.

valence

number \

A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

```
>= 0
<= 1
```

Descriptive Statistics and Data Features

In [5]:
```
# sizes of the datasets
df_10M.shape, df_B100.shape, df_B100_songs.shape, df_B100_songs_genre.shape
```

Out[5]:
```
((9431755, 19), (330087, 23), (21274, 18), (19750, 18))
```

In [6]:
```
# are any keys unknown (key == -1)
sorted(df_B100_songs['key'].unique()), sorted(df_10M['key'].unique())
# no
```

Out[6]:
```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

In [7]:
```
# data types
pd.concat(
    [df_10M.dtypes, df_B100.dtypes, df_B100_songs.dtypes],
    keys=['df_10M.dtypes', 'df_B100.dtypes', 'df_B100_songs.dtypes'],
    axis=1
)
```

Out[7]:

	df_10M.dtypes	df_B100.dtypes	df_B100_songs.dtypes
id	object	object	object
song	object	object	object
artist	object	object	object
genre	object	object	object
release_date	datetime64[ns]	datetime64[ns]	datetime64[ns]
acousticness	float32	float32	float32
danceability	float32	float32	float32
duration_ms	Int64	Int64	Int64
energy	float32	float32	float32
instrumentalness	float32	float32	float32
key	Int16	Int16	Int16
liveness	float32	float32	float32
loudness	float32	float32	float32
mode	Int16	Int16	Int16
speechiness	float32	float32	float32
tempo	float32	float32	float32
time_signature	Int16	Int16	Int16
valence	float32	float32	float32
in_B100	bool	NaN	NaN
date	NaN	datetime64[ns]	NaN
rank	NaN	Int16	NaN
last-week	NaN	Int16	NaN
peak-rank	NaN	Int16	NaN
weeks-on-board	NaN	Int16	NaN

In [8]:
```
# Date Range for Billboard Hot 100

df_B100.date.min(), df_B100.date.max()
```

Out[8]:
```
(Timestamp('1958-08-04 00:00:00'), Timestamp('2021-11-06 00:00:00'))
```

In [9]:
```
df_10M.describe().loc['mean':'max'].T
```

Out[9]:

	mean	std	min	25%	50%	75%	max
acousticness	0.422	0.373	0.000	0.033	0.337	0.819	0.996
danceability	0.527	0.191	0.000	0.395	0.544	0.675	1.000
duration_ms	238311.970	156917.572	1000.000	169587.000	216933.000	275400.000	6072187.000
energy	0.545	0.282	0.000	0.309	0.566	0.789	1.000
instrumentalness	0.260	0.375	0.000	0.000	0.002	0.657	1.000
key	5.238	3.542	0.000	2.000	5.000	8.000	11.000
liveness	0.209	0.180	0.000	0.096	0.129	0.262	1.000
loudness	-10.997	6.339	-60.000	-13.723	-9.208	-6.401	7.234
mode	0.661	0.473	0.000	0.000	1.000	1.000	1.000
speechiness	0.098	0.136	0.000	0.036	0.047	0.083	0.974
tempo	118.619	30.869	0.000	95.045	118.912	137.494	249.987
time_signature	3.838	0.569	0.000	4.000	4.000	4.000	5.000
valence	0.473	0.278	0.000	0.232	0.466	0.706	1.000

In [10]:
```
df_B100.describe().loc['mean':'max'].T
```

Out[10]:

	mean	std	min	25%	50%	75%	max
rank	50.501	28.866	1.000	26.000	51.000	76.000	100.000
last-week	47.592	28.054	1.000	23.000	47.000	72.000	100.000
peak-rank	40.971	29.347	1.000	13.000	38.000	65.000	100.000
weeks-on-board	9.162	7.618	1.000	4.000	7.000	13.000	90.000
acousticness	0.278	0.275	0.000	0.041	0.178	0.467	0.995
danceability	0.601	0.149	0.000	0.505	0.610	0.706	0.988
duration_ms	226926.400	65973.000	37013.000	183560.000	221400.000	258533.000	1292293.000
energy	0.626	0.198	0.018	0.482	0.644	0.787	0.996
instrumentalness	0.031	0.133	0.000	0.000	0.000	0.001	0.985
key	5.220	3.560	0.000	2.000	5.000	8.000	11.000
liveness	0.187	0.158	0.015	0.089	0.126	0.240	0.999
loudness	-8.588	3.591	-29.519	-10.975	-8.107	-5.783	2.291
mode	0.733	0.442	0.000	0.000	1.000	1.000	1.000
speechiness	0.064	0.069	0.000	0.032	0.040	0.060	0.941
tempo	120.413	27.945	0.000	99.789	118.957	136.166	241.009
time_signature	3.941	0.292	0.000	4.000	4.000	4.000	5.000
valence	0.606	0.237	0.000	0.421	0.628	0.806	0.989

In [11]:
```
df_B100_songs.describe().loc['mean':'max'].T
```

Out[11]:

	mean	std	min	25%	50%	75%	max
acousticness	0.316	0.290	0.000	0.053	0.224	0.555	0.995
danceability	0.589	0.152	0.000	0.490	0.597	0.697	0.988
duration_ms	217614.286	67767.959	37013.000	169707.000	210533.000	251260.250	1292293.000
energy	0.611	0.203	0.018	0.464	0.625	0.775	0.996
instrumentalness	0.036	0.144	0.000	0.000	0.000	0.001	0.985
key	5.198	3.553	0.000	2.000	5.000	8.000	11.000

	mean	std	min	25%	50%	75%	max
liveness	0.194	0.163	0.015	0.091	0.131	0.250	0.999
loudness	-8.904	3.615	-29.519	-11.323	-8.524	-6.079	2.291
mode	0.743	0.437	0.000	0.000	1.000	1.000	1.000
speechiness	0.066	0.075	0.000	0.032	0.040	0.061	0.941
tempo	120.504	28.186	0.000	99.644	119.003	136.727	241.009
time_signature	3.925	0.328	0.000	4.000	4.000	4.000	5.000
valence	0.610	0.240	0.000	0.423	0.635	0.814	0.989

In [12]:
```
df_B100_songs_genre.describe().loc['mean':'max'].T
```

Out[12]:

	mean	std	min	25%	50%	75%	max
acousticness	0.309	0.288	0.000	0.051	0.215	0.539	0.995
danceability	0.589	0.152	0.000	0.491	0.597	0.697	0.988
duration_ms	219136.535	67455.535	37013.000	171881.250	212027.000	252236.750	1292293.000
energy	0.613	0.202	0.018	0.465	0.626	0.776	0.996
instrumentalness	0.033	0.135	0.000	0.000	0.000	0.001	0.985
key	5.206	3.556	0.000	2.000	5.000	8.000	11.000
liveness	0.193	0.162	0.015	0.091	0.131	0.249	0.999
loudness	-8.840	3.613	-29.029	-11.269	-8.433	-6.003	-0.380
mode	0.741	0.438	0.000	0.000	1.000	1.000	1.000
speechiness	0.067	0.075	0.000	0.032	0.040	0.061	0.941
tempo	120.602	28.203	0.000	99.692	119.042	136.998	231.028
time_signature	3.927	0.325	0.000	4.000	4.000	4.000	5.000
valence	0.605	0.240	0.000	0.416	0.626	0.808	0.989

In [13]:
```
# Genre Counts

# percentage of songs with audio features that also have genre data
# NOTE: not all songs on Spotify have an associated genre
df_B100.groupby(['song', 'artist']).count().shape[0], df_B100_songs.shape[0], df_B100_songs_genre.shape[0]

# all, AF, genre
```

Out[13]:
```
(29681, 21274, 19750)
```

In [14]:
```
# number of total songs which include audio features with and without genre
df_10M.genre.count(), df_10M.shape[0], df_10M.genre.count() / df_10M.shape[0]
```

Out[14]:
```
(6320843, 9431755, 0.6701661567757008)
```

Proportion of Songs With Audio Feature Data:

~75% of songs on the Billboard list are available on Spotify, and weren't removed for data errors

In [15]:
```
# All Billboard 100 lists
(
    df_B100[df_B100.id.notnull()].shape[0],
    df_B100.shape[0],
    df_B100[df_B100.id.notnull()].shape[0] / df_B100.shape[0]
)
# number not null, total, proportion not null
```

Out[15]:
```
(243051, 330087, 0.7363240600205401)
```

In [16]:
```
# All songs from Billboard 100 lists
(
    df_B100_songs.shape[0],
    df_B100.groupby(['song', 'artist']).count().shape[0],
    df_B100_songs.shape[0] / df_B100.groupby(['song', 'artist']).count().shape[0]
)
# number not null, total, proportion not null
```

Out[16]:
```
(21274, 29681, 0.716754826319868)
```

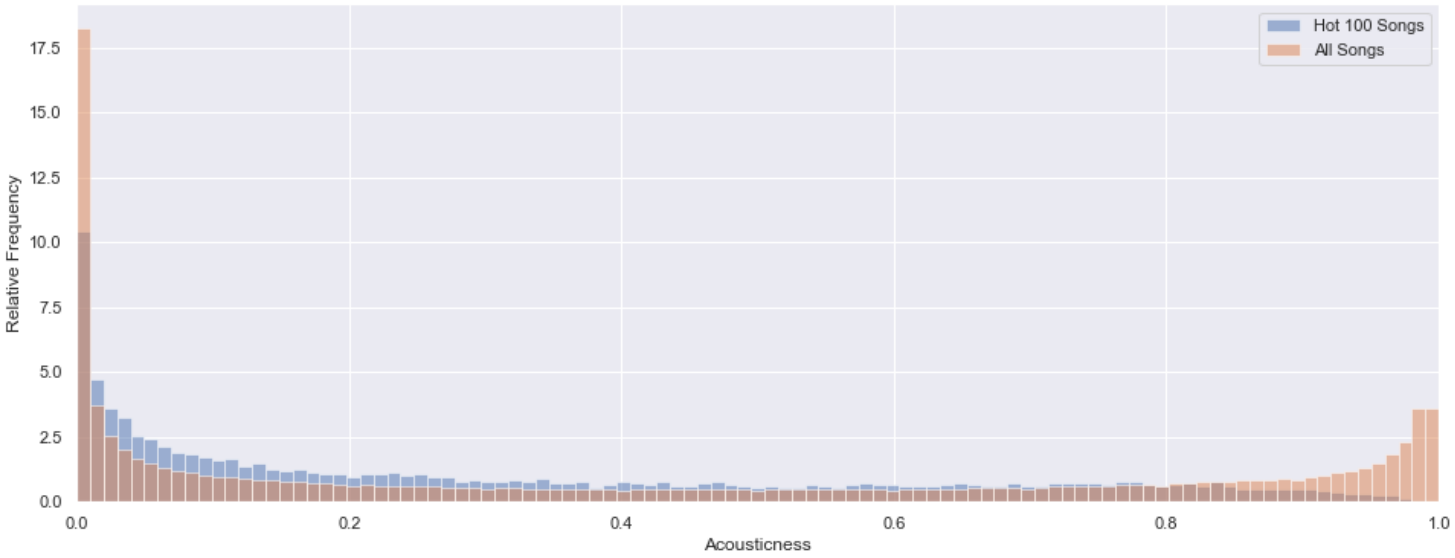
EXPLORATORY DATA ANALYSIS

Histograms

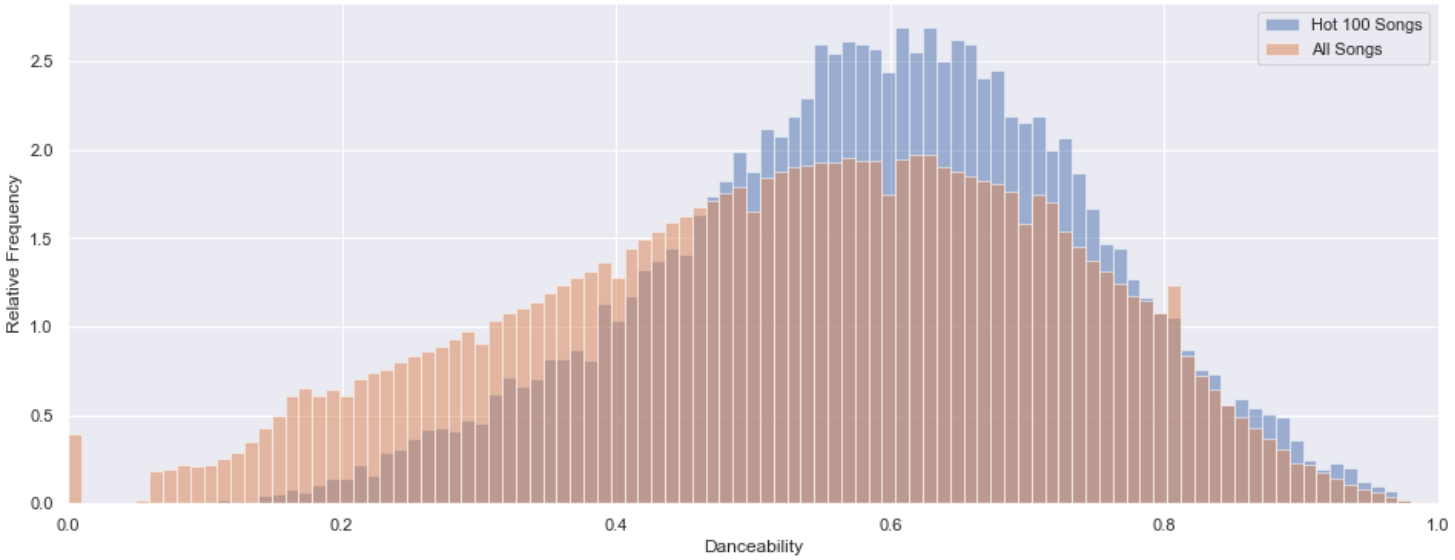
In [17]:
```
# main features
features = ['acousticness', 'danceability', 'energy', 'instrumentalness',
            'liveness', 'loudness', 'speechiness', 'tempo', 'valence']

def compare_histograms(feature, bins=50, logy=False, figsize=(16, 6)):

    plt.figure(figsize=figsize)

    # bins don't line up unless you do this
    if feature == 'loudness':
        xmin, xmax = -16, 0
        plt.xlim(xmin, xmax)  # reasonable range for loudness in music
        bins_plot = [(x/(bins+1))*(xmax-xmin)+xmin for x in range(bins+2)]
    elif feature == 'tempo':
        xmin, xmax = 40, 200
        plt.xlim(xmin, xmax)  # reasonable range for tempo in music
        bins_plot = [(x/(bins+1))*(xmax-xmin)+xmin for x in range(bins+2)]
    else:
        xmin, xmax = 0, 1
        plt.xlim(xmin, xmax)  # most audio features vary between 0 and 1
        bins_plot = [(x/(bins+1))*(xmax-xmin)+xmin for x in range(bins+2)]
        # NOTE: mode is an int, either 0 or 1 (minor/major),
        # but the average shows how commonly a song is in either mode

    # plots
    plt.hist(df_B100_songs[feature], bins_plot, alpha=0.5, label='Hot 100 Songs', density=True)
    plt.hist(df_10M[feature], bins_plot, alpha=0.5, label='All Songs', density=True)

    title = f'{feature.title()} Histogram'
#     plt.title(title)
    plt.xlabel(feature.title().replace('_', ' '))
    plt.legend(loc='upper right')
    if logy:
        plt.yscale('log')
        plt.ylabel('Log Relative Frequency')
    else:
        plt.ylabel('Relative Frequency')

    # save the image
    plt.savefig(f'figures/histograms/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

In [18]:
```
# Histograms of Features
for feature in features:
    compare_histograms(feature, 100, logy=(feature in ['duration_ms', 'instrumentalness', 'speechiness']))
```

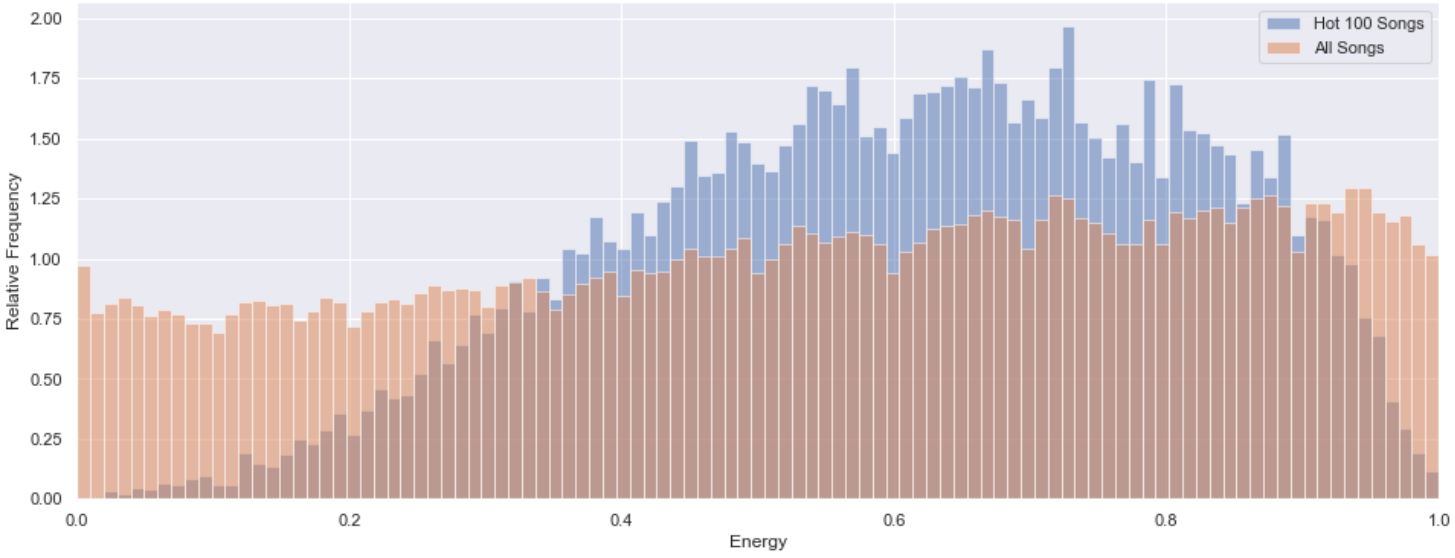
```
Acousticness Histogram
```



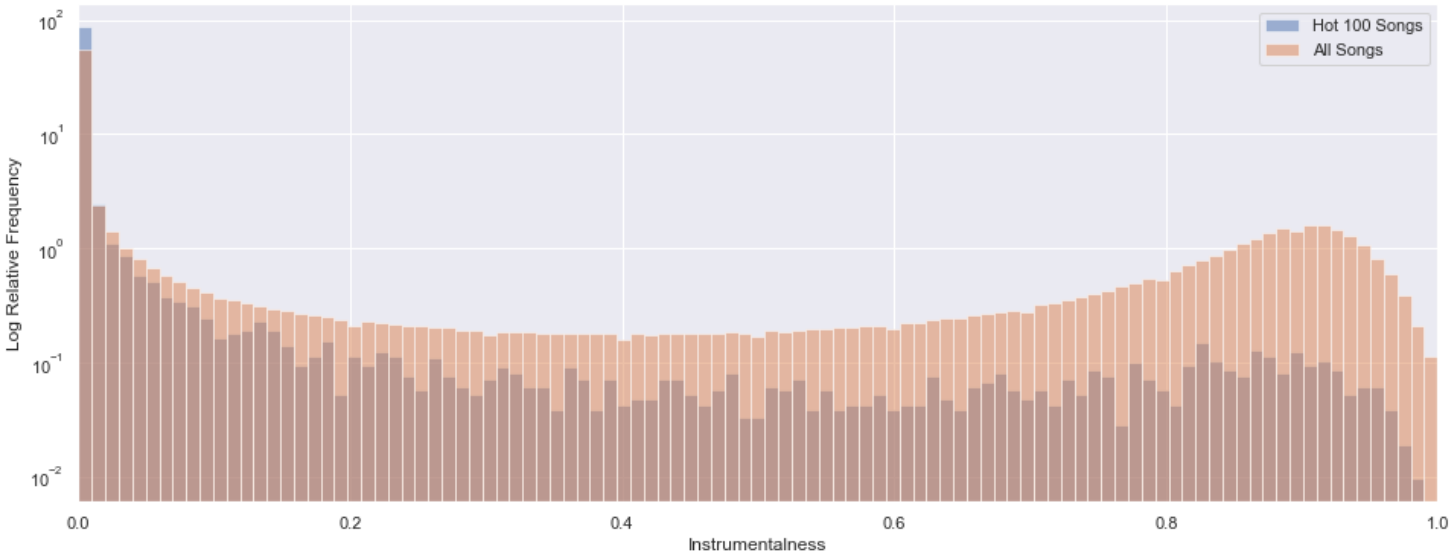
Danceability Histogram



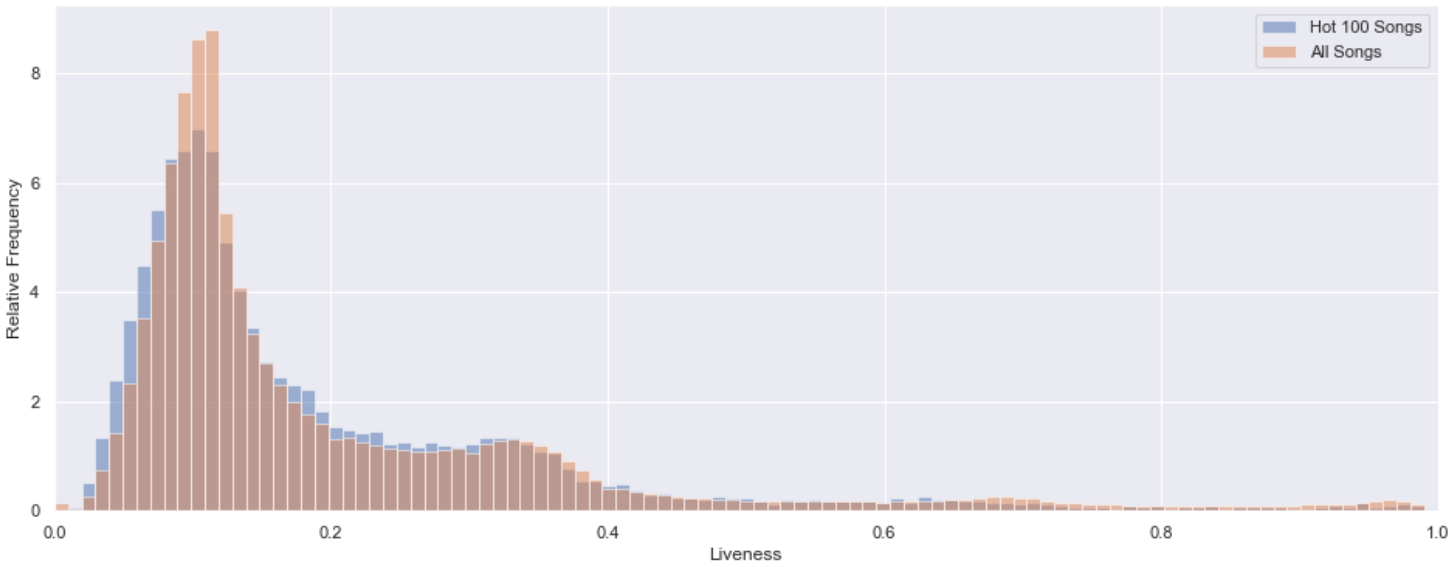
Energy Histogram



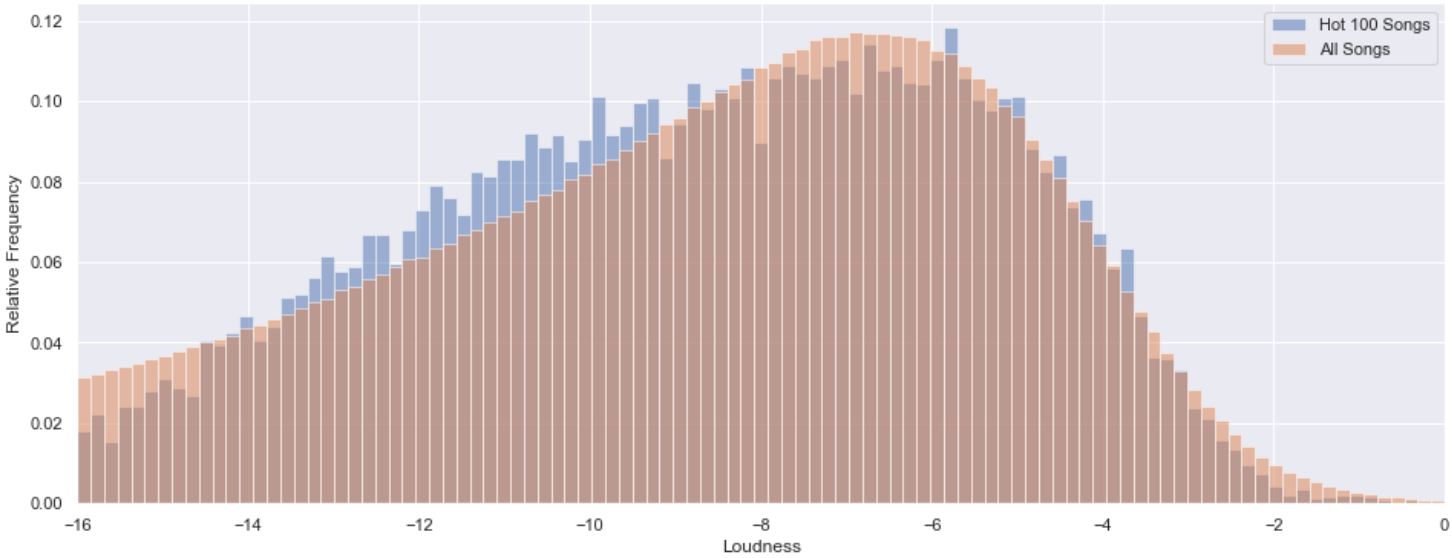
Instrumentalness Histogram



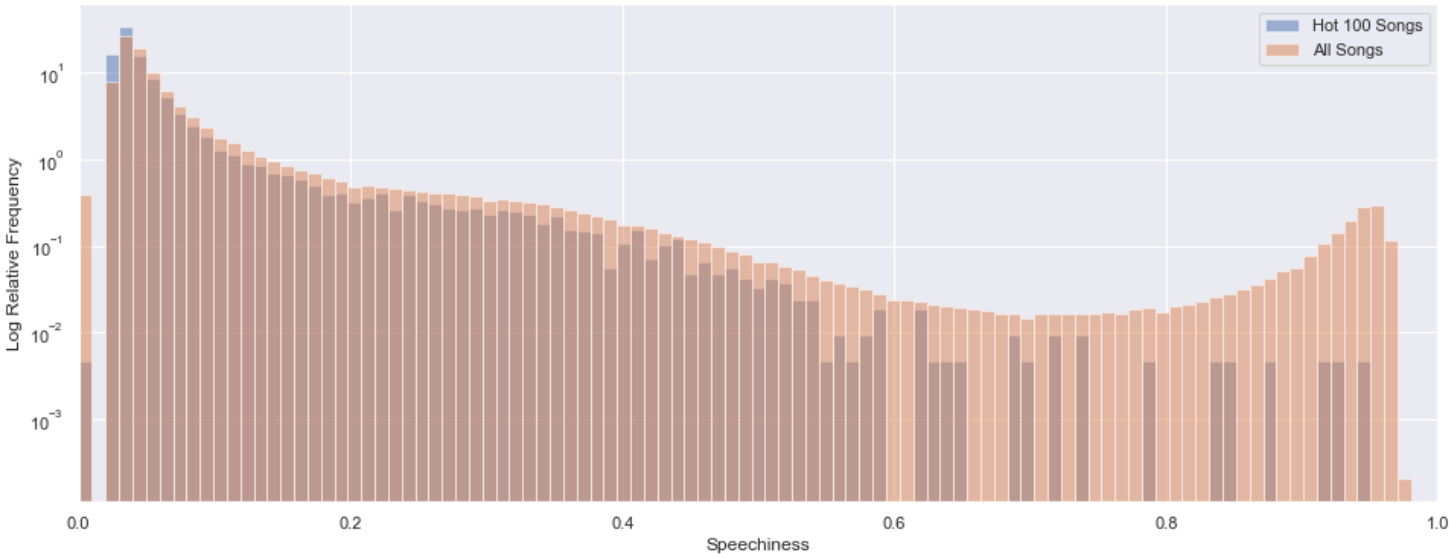
Liveness Histogram



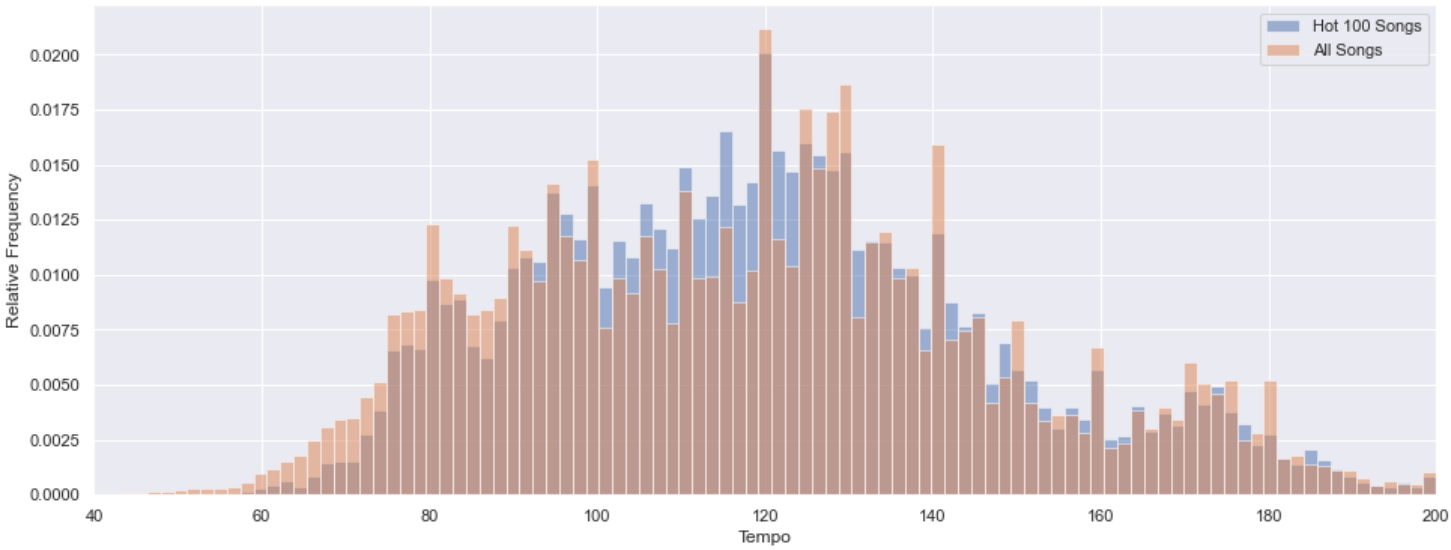
Loudness Histogram



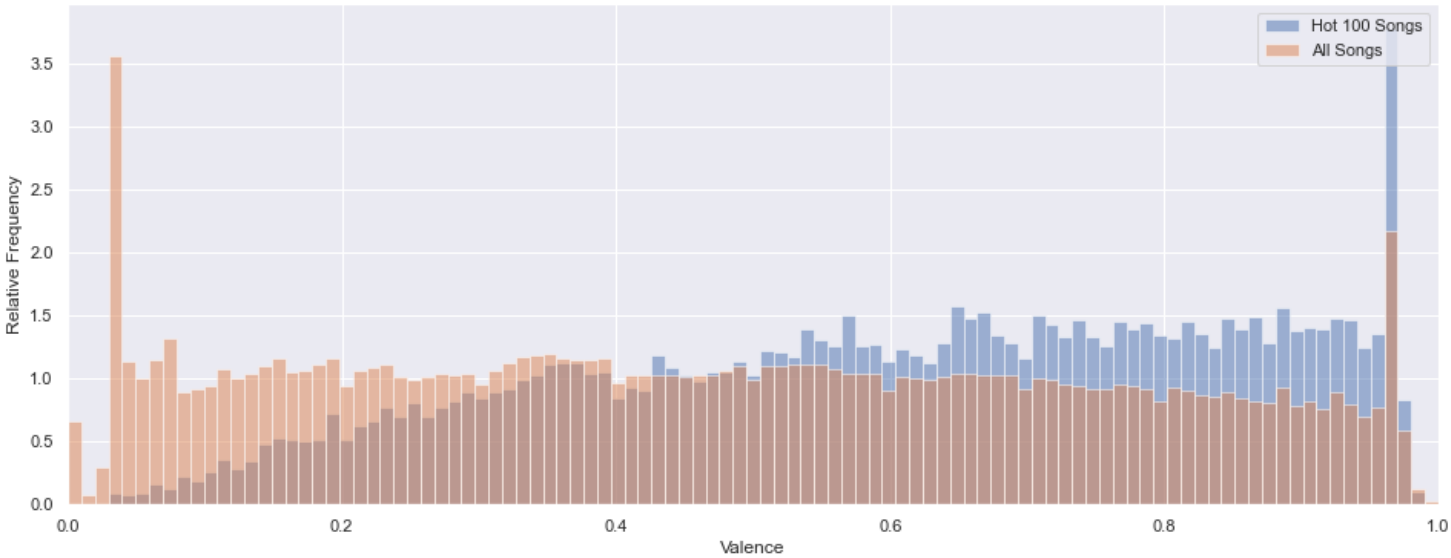
Speechiness Histogram



Tempo Histogram



Valence Histogram



In [19]:

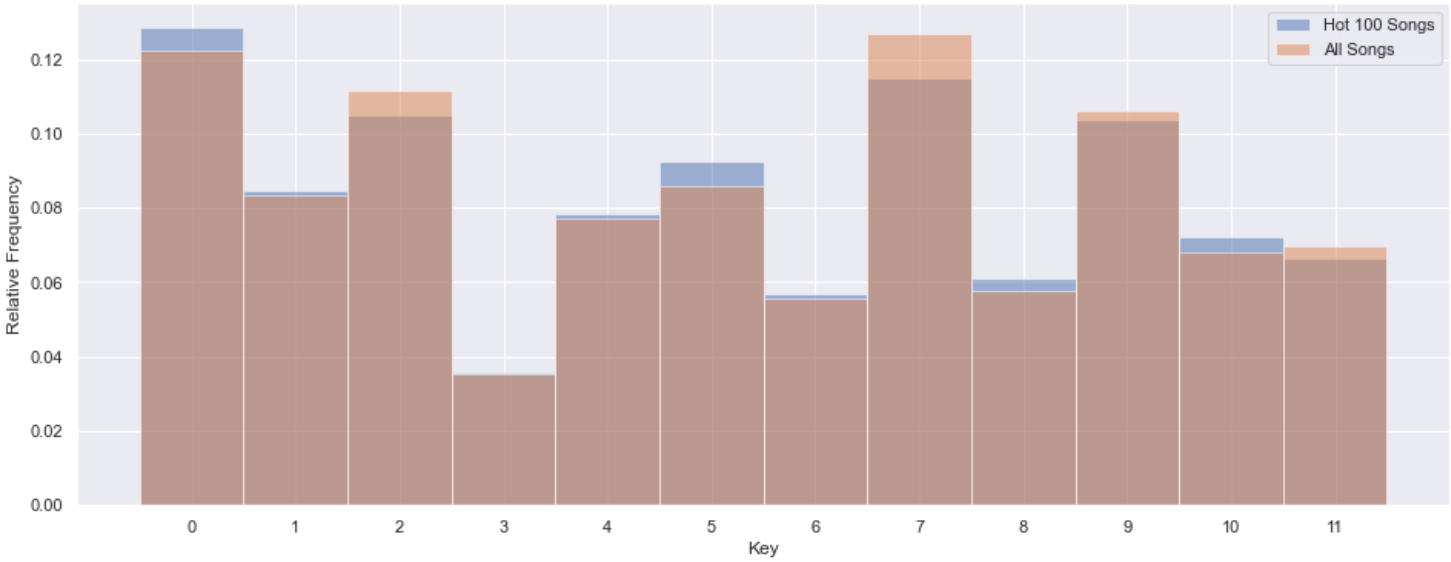
```
# features requiring specific bin sizes
other_features = ['key', 'mode', 'time_signature']

# Histograms of Features with Specific Bin Sizes
other_bins = {'key': 12, 'mode': 2, 'time_signature': 5}

def compare_histograms_discrete(feature, bins=50, logy=False, figsize=(16, 6)):
```

```
    fig, ax = plt.subplots(figsize=figsize)

    # use range(bins+1) to offset tick labels and line up with appropriate labels
    plt.hist(df_B100_songs[feature], [x for x in range(bins+1)], alpha=0.5, label='Hot 100 Songs', density=True, align='left')
    plt.hist(df_10M[feature], [x for x in range(bins+1)], alpha=0.5, label='All Songs', density=True, align='left')

    title = f'{feature.title()} Histogram'
#     plt.title(title)
    plt.xlabel(feature.title().replace('_', ' '))
    plt.legend(loc='upper right')
    if logy:
        plt.yscale('log')
        plt.ylabel('Log Relative Frequency')
    else:
        plt.ylabel('Relative Frequency')

    ax.set(xticks=[x for x in range(bins)])

    # save the image
    plt.savefig(f'figures/histograms/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

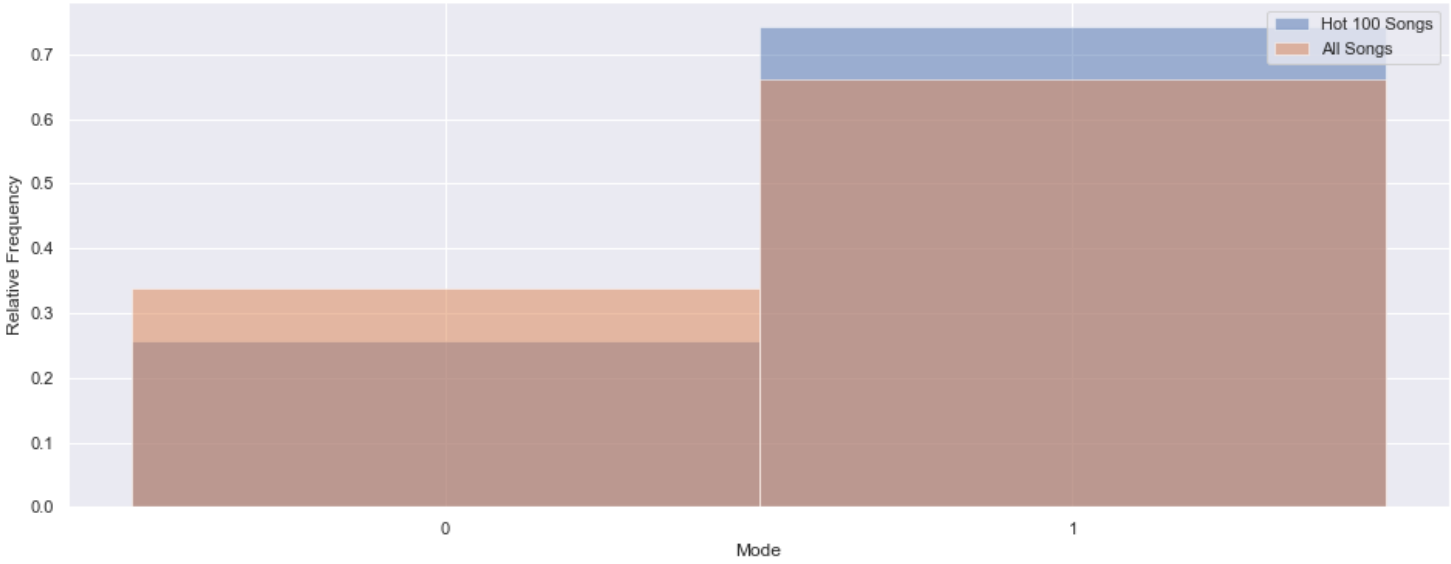
In [20]:

```
# Histograms of Features with Specific Bin Sizes

# Histograms of Features
for feature in other_features:
    compare_histograms_discrete(feature, other_bins[feature])
```

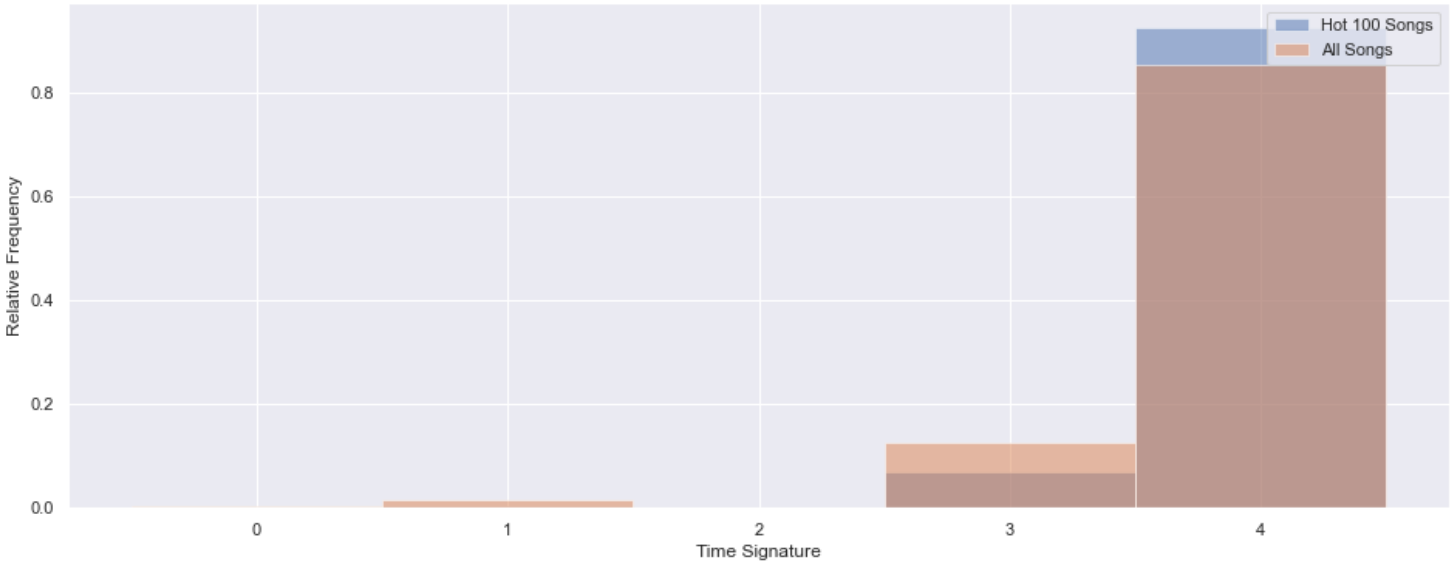
```
Key Histogram
```




```
Mode Histogram
```



Time_Signature Histogram

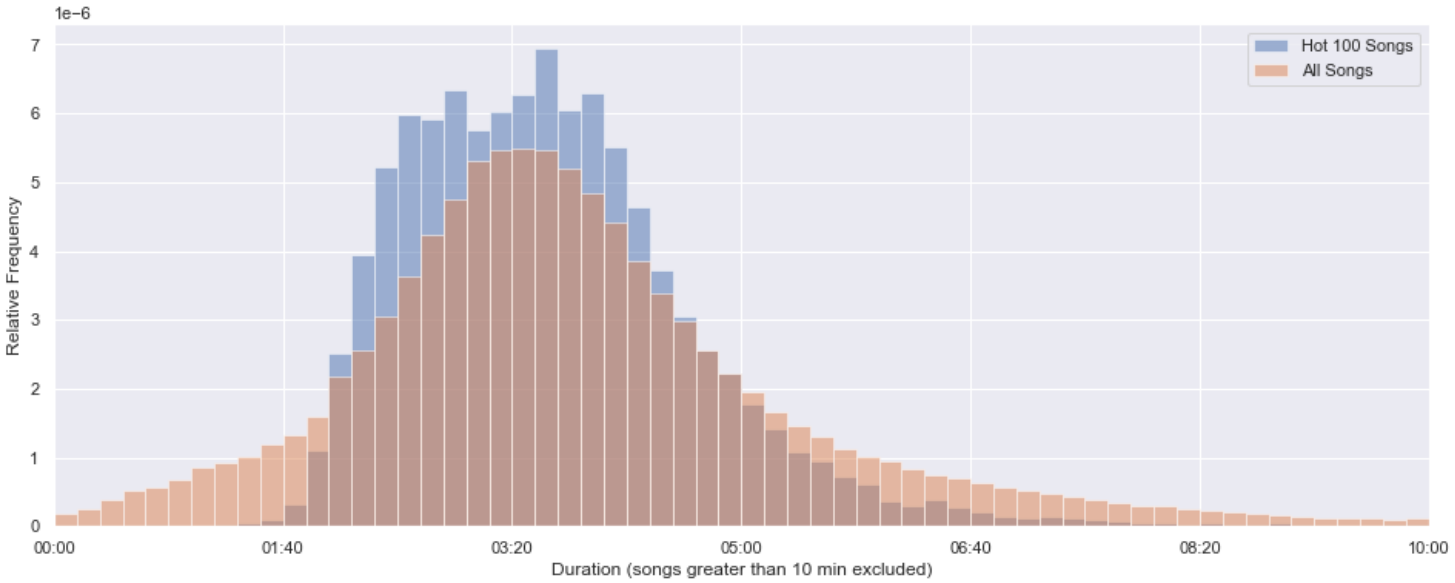


In [21]:

```
# duration_ms needs to share the x-axis

fig, ax = plt.subplots(figsize=(16, 6))

bins = 1000
maxx = df_10M['duration_ms'].max()
maxx = round(maxx, ndigits=-7)
increment = maxx / bins  # need to do this first or get integer wrap around error
bins = [x for x in range(bins) * increment]

plt.hist(df_B100_songs['duration_ms'], bins, alpha=0.5, label='Hot 100 Songs', density=True)
plt.hist(df_10M['duration_ms'], bins, alpha=0.5, label='All Songs', density=True)

title = 'Duration Histogram'
# plt.title(title)
plt.xlabel('Duration (songs greater than 10 min excluded)')
plt.legend(loc='upper right')
plt.ylabel('Relative Frequency')
ax.set_xlim((0, 600000))

# https://stackoverflow.com/questions/40395227/minute-and-second-format-for-x-label-of-matplotlib
formatter = mpl.ticker.FuncFormatter(lambda ms, x: time.strftime('%M:%S', time.gmtime(ms // 1000)))
ax.xaxis.set_major_formatter(formatter)

# save the image
plt.savefig(f'figures/histograms/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)
```

```
plt.show(feature.title())
```

```
Duration Histogram
```



Time Series

In [22]:

```
# audio features
main_features = [
    'acousticness', 'danceability', 'energy', 'instrumentalness',
    'liveness', 'loudness', 'mode', 'speechiness', 'tempo', 'valence'
]

# date filter: this affects all time series plotting functions below
# date_filter = 'release_date >= 1958'  # original filter, revised based on later section (see note below)
date_filter = '1958 <= release_date < 2021'
```

NOTE:

- The revised date_filter is based on data quality past 2020
- this is based on analysis later in this notebook: Time Series Counts - Preferred Date Filter
  - this analysis is later in the notebook because it was conducted to explain abberations in time series plots

In [23]:

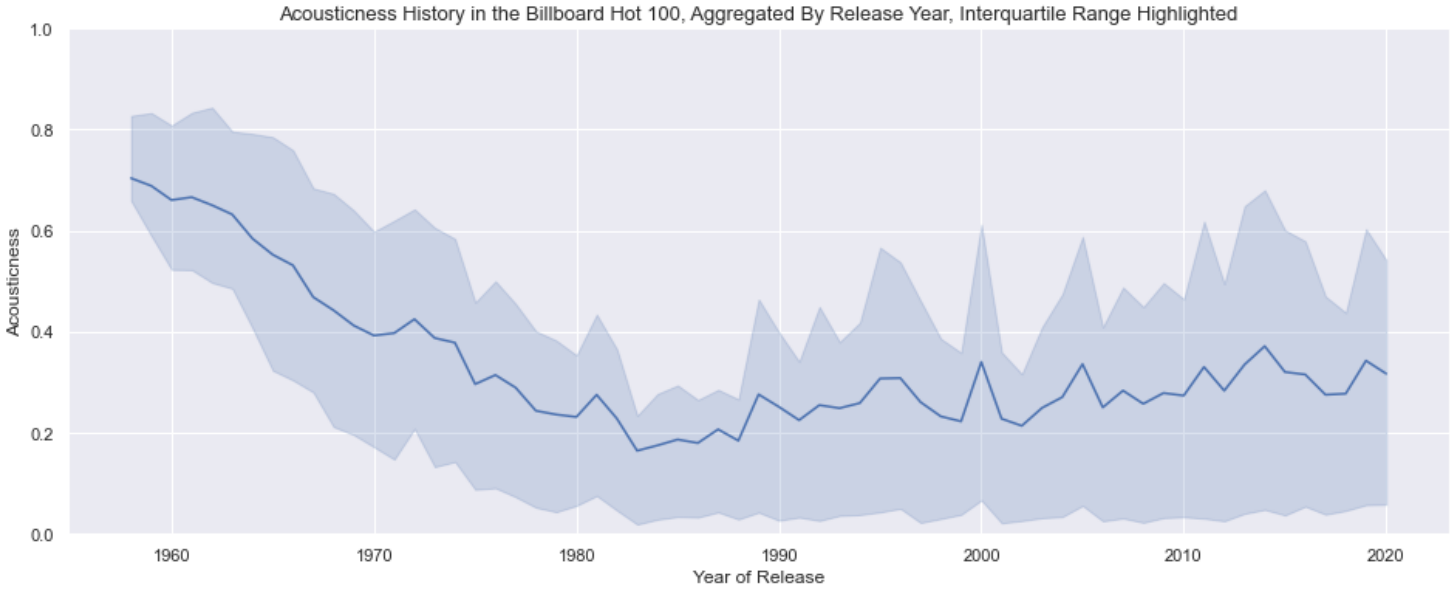
```
def plot_attribute_history(feature, dataframe=df_B100_songs, title=None, colour=0):
    """
    Uses songs df_B100_songs by default
    Plots Billboard Hot 100 audio features over time
    """
    # drop values before billboard 100 (lower quality data)
    temp_df = dataframe.query(date_filter)

    plt.figure(figsize=(16, 6))
    ax = sns.lineplot(
        x=temp_df.release_date.dt.year,
        y=temp_df[feature],
        color=sns.color_palette()[colour],
        errorbar=('pi', 50)  # middle 50% or IQR
    )

    if title:
        title=title
    else:
        title=f'{feature.capitalize()} History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highl

    if feature == 'loudness':
        plt.ylim(-16, 0)  # reasonable range for loudness in music
    elif feature == 'tempo':
        plt.ylim(40, 200)  # reasonable range for tempo in music
    else:
        plt.ylim(0, 1)  # most audio features vary between 0 and 1
```

```
        # NOTE: mode is an int, either 0 or 1 (minor/major),
        # but the average shows how commonly a song is in either mode

    plt.title(title, fontsize=13)
    plt.xlabel('Year of Release')
    plt.ylabel(feature.capitalize().replace('_', ' '))

    # save the image
    plt.savefig(f'figures/timeseries/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

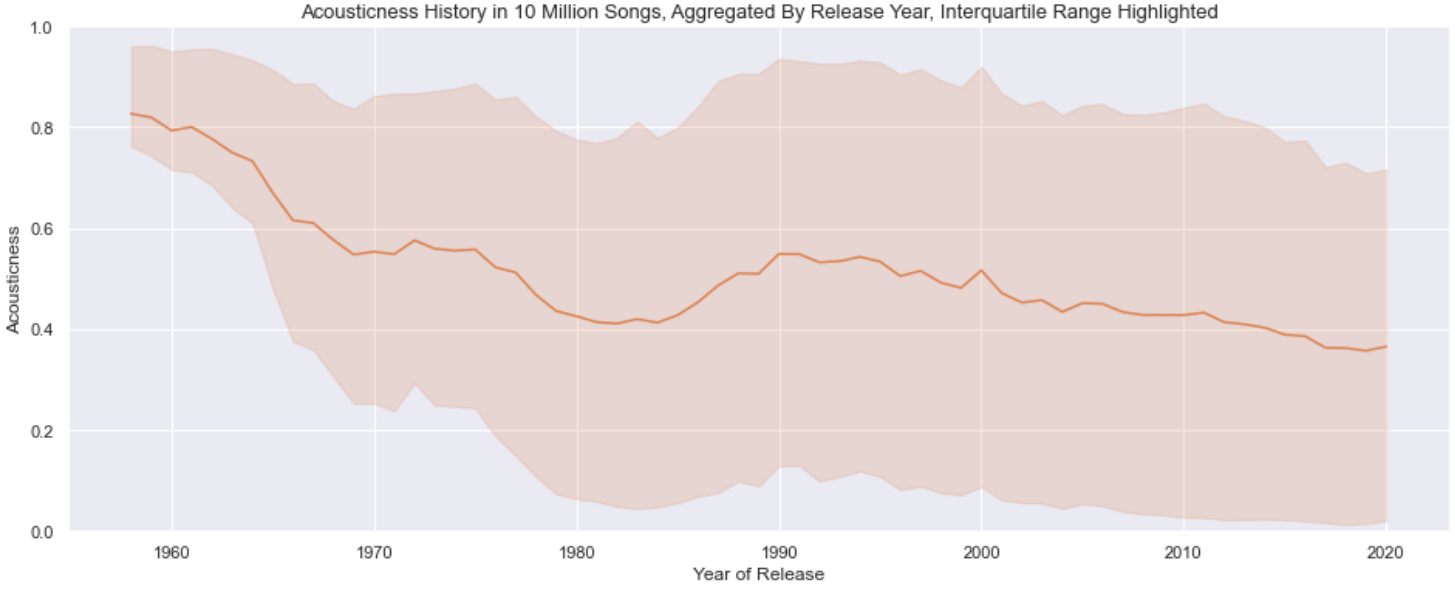
In [24]:

```
%%time
# plot features for Billboard Hot 100 and All Songs
for feature in main_features:
    # first plot the Billboard Hot 100
    plot_attribute_history(feature)
    # then plot the same feature from the large dataset
    title = f'{feature.capitalize()} History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted'
    plot_attribute_history(feature, dataframe=df_10M, title=title, colour=1)
```

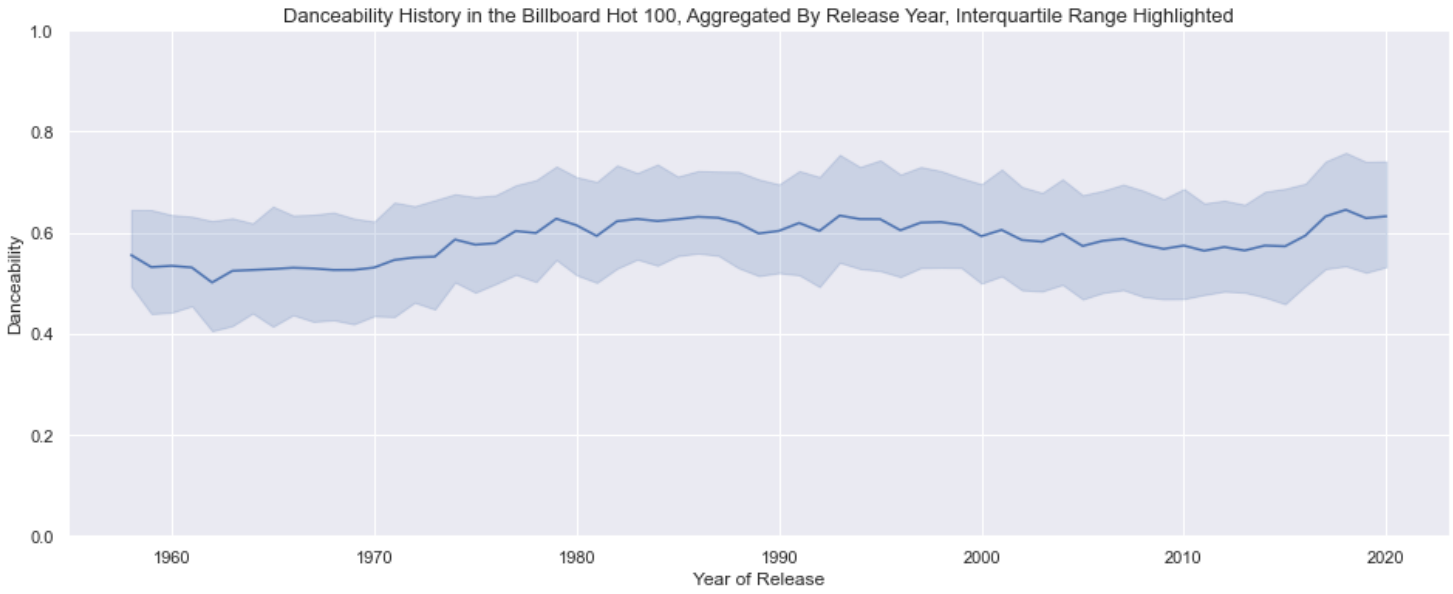
```
Acousticness History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted
```



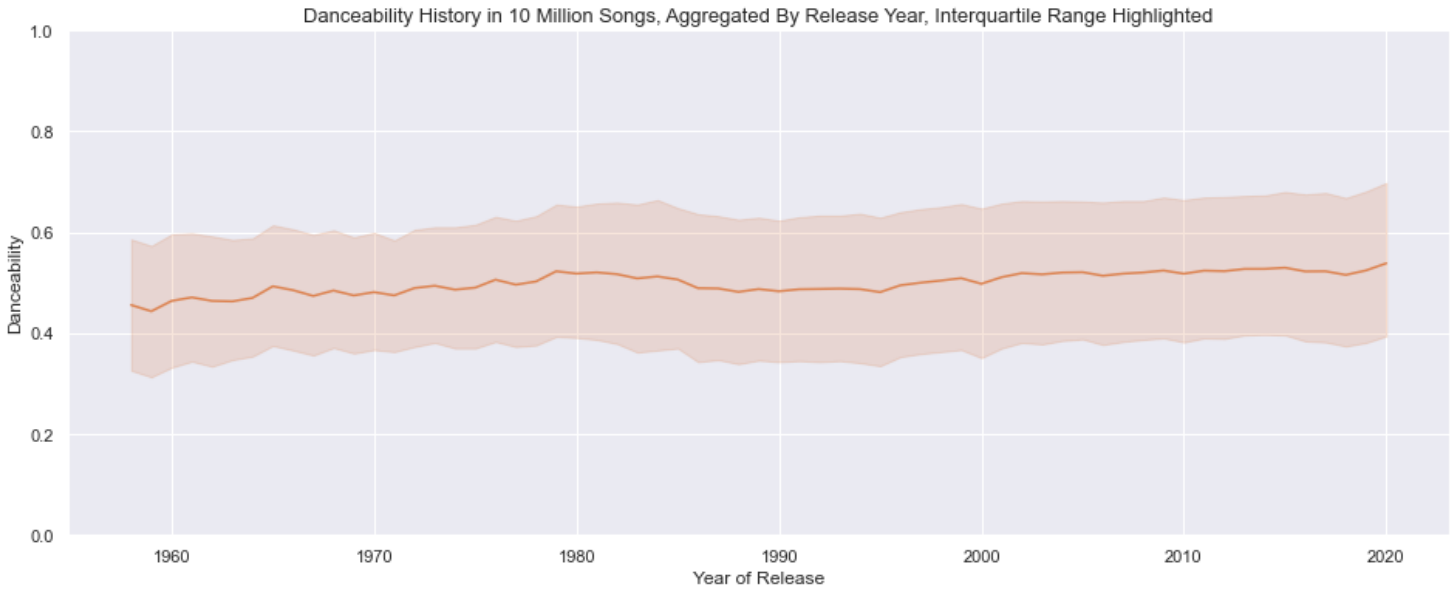

```
Acousticness History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted
```



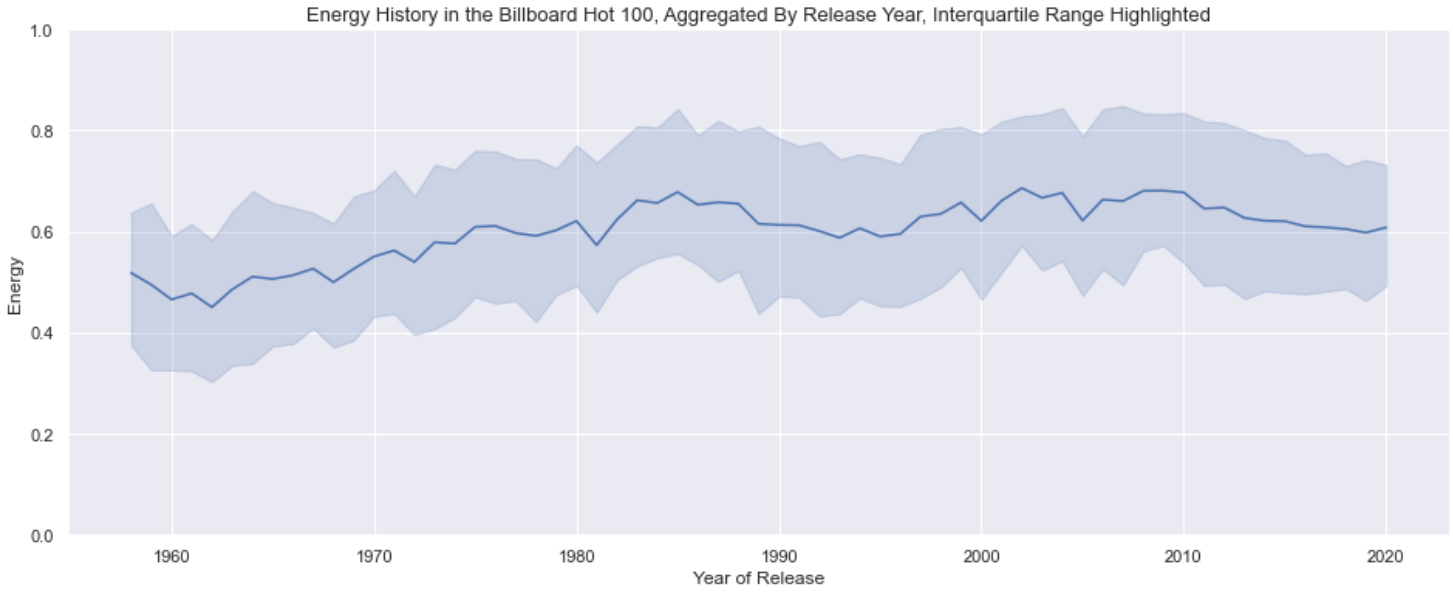

```
Danceability History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted
```



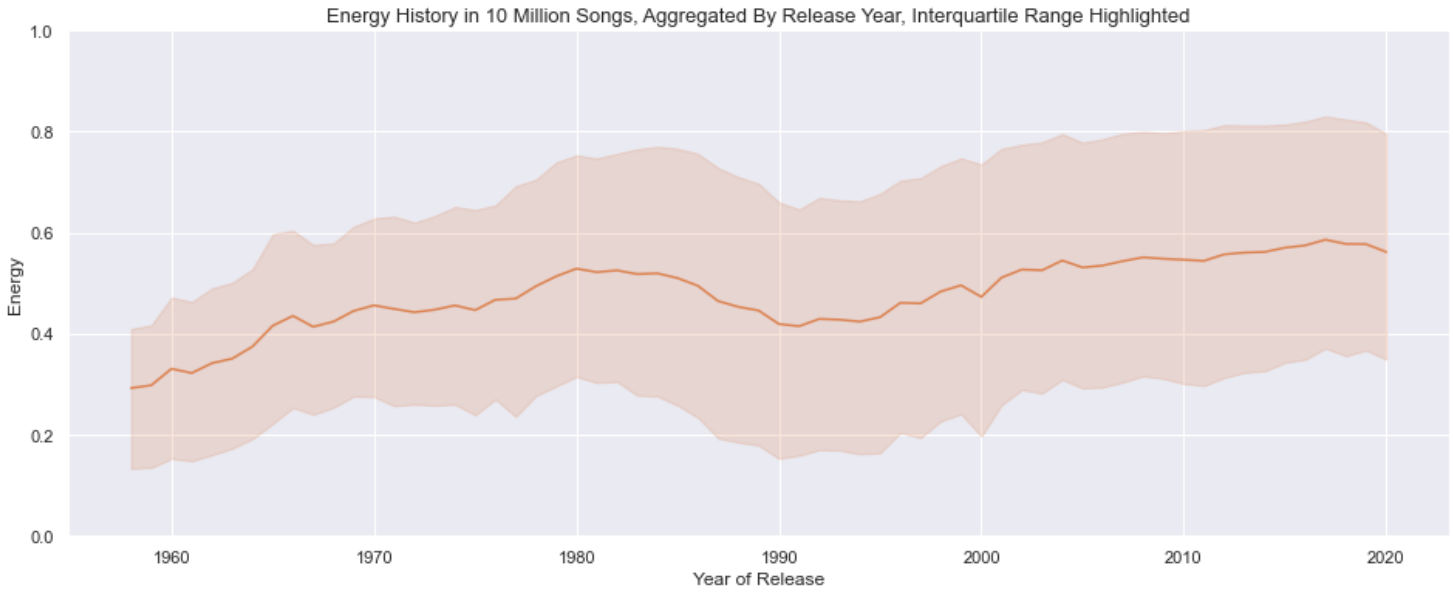
Danceability History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted



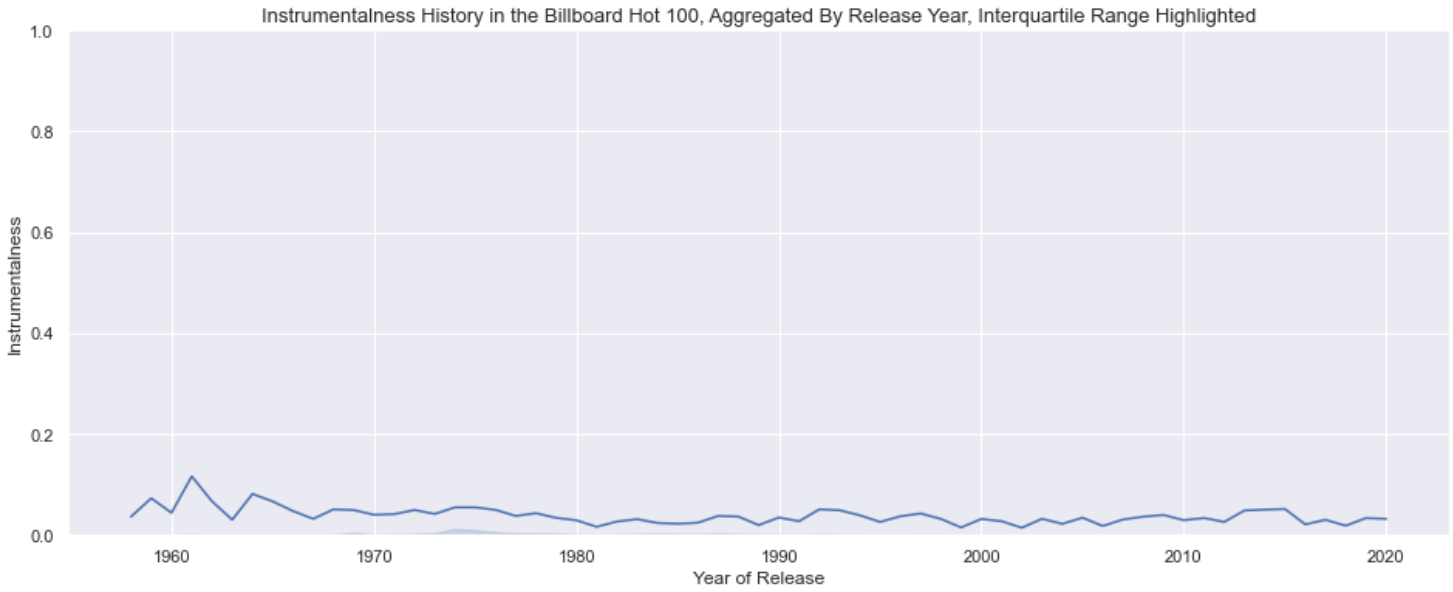
Energy History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted



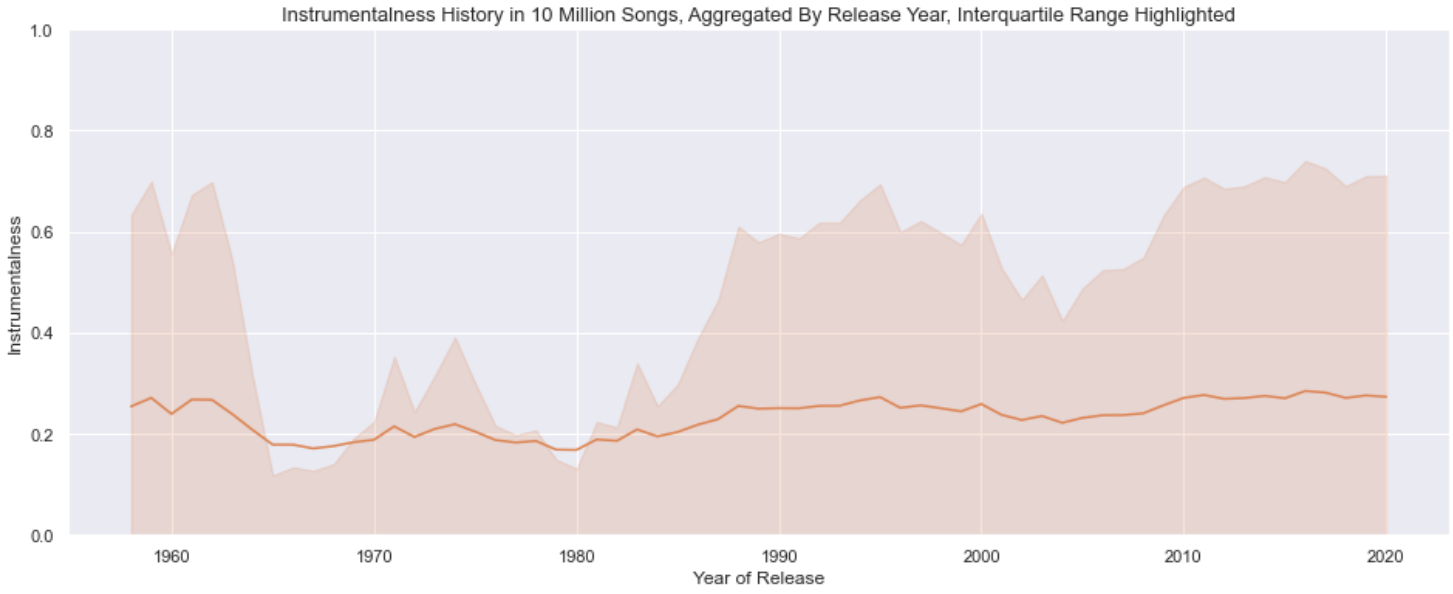
Energy History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted



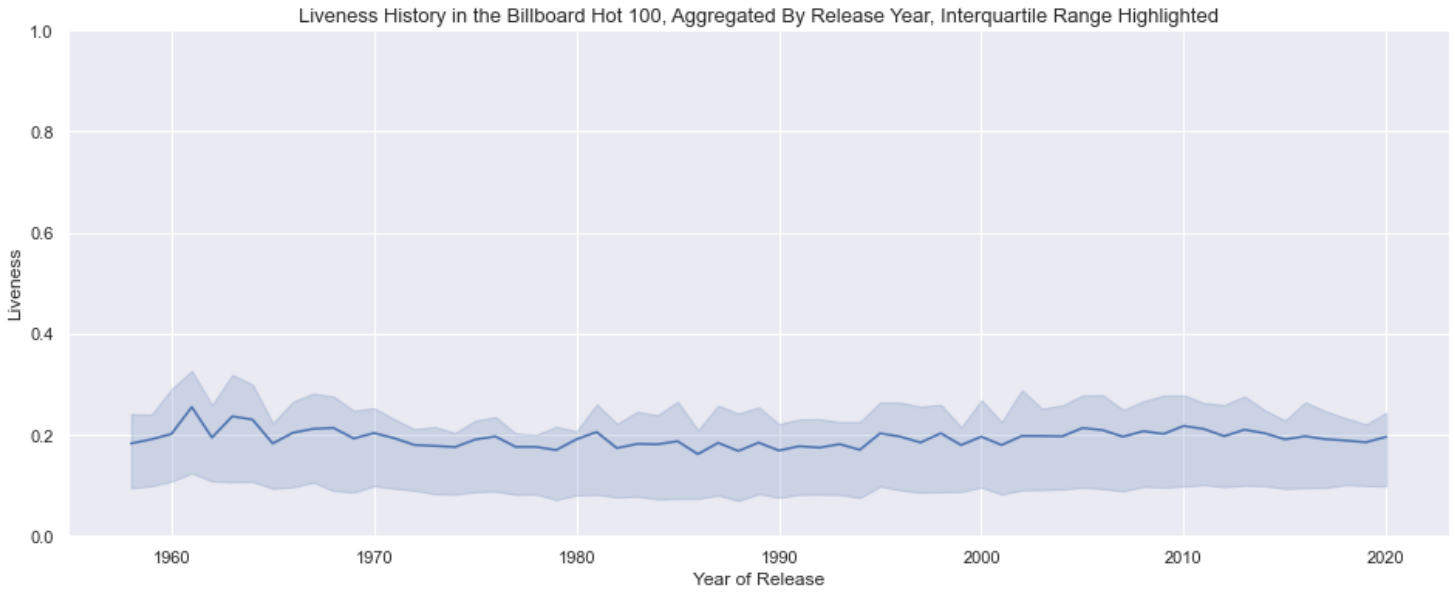
Instrumentalness History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted



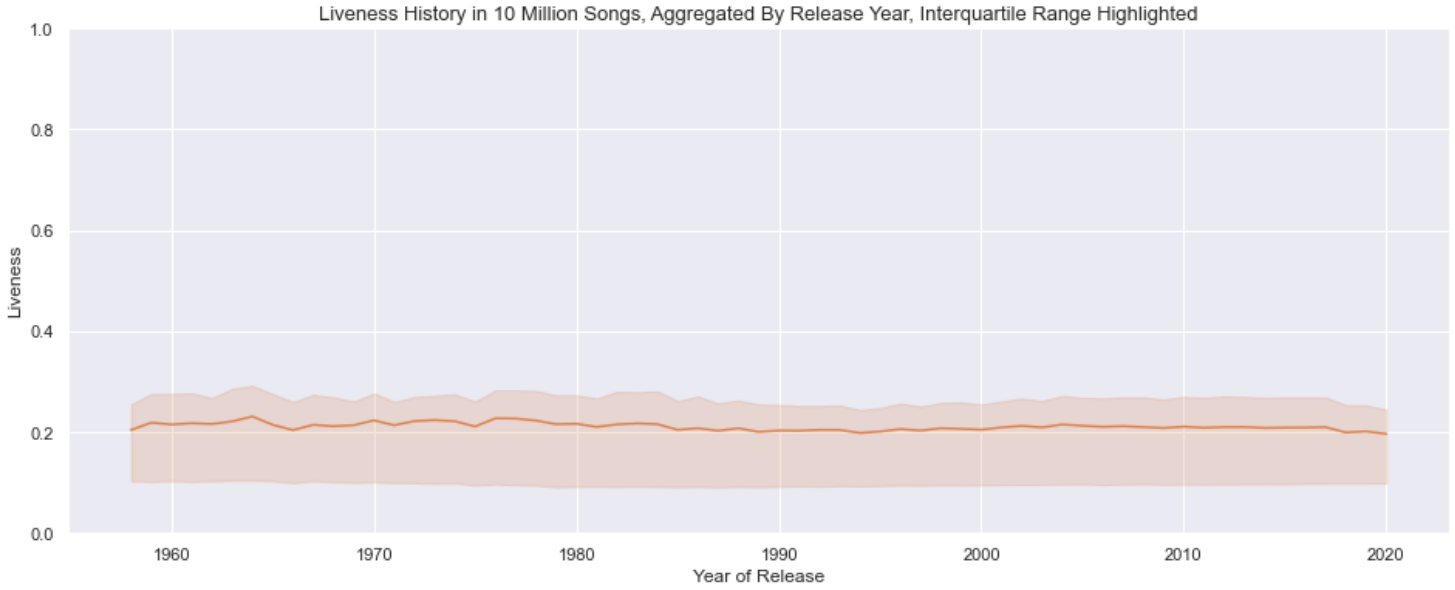
Instrumentalness History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted



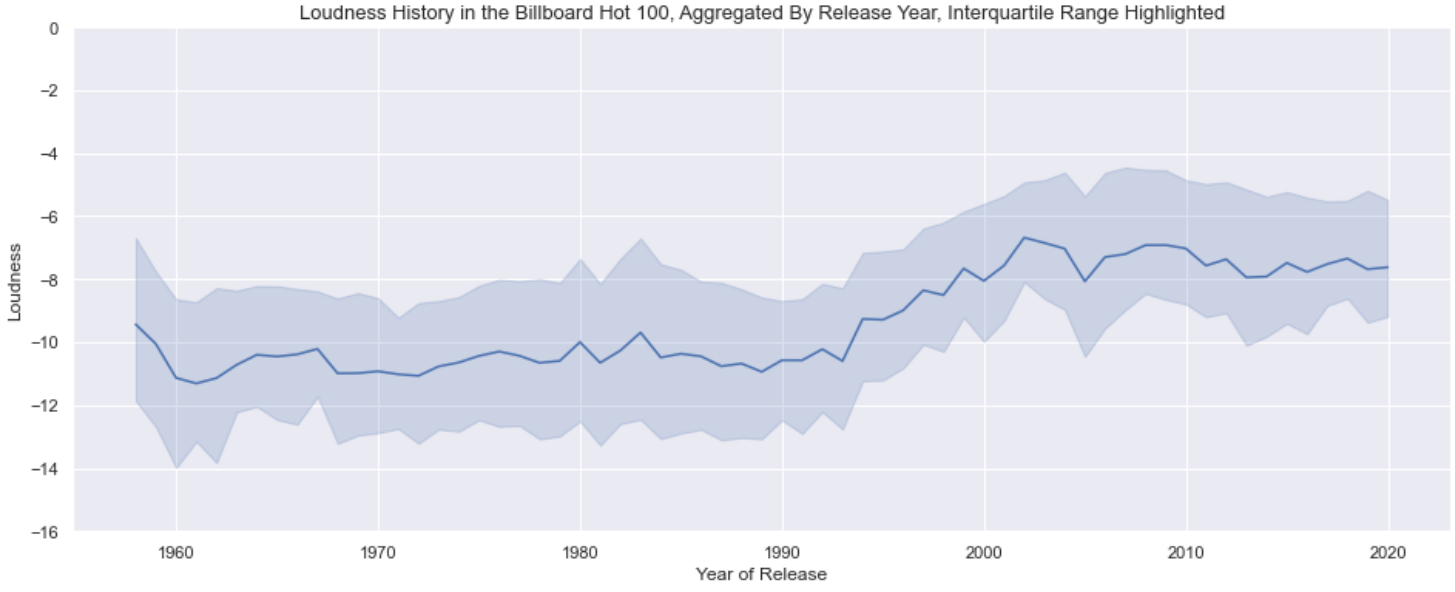
Liveness History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted



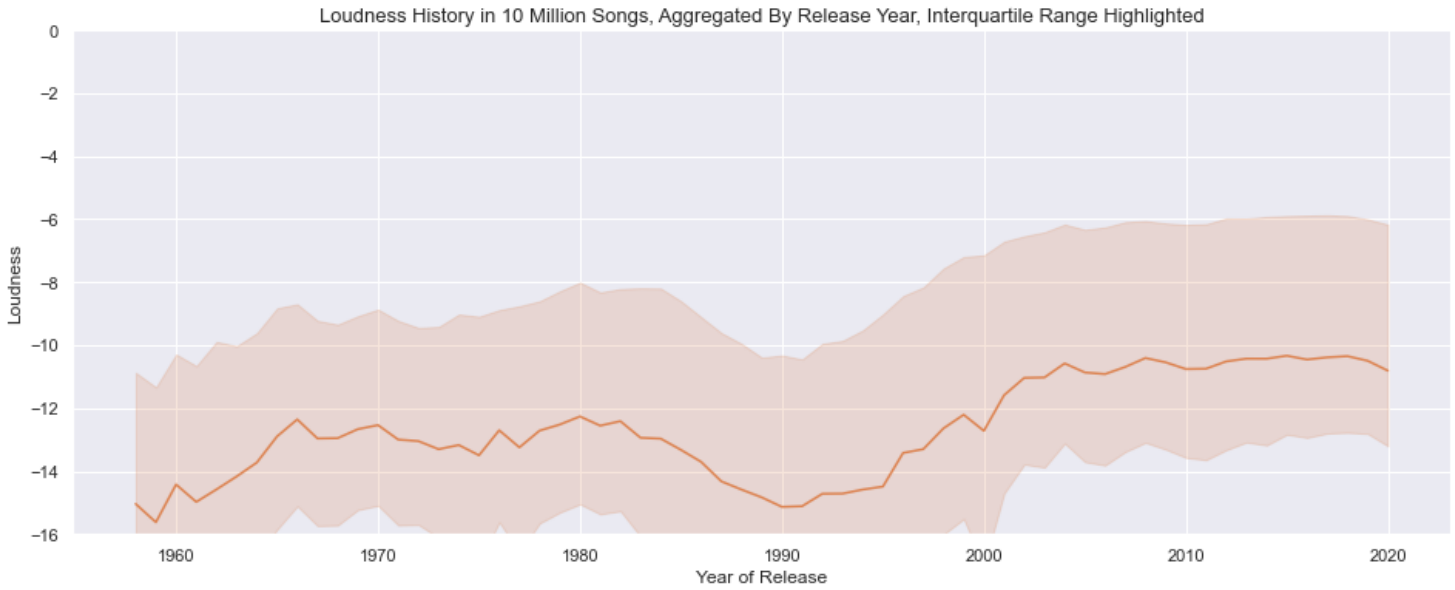
Liveness History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted



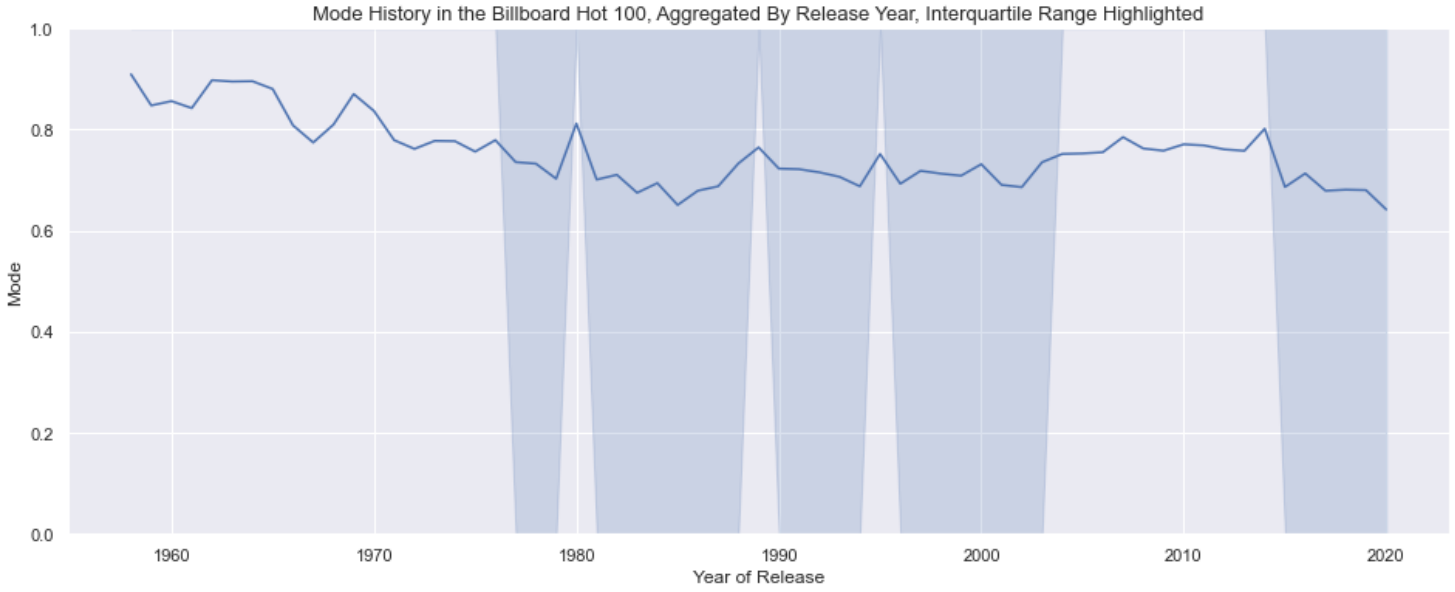
Loudness History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted



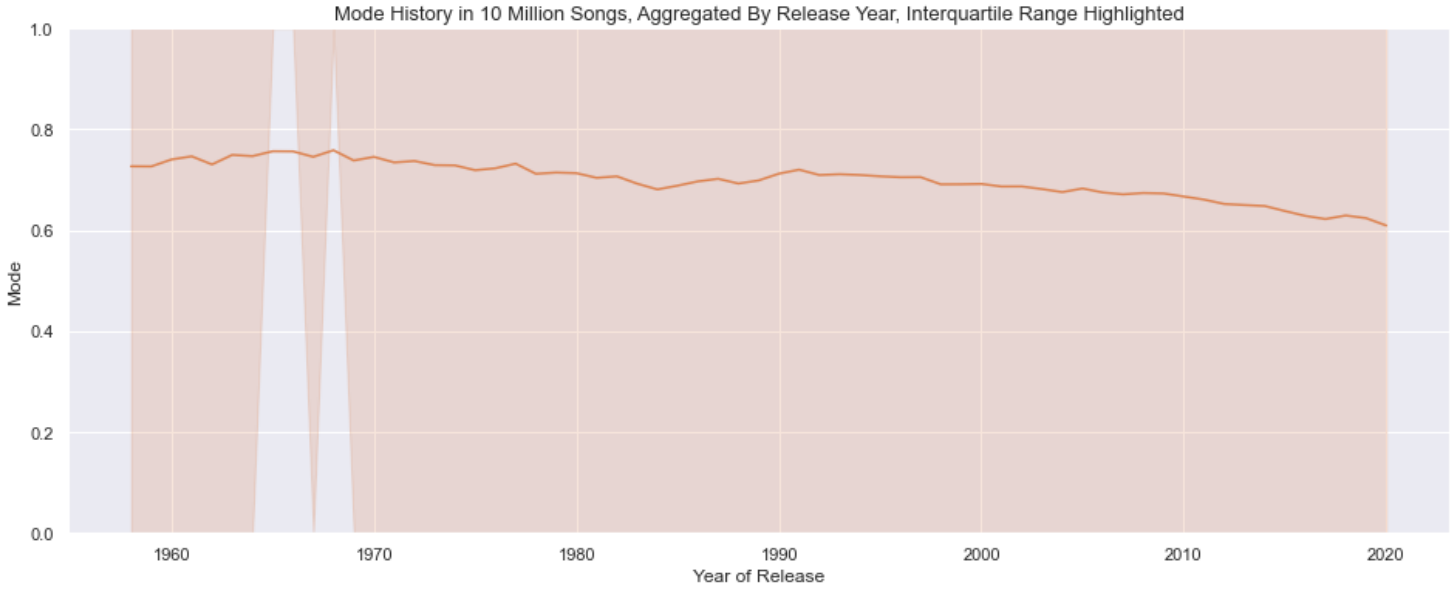
Loudness History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted



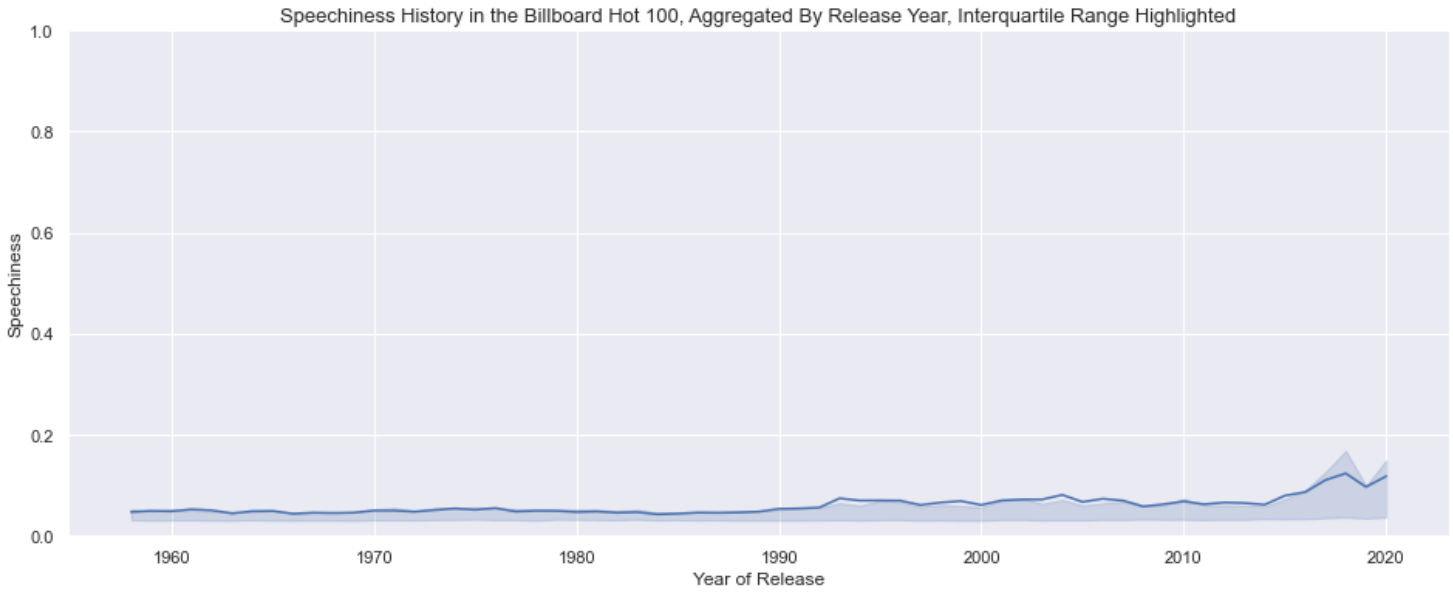
Mode History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted



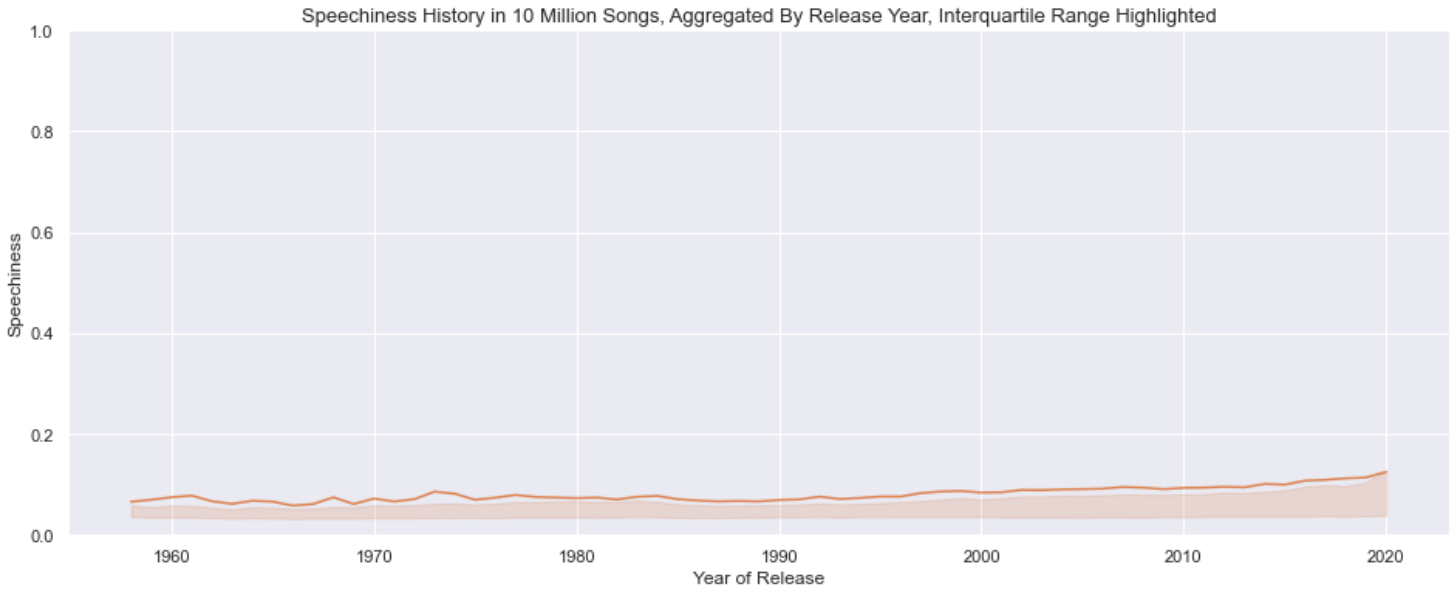
Mode History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted



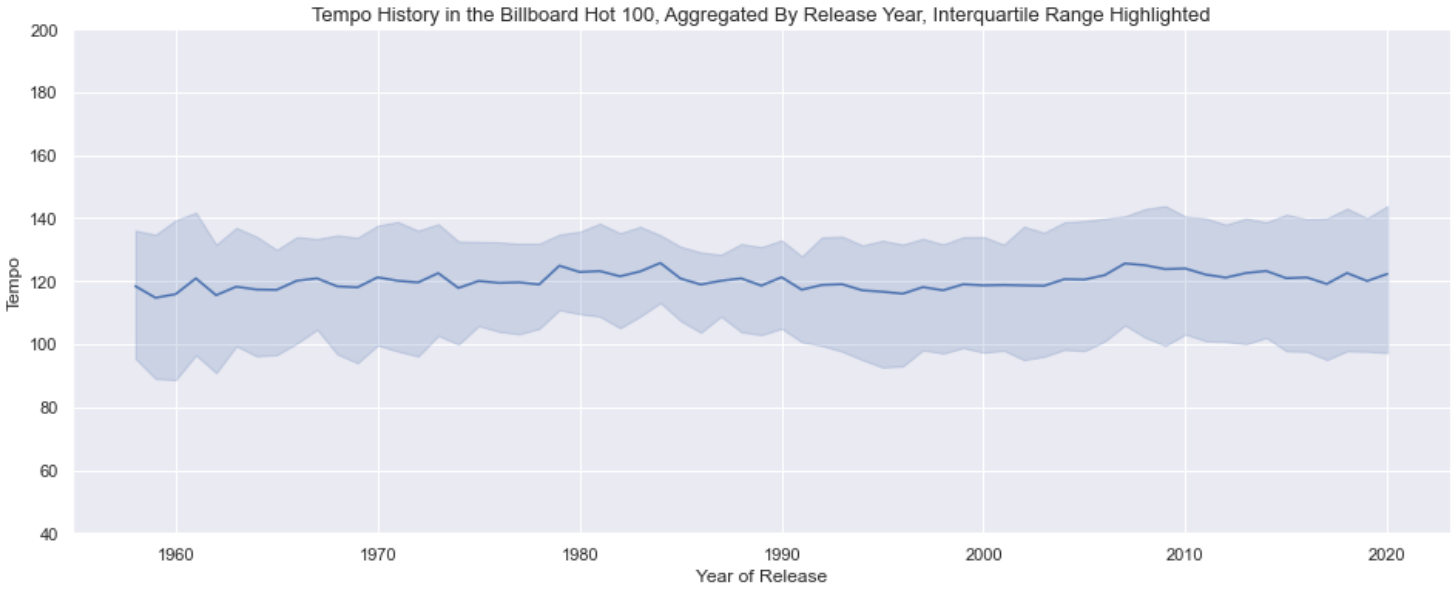
Speechiness History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted



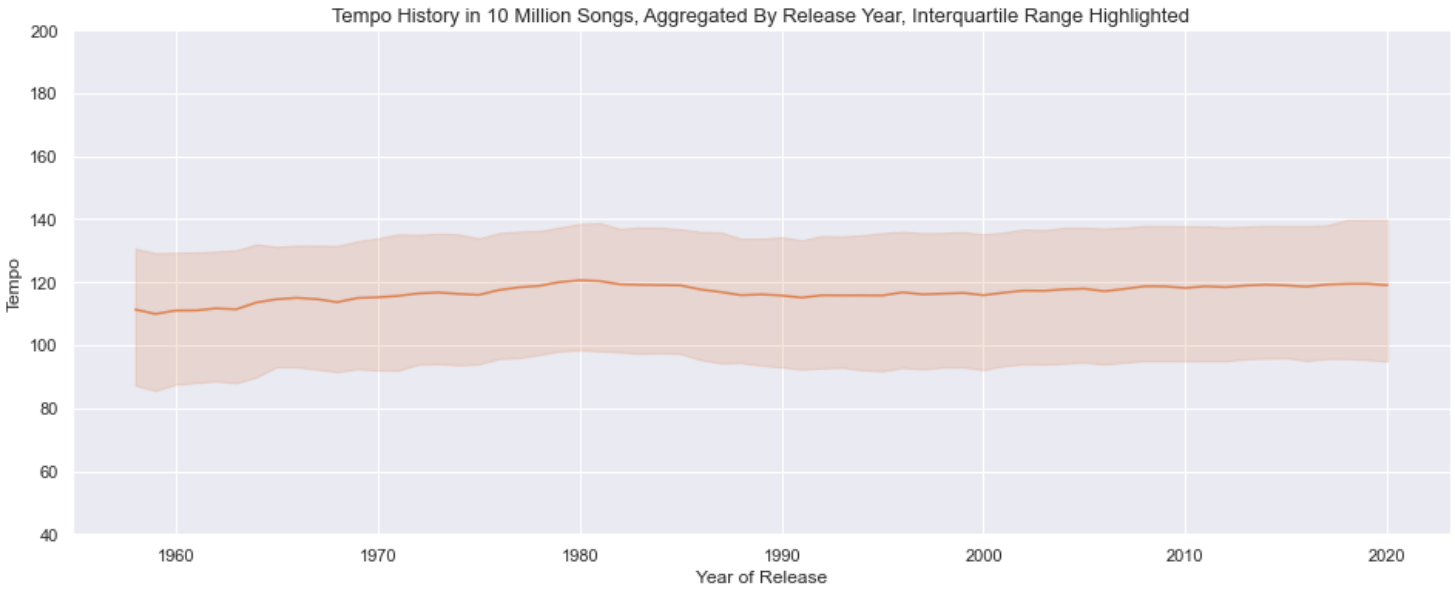
Speechiness History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted



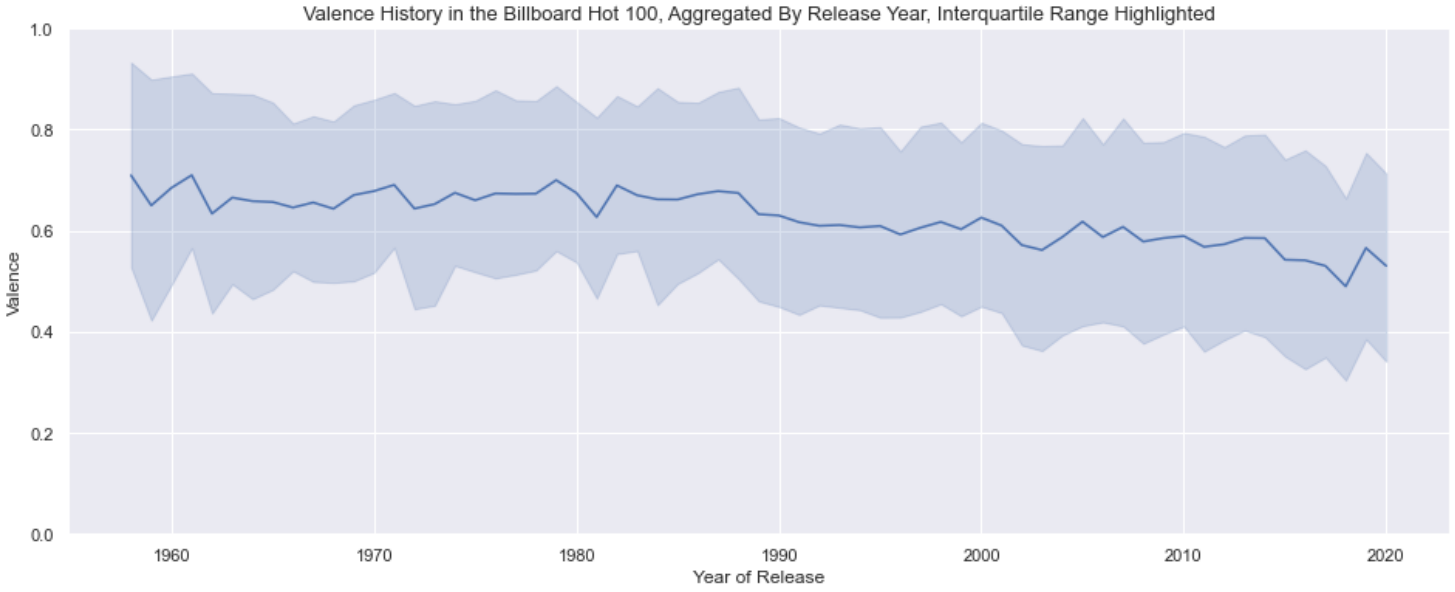
Tempo History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted



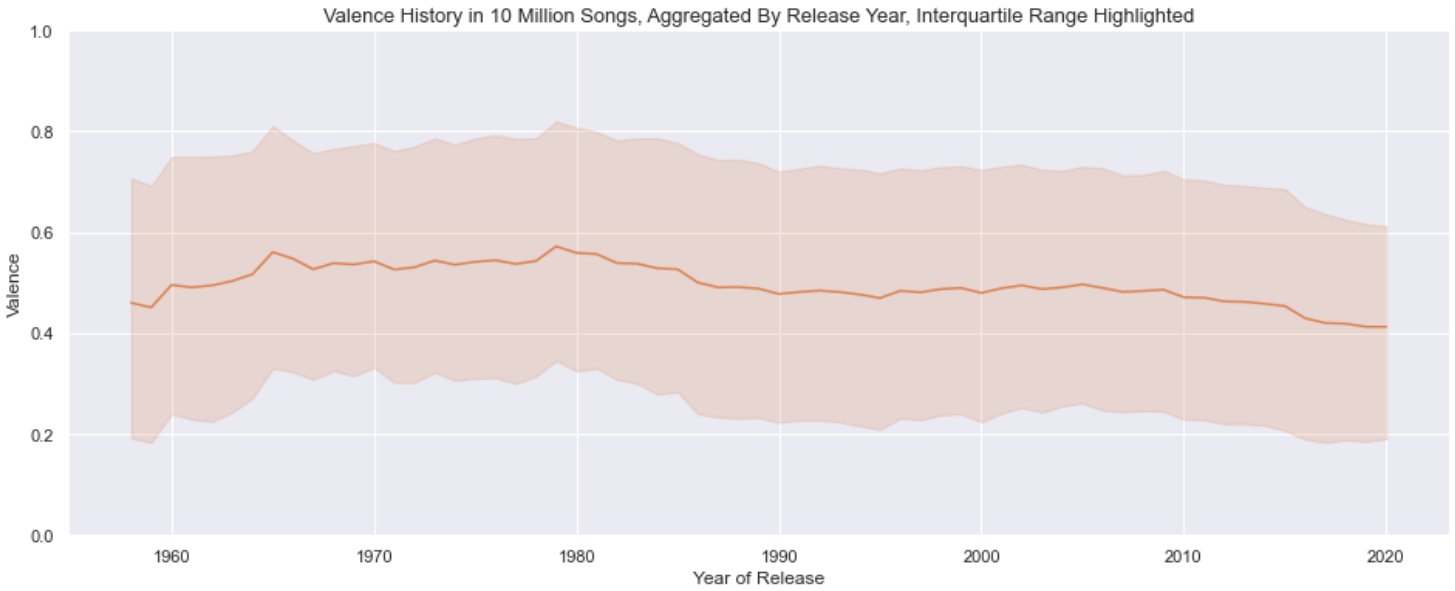
Tempo History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted




```
Valence History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted
```




```
Valence History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted
```




```
Wall time: 1min
```

In [25]:

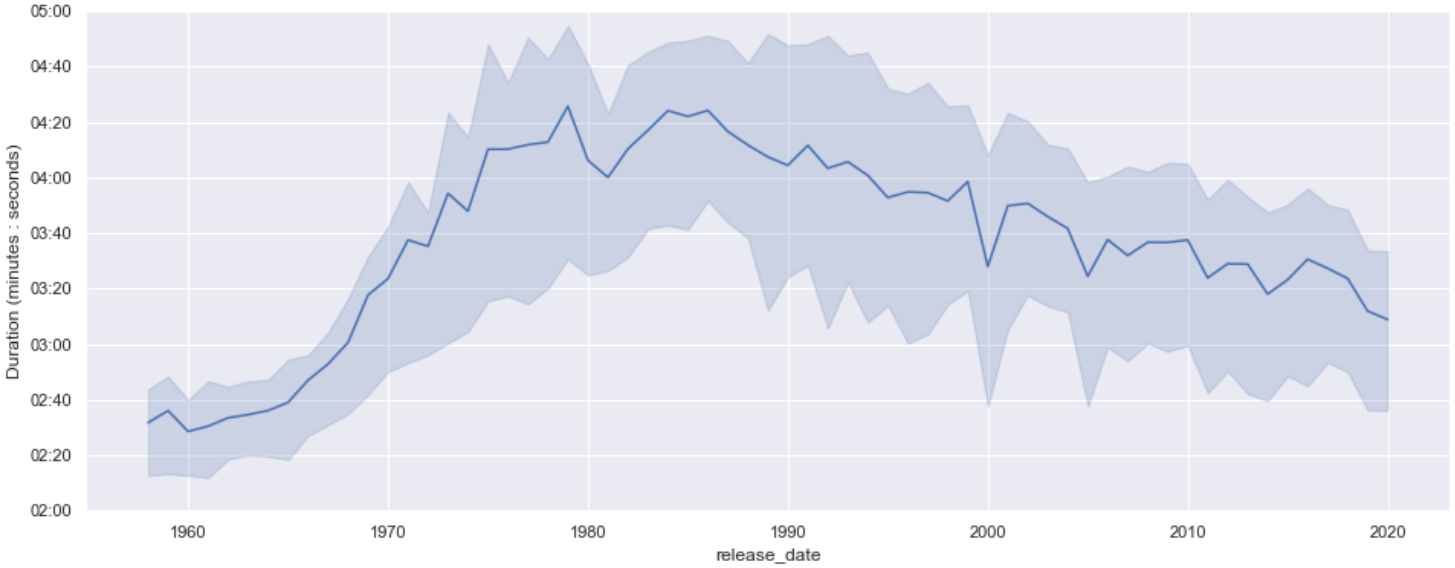
```
# audio features
other_features = [
```

```
    'key', 'time_signature', # integers don't make sense in timeseries lineplot (maybe heatplot?)
    'duration_ms',  # irregular feature, plotting requires some extra work
]

def plot_duration(feature='duration_ms', dataframe=df_B100_songs, title=None, colour=0):
    """
    Uses songs df_B100_songs
    Plots Billboard Hot 100 audio features over time
    """
    # drop values before billboard 100 (lower quality data)
    temp_df = dataframe.query(date_filter)

    plt.figure(figsize=(16, 6))
    ax = sns.lineplot(
        x=temp_df.release_date.dt.year,
        y=temp_df[feature],
        color=sns.color_palette()[colour],
        errorbar=('pi', 50)  # middle 50% or IQR
    )

    formatter = mpl.ticker.FuncFormatter(lambda ms, x: time.strftime('%M:%S', time.gmtime(ms // 1000)))
    ax.yaxis.set_major_formatter(formatter)
    plt.ylabel('Duration (minutes : seconds)')

    if title:
        title=title
    else:
        title=f'Song Duration History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted'

    plt.ylim(120000, 300000)  # reasonable range for duration of music

#     plt.title(title, fontsize=13)

    # save the image
    plt.savefig(f'figures/timeseries/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

In [26]:

```
plot_duration(feature='duration_ms')
```

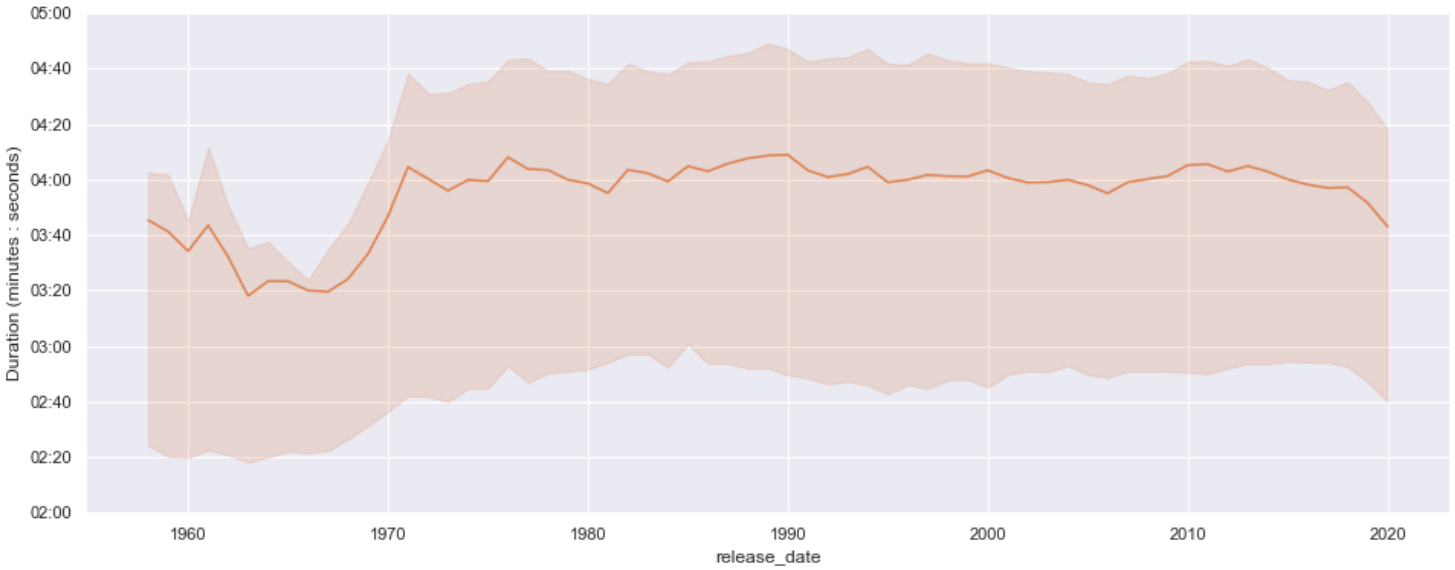
```
Song Duration History in the Billboard Hot 100, Aggregated By Release Year, Interquartile Range Highlighted
```



In [27]:

```
%%time
title = 'Song Duration History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted'
plot_duration(feature='duration_ms', dataframe=df_10M, title=title, colour=1)
```

```
Song Duration History in 10 Million Songs, Aggregated By Release Year, Interquartile Range Highlighted
```




```
Wall time: 9.14 s
```

Comparative Time Series

In [28]:

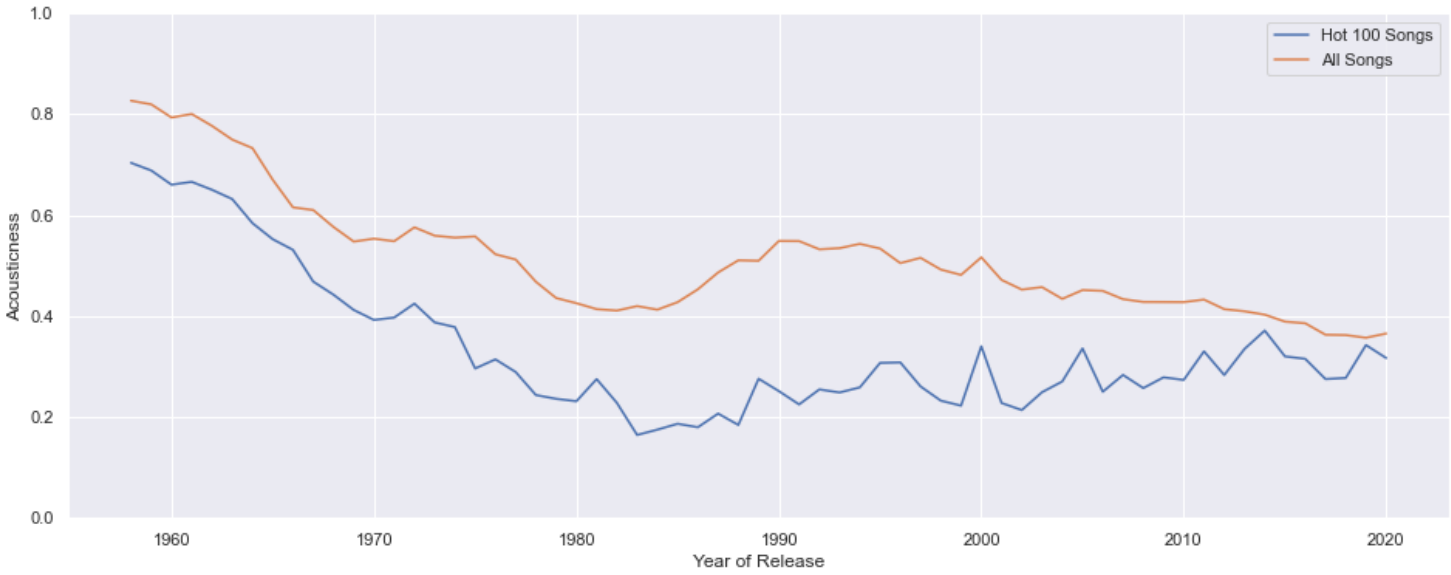
```python
def plot_feature_comparison(feature):
    """
    Uses songs df_B100_songs
    Plots Billboard Hot 100 audio features over time
    """

    fig, ax = plt.subplots(figsize=(16, 6))

    df1 = df_B100_songs.query(date_filter)
    ax = sns.lineplot(
        x=df1.release_date.dt.year,
        y=df1[feature],
        color=sns.color_palette()[0],
        errorbar=None,
        label='Hot 100 Songs'
    )

    df2 = df_10M.query(date_filter)
    ax = sns.lineplot(
        x=df2.release_date.dt.year,
        y=df2[feature],
        color=sns.color_palette()[1],
        errorbar=None,
        label='All Songs'
    )

    title=f'{feature.capitalize()} History Comparing the Billboard Hot 100 with All Songs'

    if feature == 'loudness':
        plt.ylim(-16, 0)  # reasonable range for loudness in music
    elif feature == 'tempo':
        plt.ylim(40, 200)  # reasonable range for tempo in music
    else:
        plt.ylim(0, 1)  # most audio features vary between 0 and 1
        # NOTE: mode is an int, either 0 or 1 (minor/major),
        # but the average shows how commonly a song is in either mode

#     plt.title(title, fontsize=13)
    plt.xlabel('Year of Release')
    plt.ylabel(feature.capitalize().replace('_', ' '))
    plt.legend(loc='upper right')

    # save the image
    plt.savefig(f'figures/timeseries/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

```
def plot_duration_comparison(feature='duration_ms'):
    """
    Uses songs df_B100_songs
    Plots Billboard Hot 100 audio features over time
    """

    fig, ax = plt.subplots(figsize=(16, 6))

    df1 = df_B100_songs.query(date_filter)
    ax = sns.lineplot(
        x=df1.release_date.dt.year,
        y=df1[feature],
        color=sns.color_palette()[0],
        errorbar=None,
        label='Hot 100 Songs'
    )

    df2 = df_10M.query(date_filter)
    ax = sns.lineplot(
        x=df2.release_date.dt.year,
        y=df2[feature],
        color=sns.color_palette()[1],
        errorbar=None,
        label='All Songs'
    )

    formatter = mpl.ticker.FuncFormatter(lambda ms, x: time.strftime('%M:%S', time.gmtime(ms // 1000)))
    ax.yaxis.set_major_formatter(formatter)
    plt.ylabel('Duration (minutes : seconds)')

    title='Song Duration History Comparing the Billboard Hot 100 with All Songs'

    plt.ylim(120000, 300000)  # reasonable range for duration of music

#     plt.title(title, fontsize=13)
    plt.xlabel('Year of Release')
    plt.legend(loc='upper right')

    # save the image
    plt.savefig(f'figures/timeseries/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

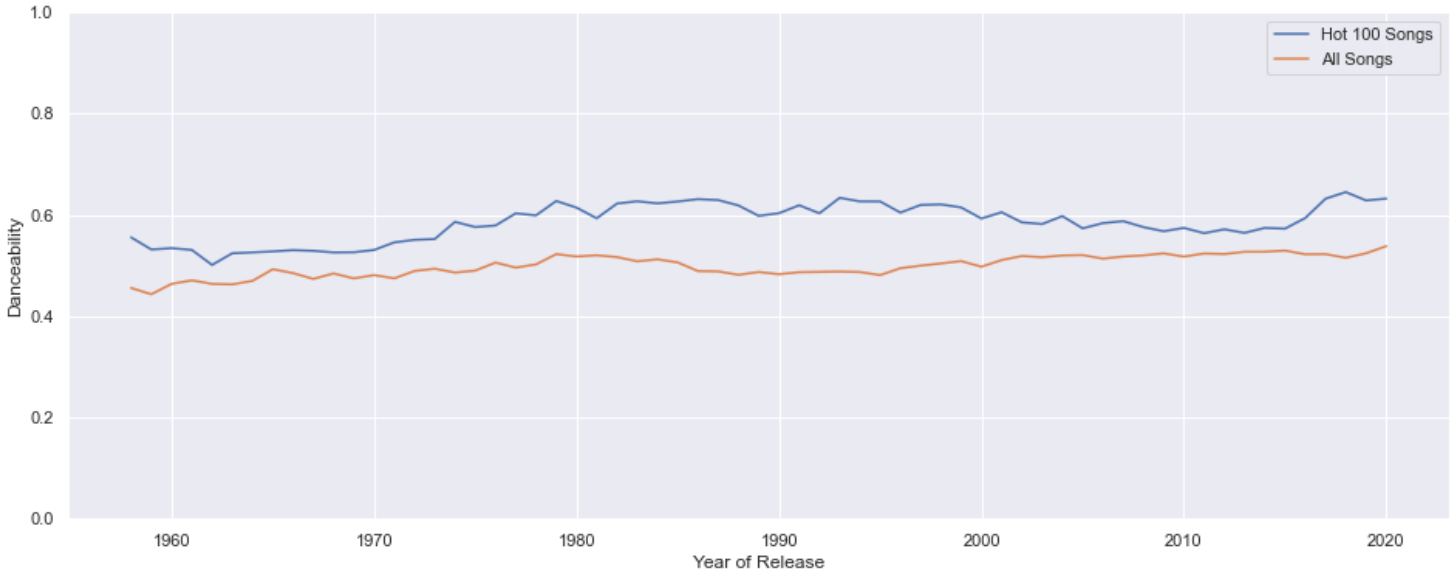
In [29]:

```
%%time
# audio features
main_features = [
    'acousticness', 'danceability', 'energy', 'instrumentalness',
    'liveness', 'loudness', 'mode', 'speechiness', 'tempo', 'valence'
]

for feature in main_features:
    plot_feature_comparison(feature)
plot_duration_comparison()
```

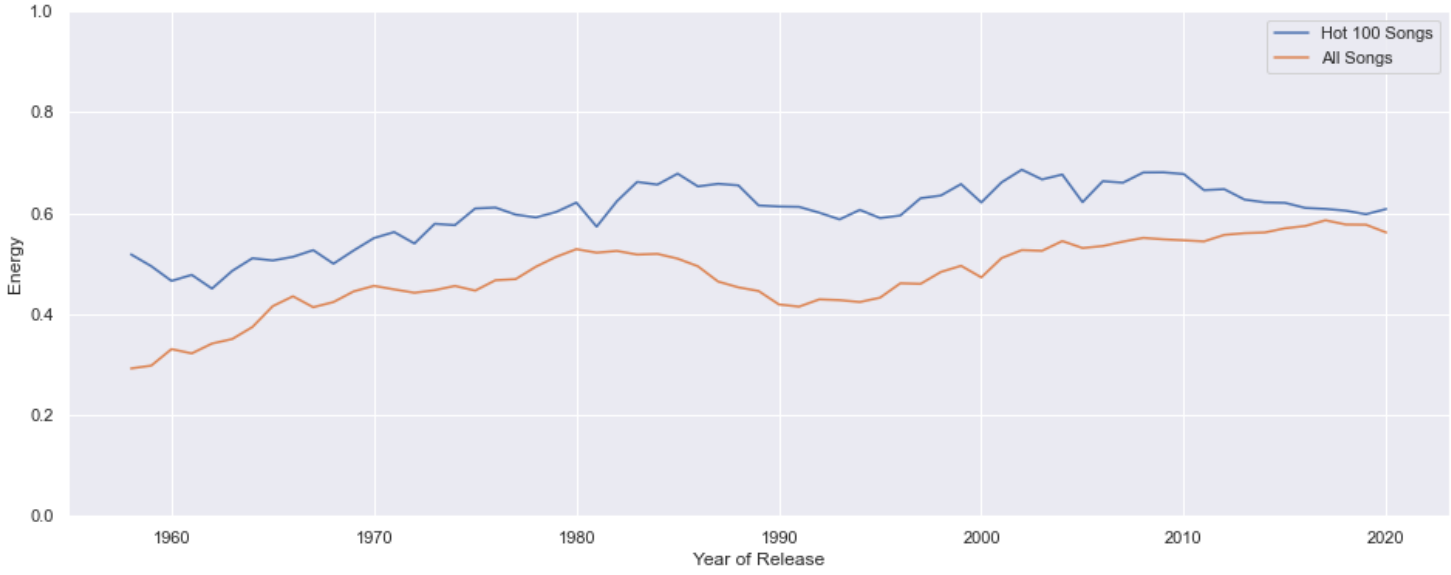
```
Acousticness History Comparing the Billboard Hot 100 with All Songs
```



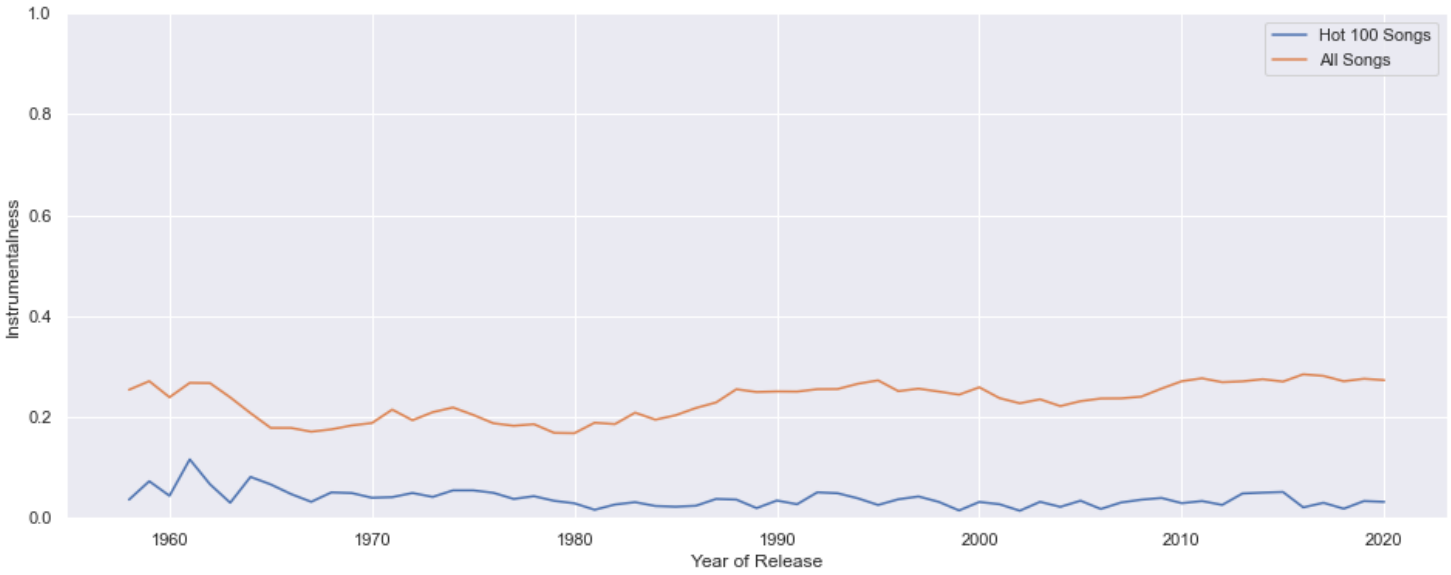
Danceability History Comparing the Billboard Hot 100 with All Songs



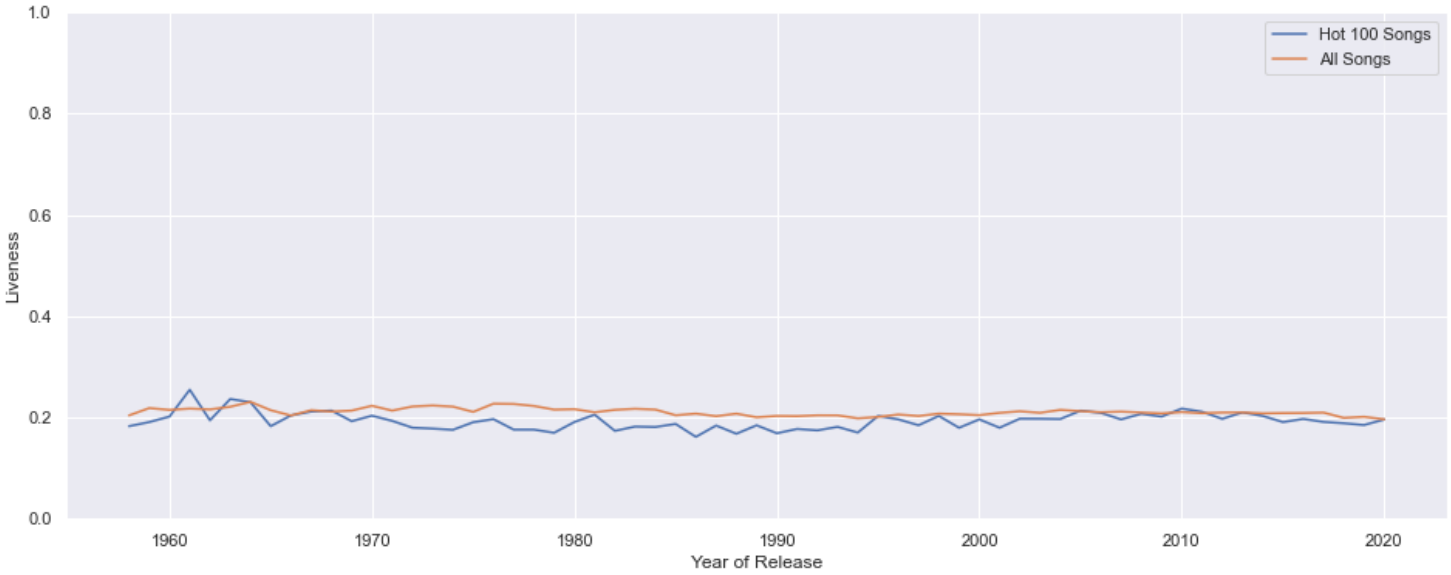
Energy History Comparing the Billboard Hot 100 with All Songs



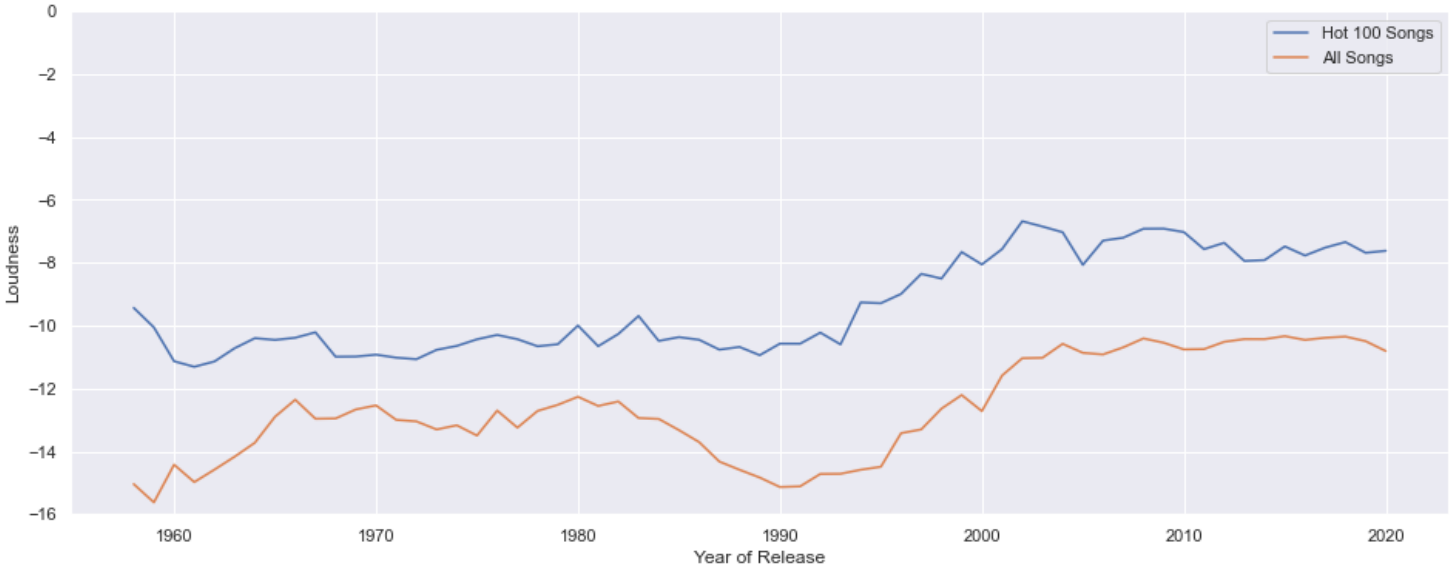
Instrumentalness History Comparing the Billboard Hot 100 with All Songs



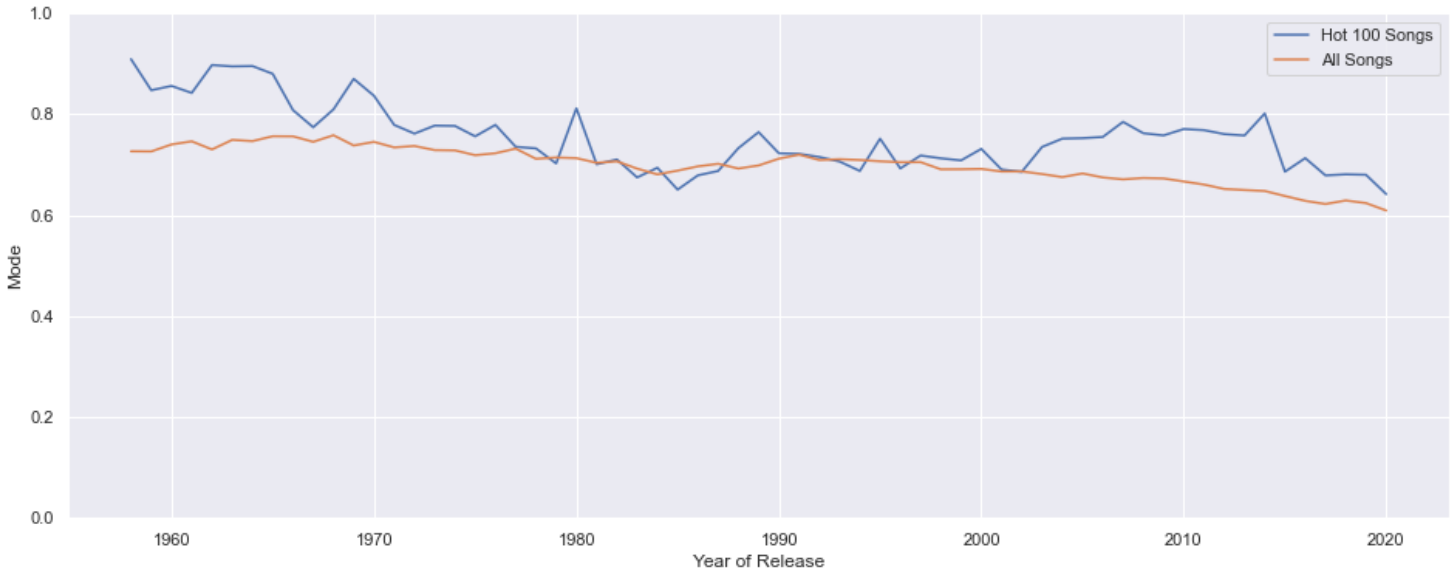
Liveness History Comparing the Billboard Hot 100 with All Songs



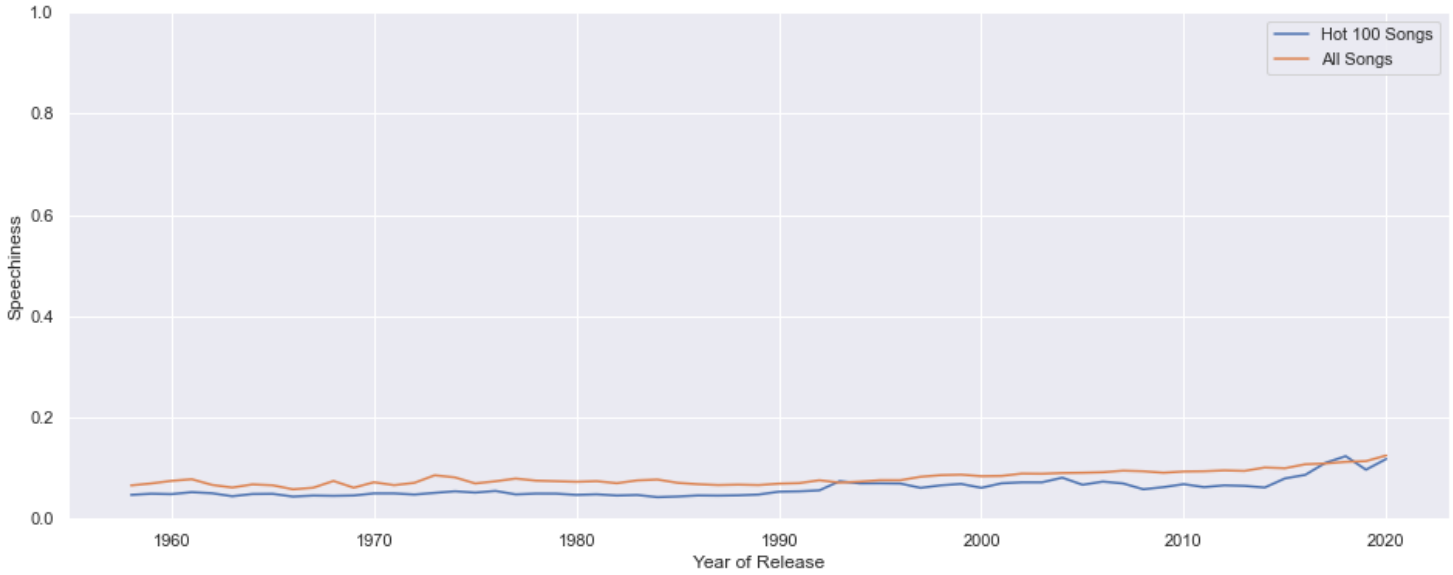
Loudness History Comparing the Billboard Hot 100 with All Songs



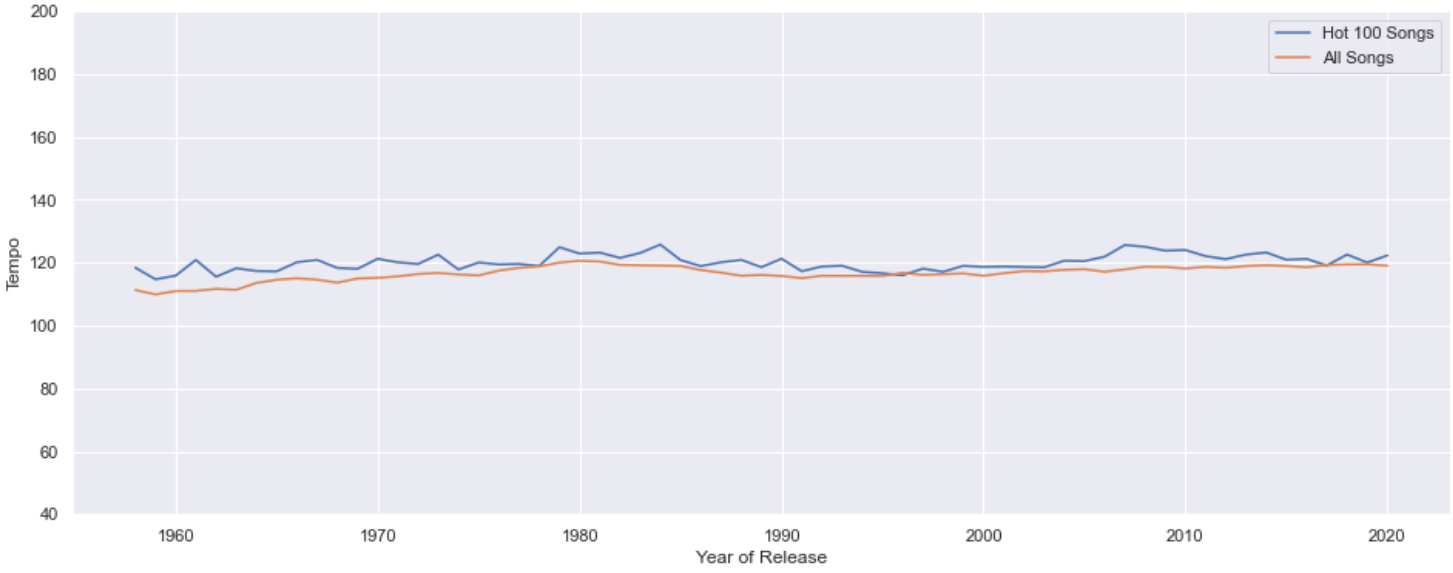
Mode History Comparing the Billboard Hot 100 with All Songs



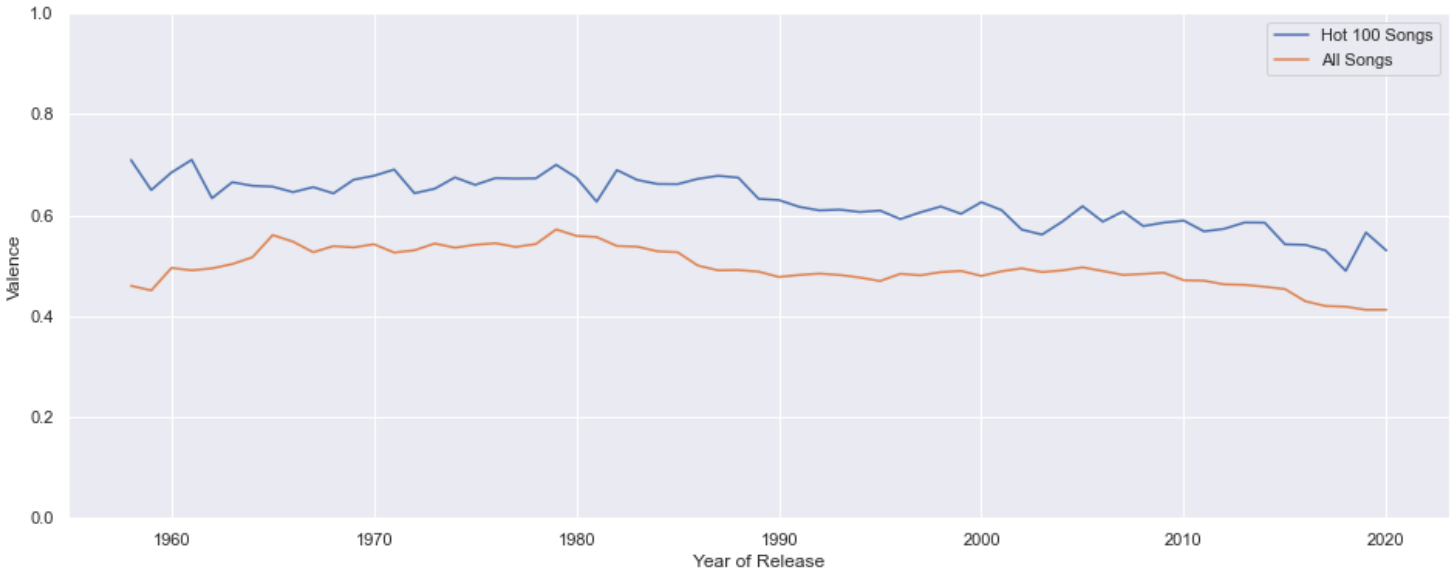
Speechiness History Comparing the Billboard Hot 100 with All Songs



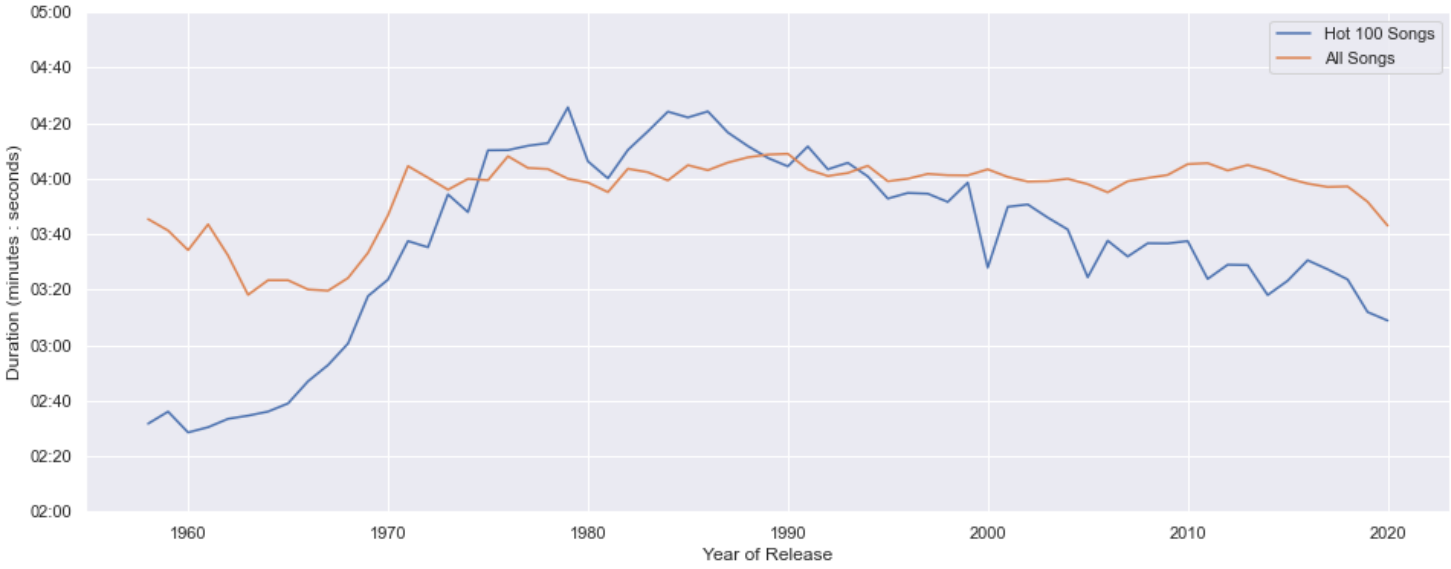
Tempo History Comparing the Billboard Hot 100 with All Songs



Valence History Comparing the Billboard Hot 100 with All Songs




```
Song Duration History Comparing the Billboard Hot 100 with All Songs
```




```
Wall time: 1min 2s
```

Time Series Counts - Preferred Date Filter

- data seems to deteriorate after 2020, with large movements in music trends (CORRECT)
- alternatively, this could be due to significant societal changes and preferences (NOT VERIFIABLE) ##### Let's find out!

In [30]:
```
df_10M.release_date.dt.year.value_counts().reset_index().sort_values('index', ascending=False).head()
```

Out[30]:

	index	release_date
98	2022.000	234
26	2021.000	71985
8	2020.000	307877
10	2019.000	280639
11	2018.000	273200

In [31]:
```
df_B100_songs.release_date.dt.year.value_counts().reset_index().sort_values('index', ascending=False).head()
```

Out[31]:

	index	release_date
57	2022	202
6	2021	524

	index	release_date
0	2020	653
5	2019	525
2	2018	564

In [32]:
```
# find max dates for each dataset

# SQL
max_date_sql = 1644537600000
pd.to_datetime(max_date_sql, unit='ms', origin='unix')
```

Out[32]:
```
Timestamp('2022-02-11 00:00:00')
```

In [33]:
```
# Billboard Hot 100
df_B100.sort_values('date', ascending=False).head(1)['date']
```

Out[33]:
```
0   2021-11-06
Name: date, dtype: datetime64[ns]
```

In [34]:
```
# CSV
pd.read_csv('D:/RYERSON/820/Datasets/Spotify 1.2M+ Songs/tracks_features.csv').sort_values('release_date', ascending=False).he
```

Out[34]:
```
1134062    2020-12-18
Name: release_date, dtype: object
```

In [35]:
```
# Latest Release Dates by Dataset
latest_sql = pd.to_datetime('2022-02-11 00:00:00')
latest_B100 = pd.to_datetime('2021-11-06 00:00:00')
latest_csv = pd.to_datetime('2020-12-18 00:00:00')

# max date option 1 (all datasets consistent):
maxdate_1 = min(latest_sql, latest_B100, latest_csv)

# max date option 2 (all Billboard 100 songs):
maxdate_2 = min(latest_sql, latest_B100)

# options for filtering
maxdate_1, maxdate_2
```

Out[35]:
```
(Timestamp('2020-12-18 00:00:00'), Timestamp('2021-11-06 00:00:00'))
```

In [36]:
```
# let's check the quantiy (not quality) of 2021 (with SQL, without CSV) vs 2020 (both)

df_10M.query('2021 <= release_date < 2022').shape[0], df_10M.query('2020 <= release_date < 2021').shape[0]
```

Out[36]:
```
(71985, 307877)
```

In [37]:
```
# only 23% of the songs during 2021 vs 2020: looks like maxdate_1 is a better choice
df_10M.query('2021 <= release_date < 2022').shape[0] / df_10M.query('2020 <= release_date < 2021').shape[0]
```

Out[37]:
```
0.23381090500427118
```

CONCLUSION: We should only include data up to, but not including 2021

In [38]:
```
if []:
    print('hi')
```

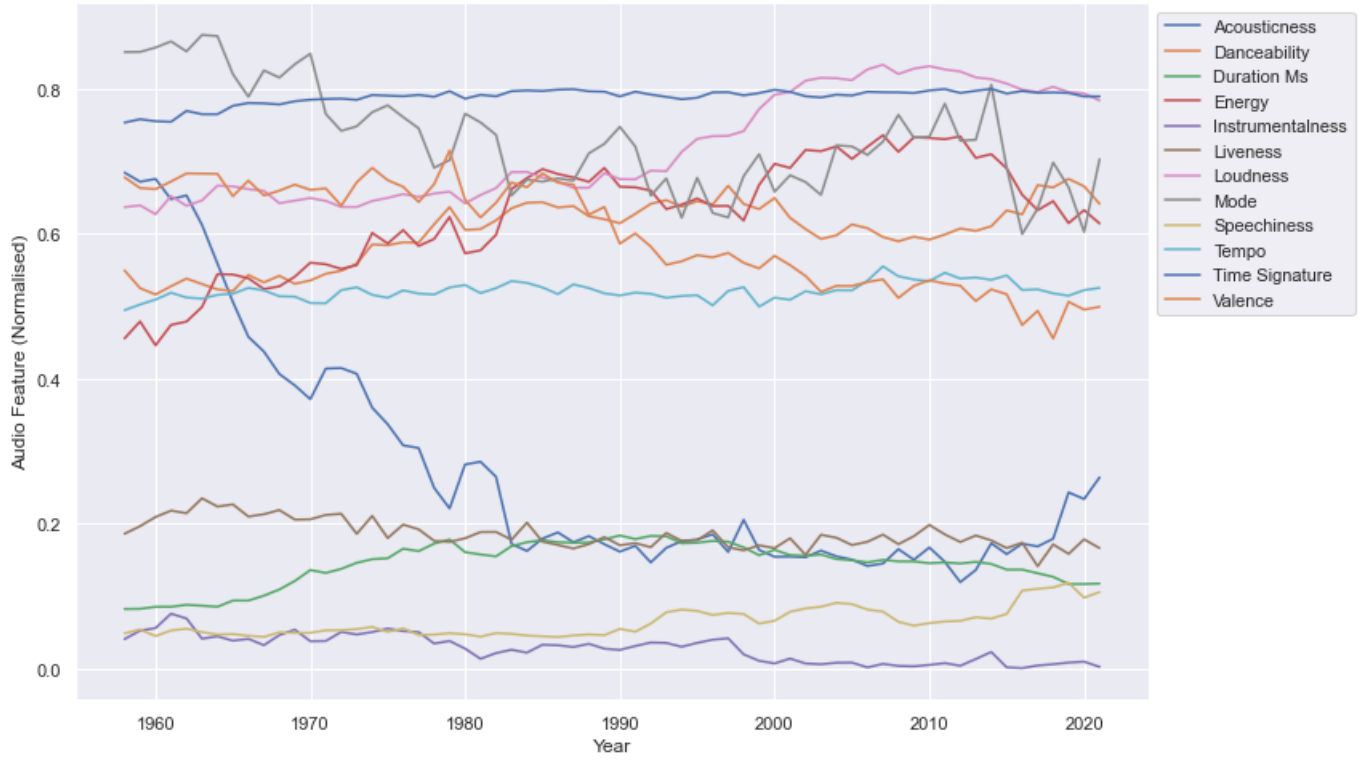
Billboard Charts Historical Plots

In [39]:
```
# audio features
normalised_features = [
    'acousticness', 'danceability', 'energy', 'instrumentalness',
    'liveness', 'mode', 'speechiness', 'valence'
]
```

```
# features to normalise
features_to_normalise = [
    'loudness', 'tempo', 'duration_ms', 'time_signature'
]

def plot_billboard_history(normalised_features, features_to_normalise):
    """
    Uses songs df_B100
    Plots Billboard Hot 100 audio features over time
    """
    if features_to_normalise:  # if there are features to normalise
        # combine lists of features
        list_of_features = sorted([*normalised_features, *features_to_normalise])

        # create the normalised dataframe
        df = df_B100.dropna().copy()
        n = features_to_normalise  # just to shorten the next line of code
        df[[*n]] = (df[[*n]]-df[[*n]].min()) / (df[[*n]].max()-df[[*n]].min())
    else:
        list_of_features = normalised_features
        df = df_B100.dropna().copy()

    # create the plot
    fig, ax = plt.subplots(figsize=(12, 8))

    for feature in list_of_features:
        ax = sns.lineplot(
            x=df.date.dt.year,
            y=df[feature],
            errorbar=None,
            linewidth=1.5,
            label=feature.replace('_', ' ').title()
        )

    title='Billboard Hot 100 Historical Charts - All Features Normalised'

#     plt.title(title, fontsize=13)
    plt.xlabel('Year')
    plt.ylabel('Audio Feature (Normalised)')
    plt.legend(loc='upper left', bbox_to_anchor=(1, 1))

    # save the image
    plt.savefig(f'figures/timeseries/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    plt.show()
```

In [40]:

```
# plot all features historically
plot_billboard_history(normalised_features, features_to_normalise)
```



In [41]:

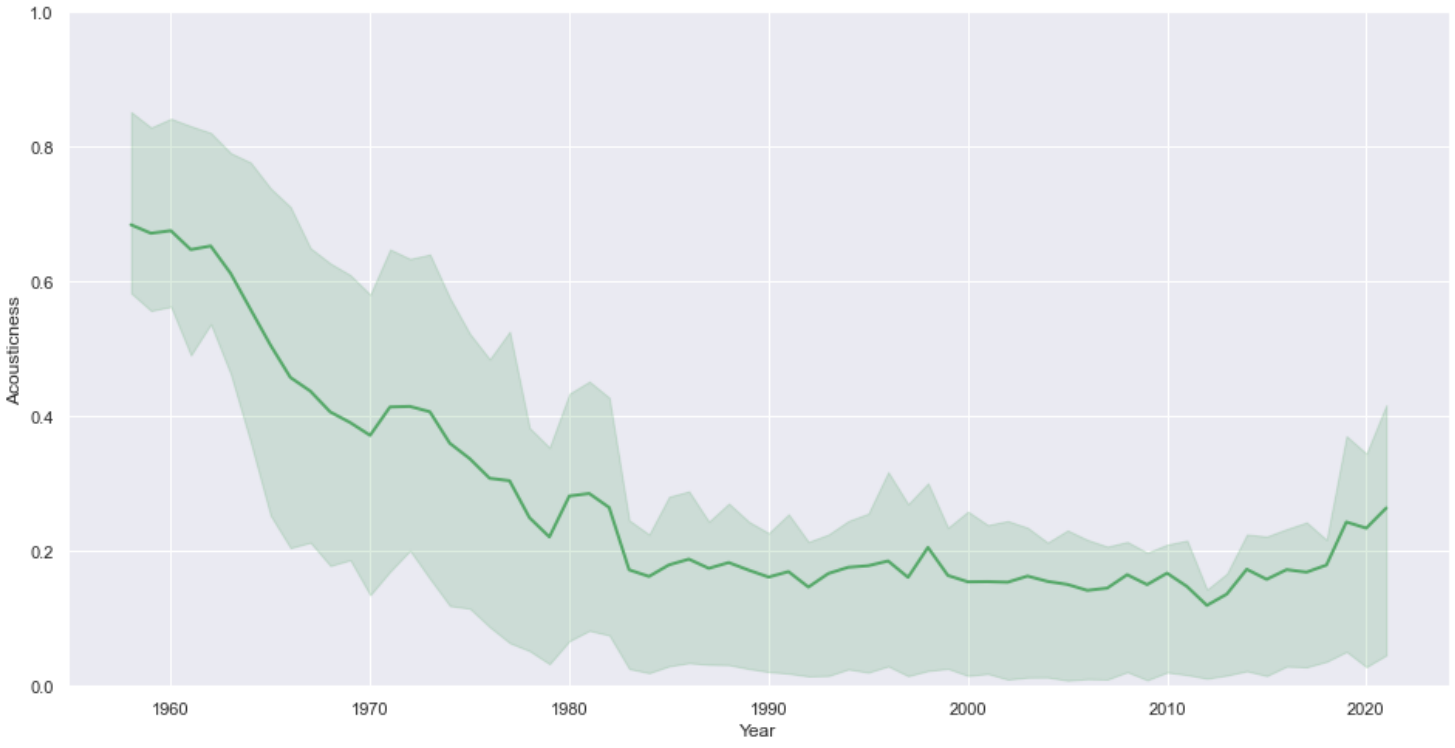
```
# plot individual features

def plot_billboard_history_by_feature(feature):
    """
    Uses songs df_B100
    Plots Billboard Hot 100 audio features over time
    """
    df = df_B100.dropna().copy()
    figsize=(16, 8)

    plt.figure(figsize=figsize)

    ax = sns.lineplot(
        x=df.date.dt.year,
        y=df[feature],
        errorbar=('pi', 50),  # IQR
        linewidth=2,
        color=sns.color_palette()[2]  # different colour to differentiate with other plots
    )

    if feature == 'loudness':
        plt.ylim(-16, 0)  # reasonable range for loudness in music
    elif feature == 'tempo':
        plt.ylim(40, 200)  # reasonable range for tempo in music
    elif feature == 'time_signature':
        plt.ylim(0, 5)
    else:
        plt.ylim(0, 1)  # most audio features vary between 0 and 1
        # NOTE: mode is an int, either 0 or 1 (minor/major),
        # but the average shows how commonly a song is in either mode

    title = 'Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - {}'.format(feature.replace('_', ' ').title

#     plt.title(title, fontsize=13)
    plt.xlabel('Year')
    plt.ylabel(feature.capitalize().replace('_', ' '))

    # save the image
    plt.savefig(f'figures/timeseries/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

```
def plot_billboard_history_duration(feature='duration_ms'):

    df = df_B100.dropna().copy()
    figsize=(16, 8)
    plt.figure(figsize=figsize)

    ax = sns.lineplot(
        x=df.date.dt.year,
        y=df[feature],
        errorbar=('pi', 50),  # IQR
        linewidth=2,
        color=sns.color_palette()[2]  # different colour to differentiate with other plots
    )

    formatter = mpl.ticker.FuncFormatter(lambda ms, x: time.strftime('%M:%S', time.gmtime(ms // 1000)))
    ax.yaxis.set_major_formatter(formatter)
    plt.ylabel('Duration (minutes : seconds)')

    plt.ylim(120000, 300000)  # reasonable range for duration of music

    title = 'Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Song Duration'

#     plt.title(title, fontsize=13)
    plt.xlabel('Year')

    # save the image
    plt.savefig(f'figures/timeseries/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

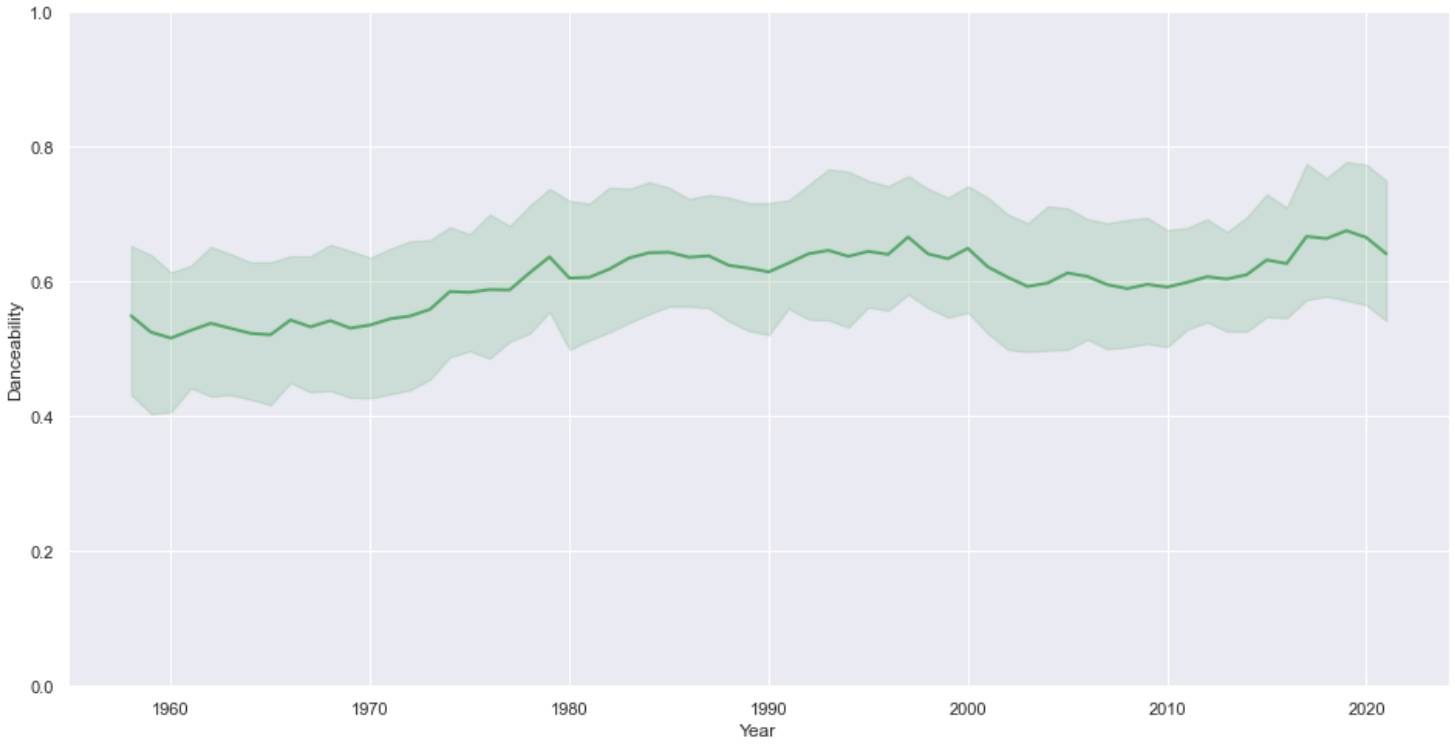
In [42]:

```
# make the plots
list_of_features = ['acousticness',  'danceability', 'energy', 'instrumentalness',
                    'liveness', 'loudness', 'mode', 'speechiness', 'tempo', 'time_signature', 'valence']

for feature in list_of_features:
    plot_billboard_history_by_feature(feature)

plot_billboard_history_duration()
```

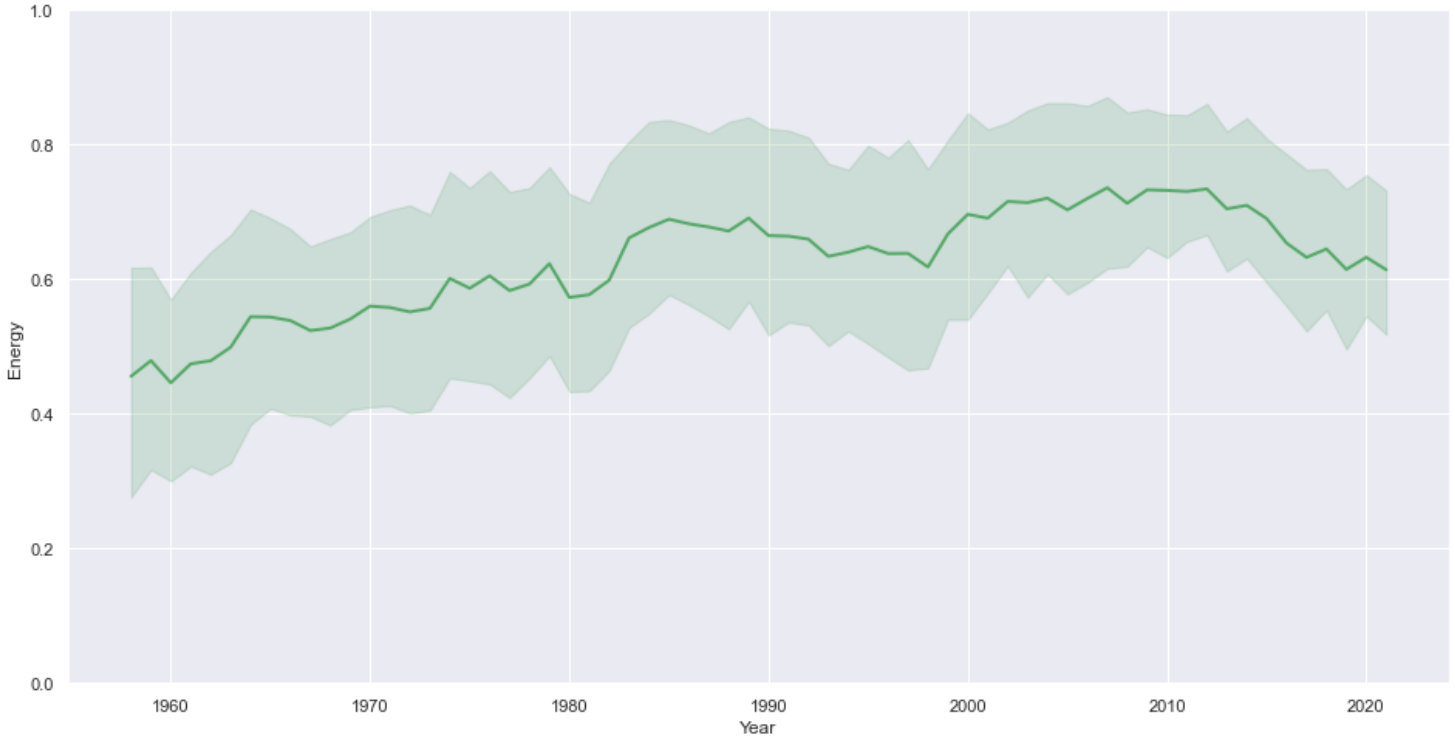
```
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Acousticness
```



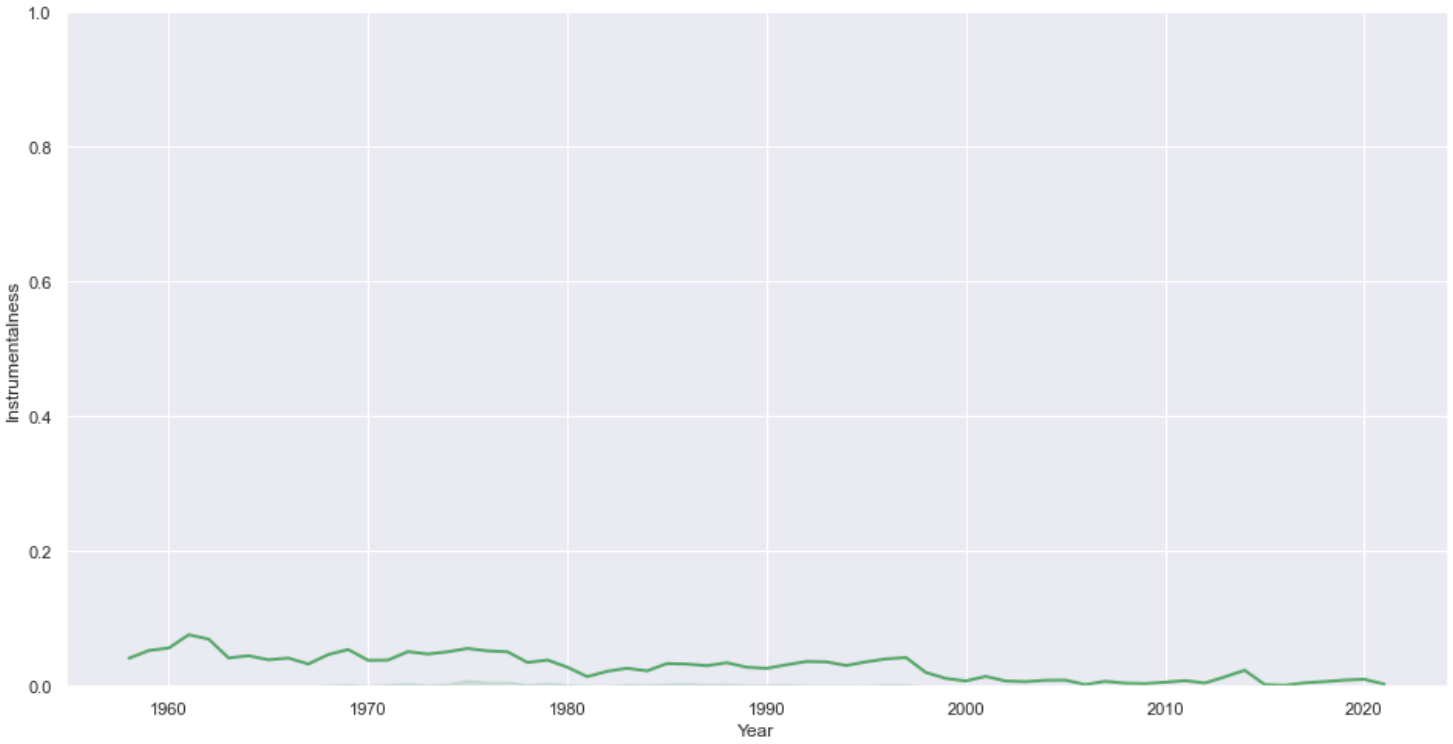

```
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Danceability
```



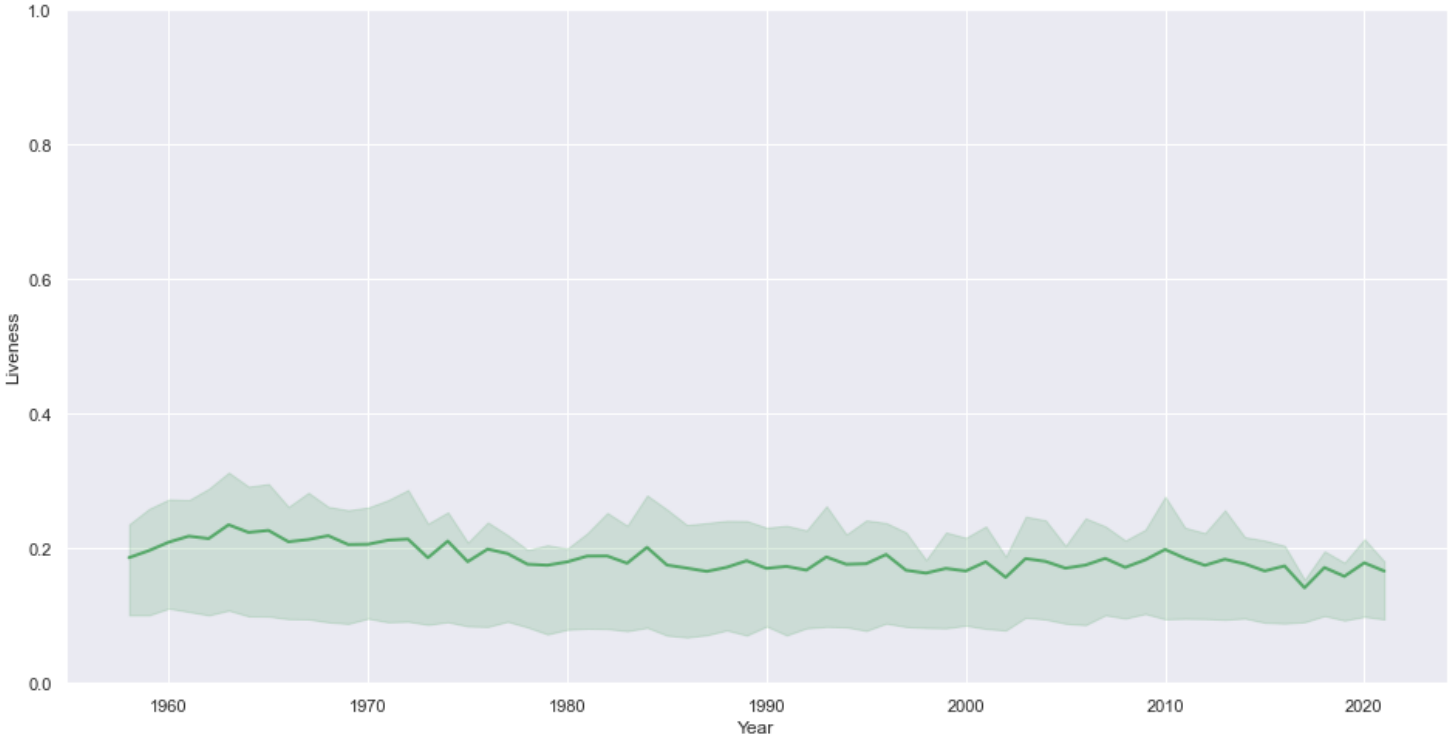
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Energy



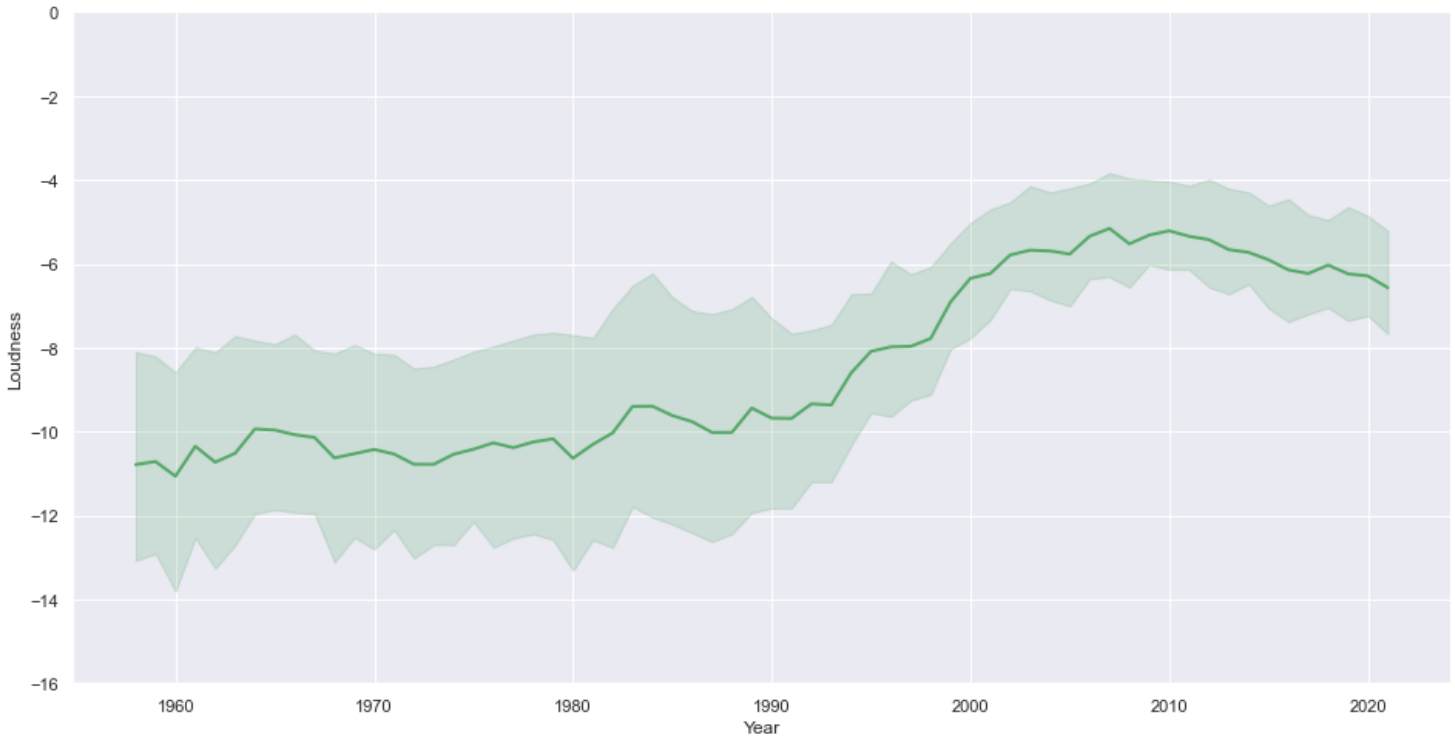
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Instrumentalness



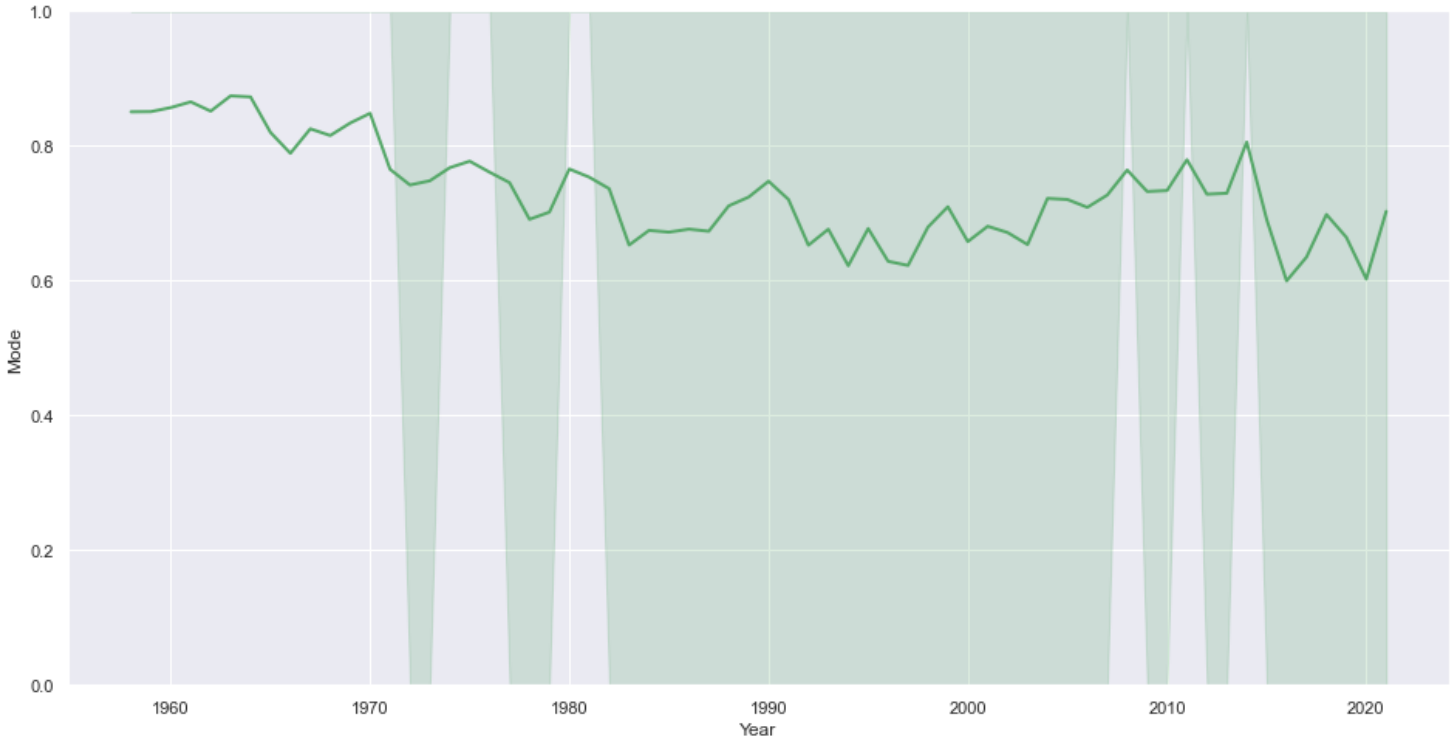
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Liveness



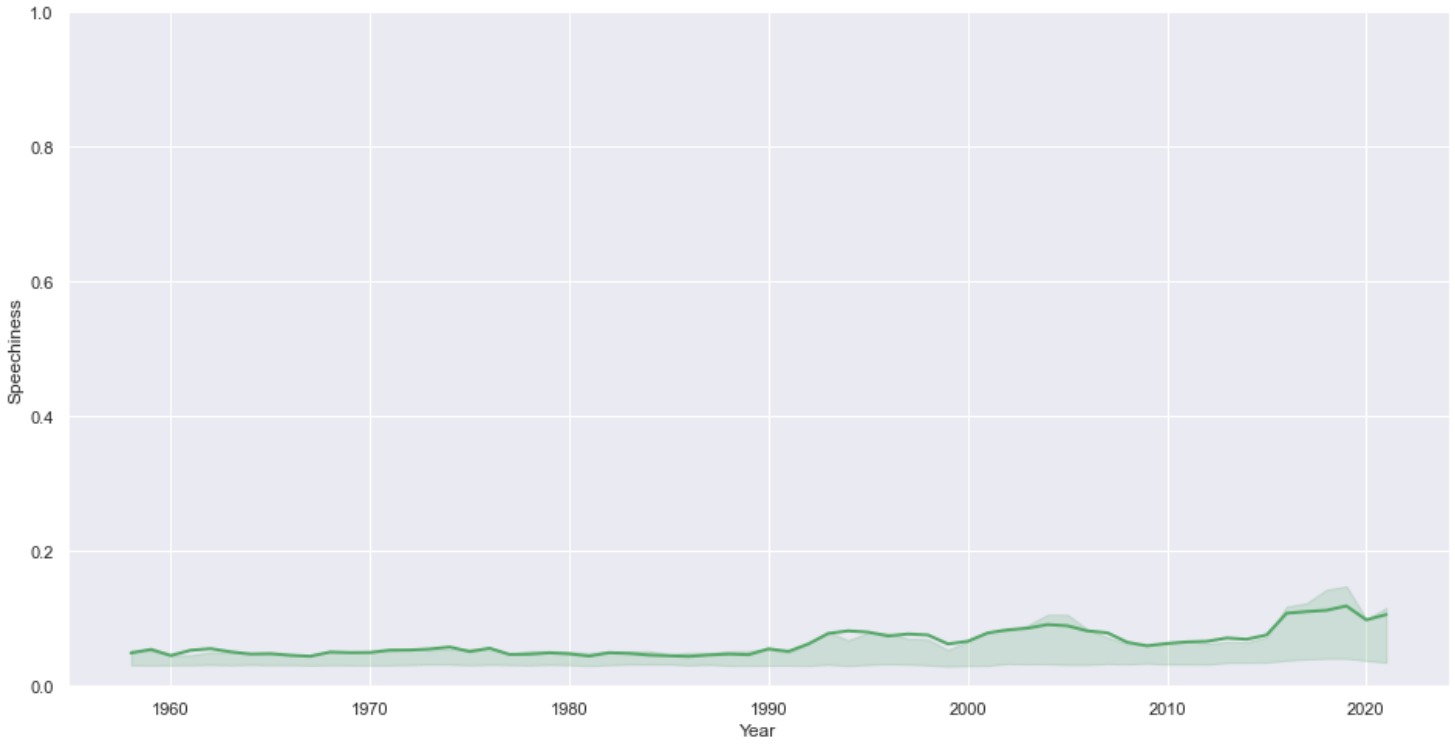
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Loudness



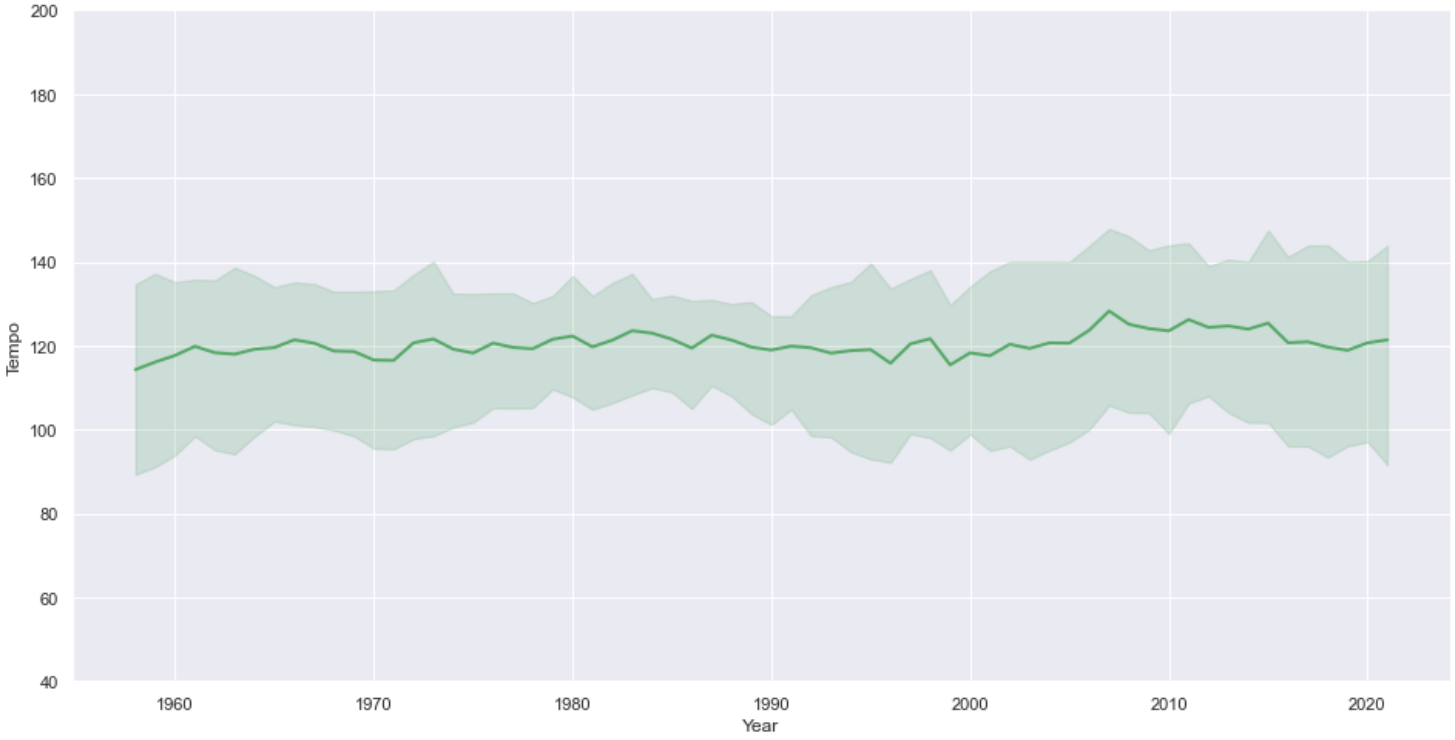
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Mode



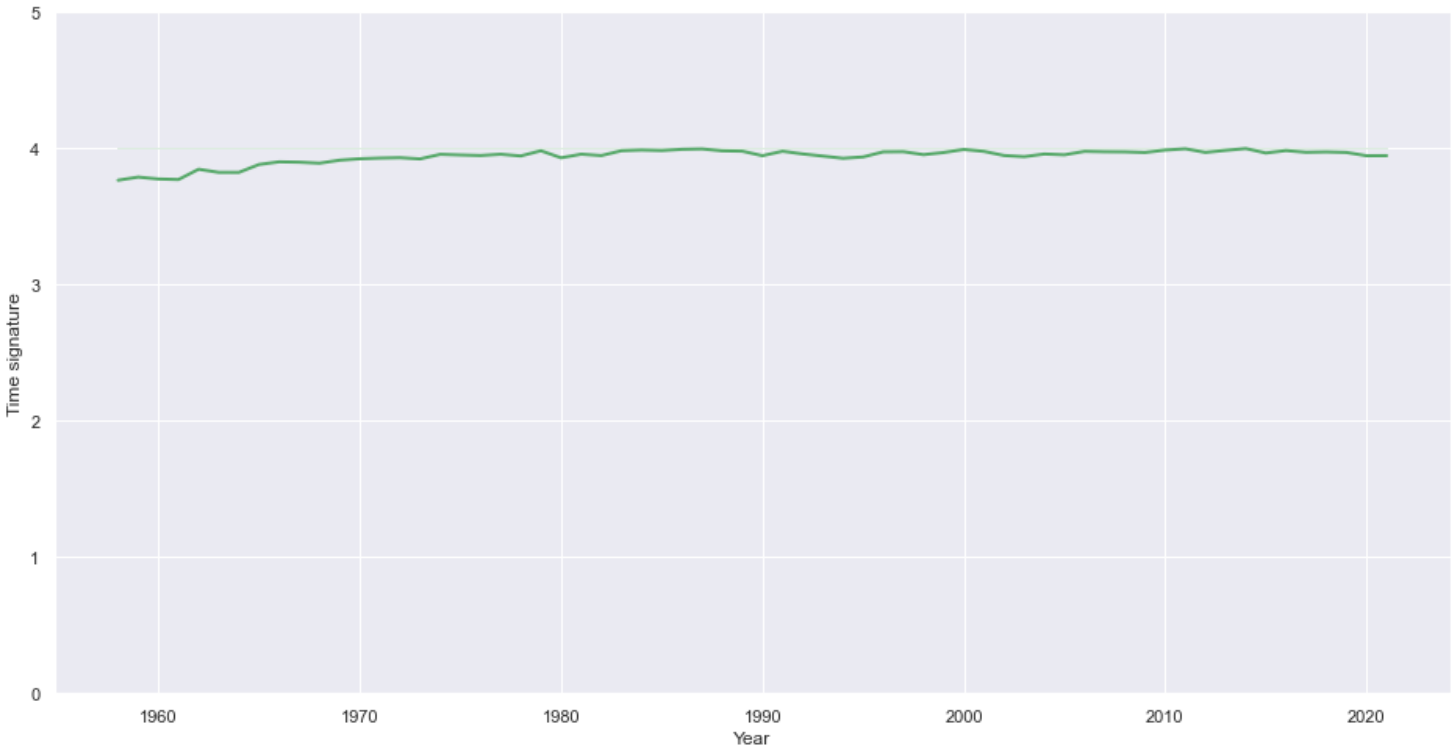
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Speechiness



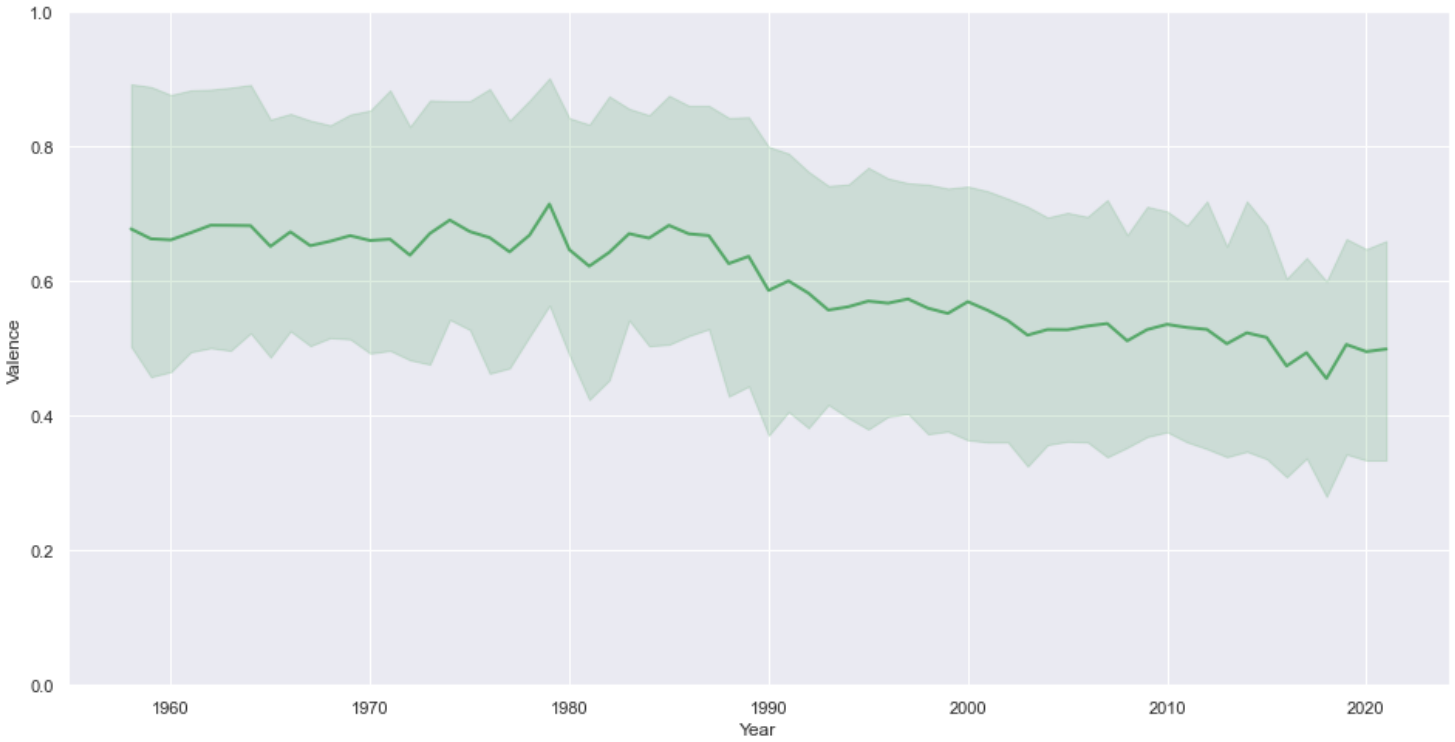
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Tempo



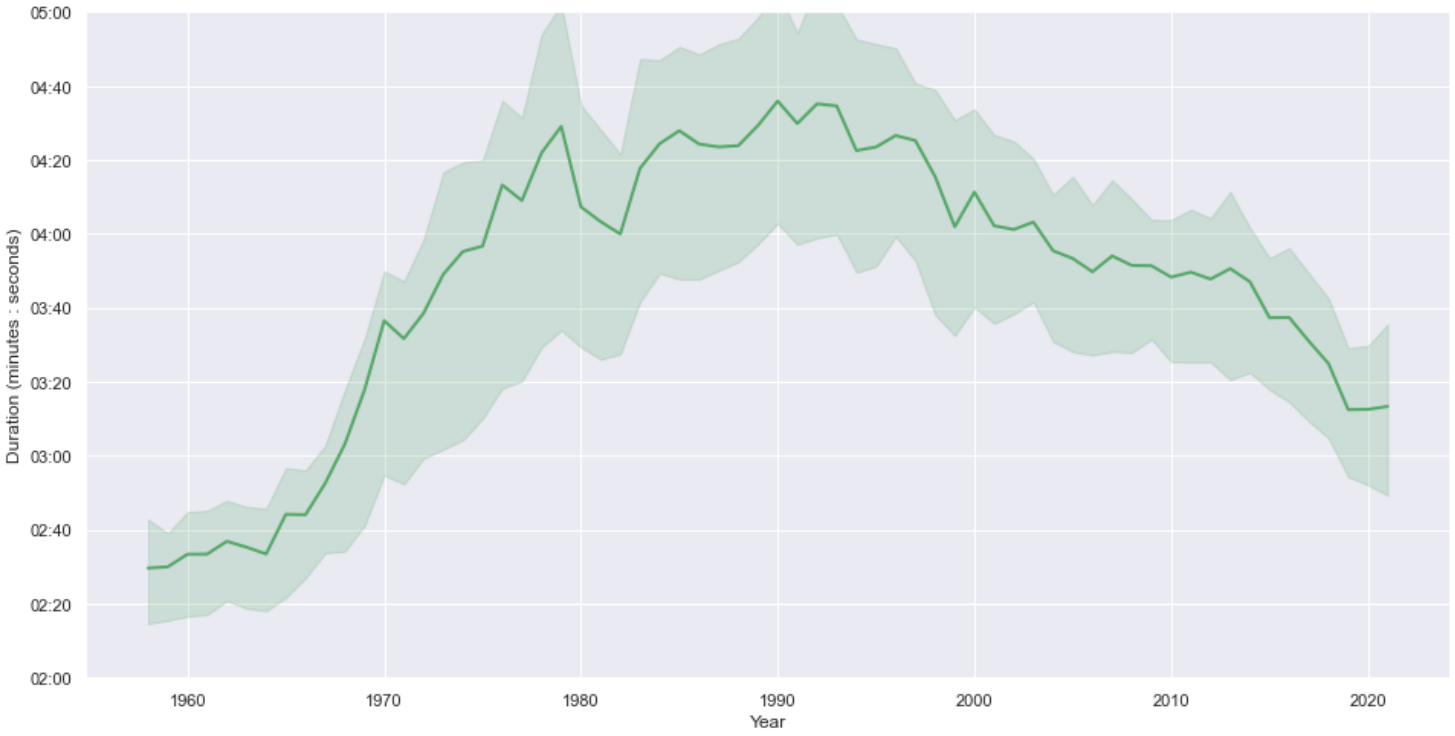
Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Time Signature



Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Valence



Billboard Hot 100 Historical Charts, Interquartile Range Highlighted - Song Duration

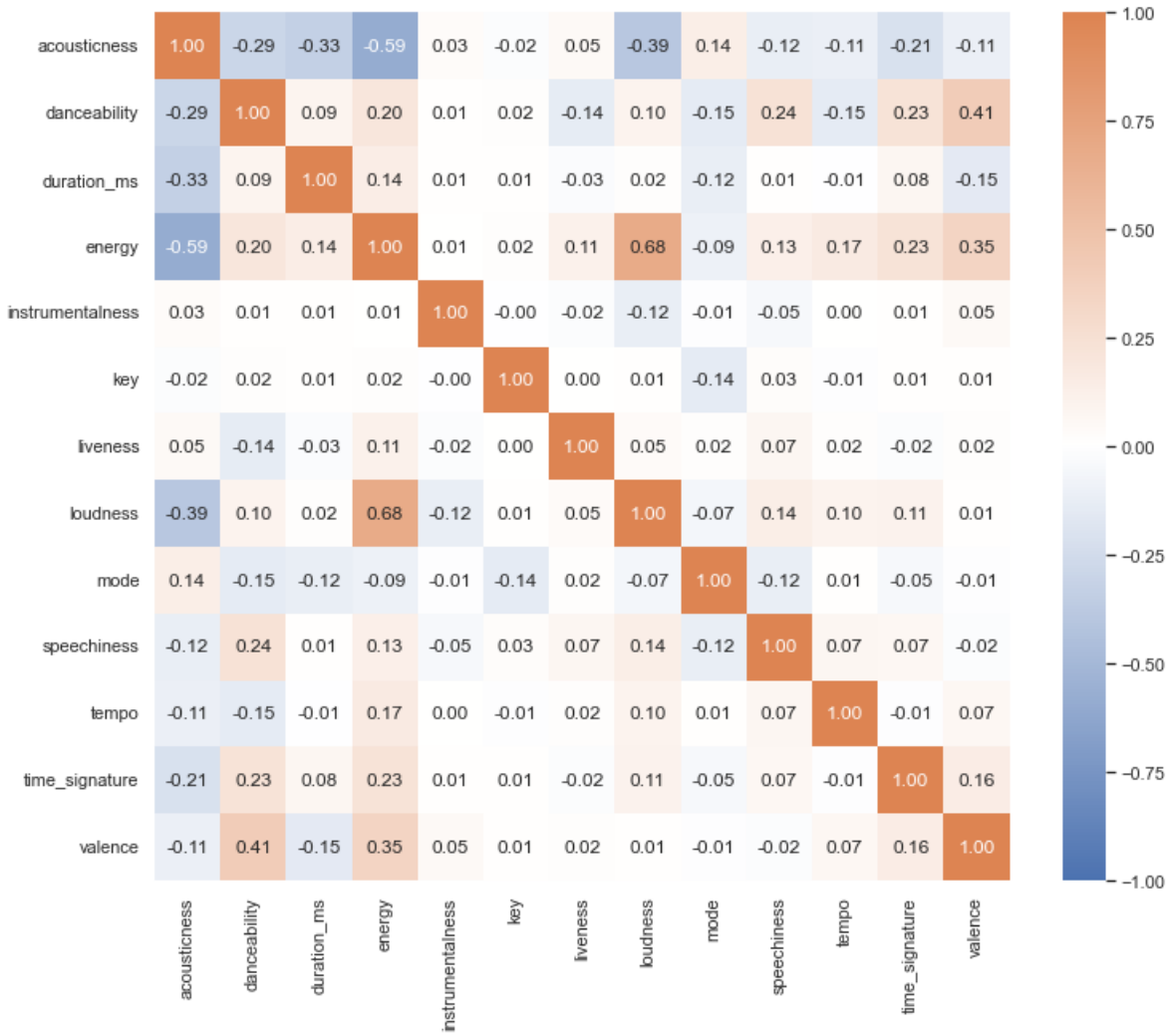


Correlation Analysis

In [43]:

```
# correlation matrix - Billboard Hot 100
corr_B100 = df_B100_songs.corr()

# plot
palette = sns.blend_palette([sns.color_palette()[0], '#ffffff', sns.color_palette()[1]], 3, as_cmap=True)
plt.subplots(figsize=(12, 10))
sns.heatmap(corr_B100, cmap=palette, center=0, vmin=-1, vmax=1, annot=True, fmt='.2f')
title='Correlation of Audio Features - Billboard Hot 100 Songs'
# plt.title(title, fontsize=13, pad=20)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()
```

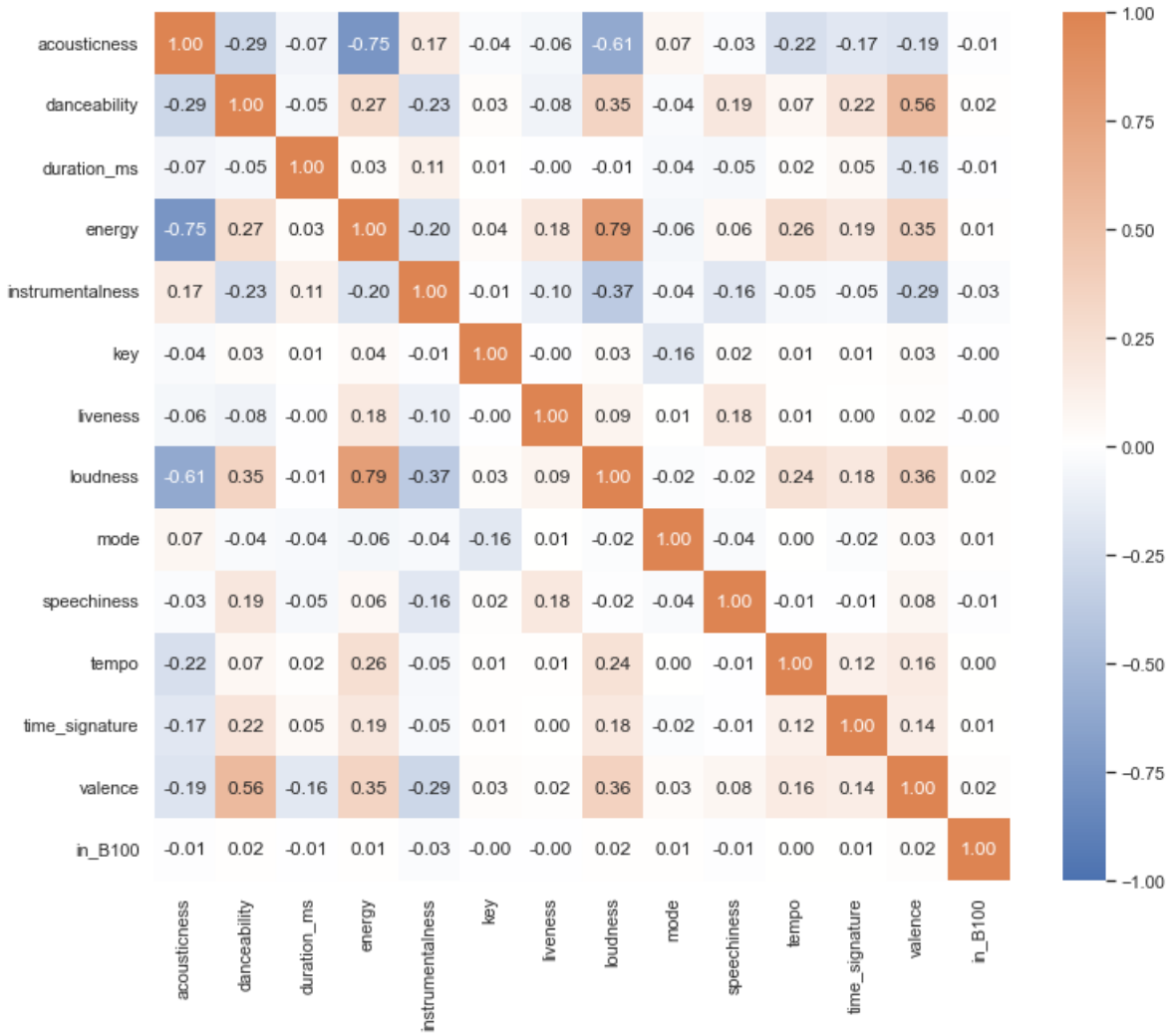
```
Correlation of Audio Features - Billboard Hot 100 Songs
```



In [44]:

```
# correlation matrix - All Songs
corr_10M = df_10M.corr()

# plot
palette = sns.blend_palette([sns.color_palette()[0], '#ffffff', sns.color_palette()[1]], 3, as_cmap=True)
plt.subplots(figsize=(12, 10))
sns.heatmap(corr_10M, cmap=palette, center=0, vmin=-1, vmax=1, annot=True, fmt='.2f')
title='Correlation of Audio Features - All Songs'
# plt.title(title, fontsize=13, pad=20)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()
```

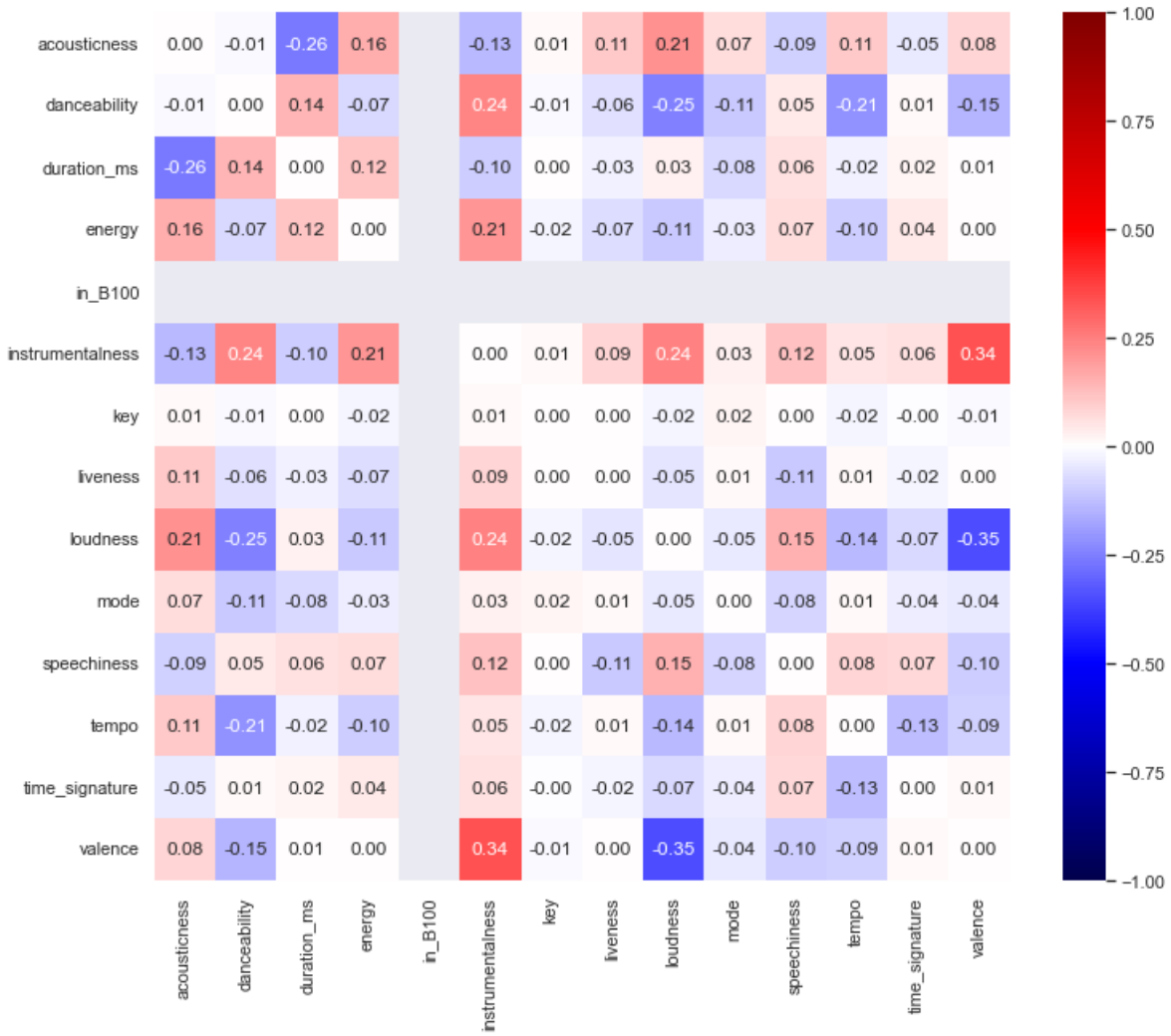
```
Correlation of Audio Features - All Songs
```



In [45]:

```
# differences between Billboard Hot 100 and All Songs
corr_diff = corr_B100 - corr_10M

# plot
palette = 'seismic'  # to differentiate from standard colour palette
plt.subplots(figsize=(12, 10))
sns.heatmap(corr_diff, cmap=palette, center=0, vmin=-1, vmax=1, annot=True, fmt='.2f')
title='Difference in Correlation of Audio Features - Billboard Hot 100 Minus All Songs'
# plt.title(title, fontsize=13, pad=20)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()
```

```
Difference in Correlation of Audio Features - Billboard Hot 100 Minus All Songs
```



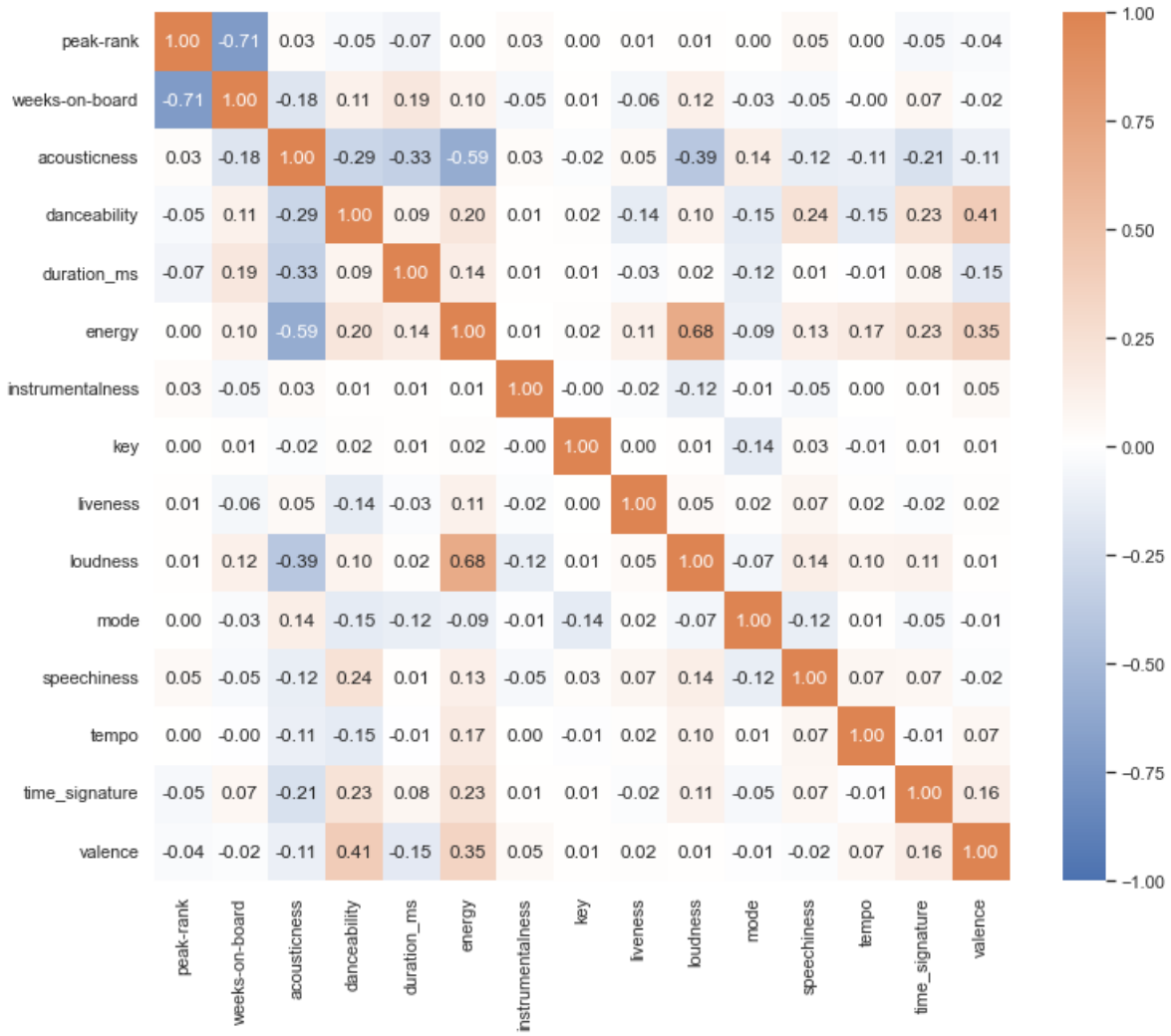
In [46]:

```
# Billboard 100 List - correlation with max weeks on board
df_B100_SORTED = df_B100.sort_values('weeks-on-board', ascending=False).drop_duplicates(subset=['song', 'artist']).drop(['date
```

In [47]:

```
# correlation matrix - All Songs
corr_billboard_weeks = df_B100_SORTED.corr()

# plot
palette = sns.blend_palette([sns.color_palette()[0], '#ffffff', sns.color_palette()[1]], 3, as_cmap=True)
plt.subplots(figsize=(12, 10))
sns.heatmap(corr_billboard_weeks, cmap=palette, center=0, vmin=-1, vmax=1, annot=True, fmt='.2f')
title='Correlation of Billboard Rankings to Audio Features'
# plt.title(title, fontsize=13, pad=20)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()
```

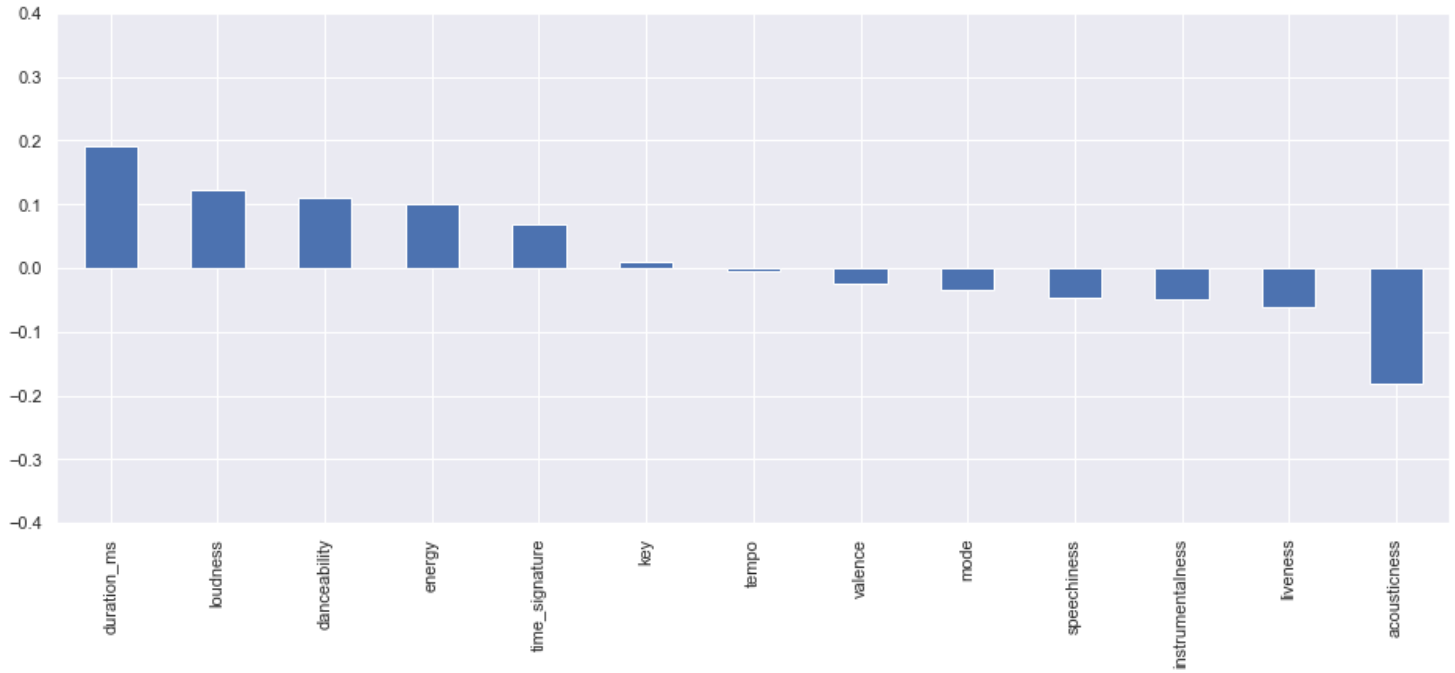
```
Correlation of Billboard Rankings to Audio Features
```



In [48]:

```
# correlated with weeks on board in Billboard Hot 100
corr_billboard_weeks['weeks-on-board'].sort_values(ascending=False)[1:-1].plot(kind='bar', figsize=(16,6))
title='Correlation of Weeks on Billboard Hot 100 vs Audio Features'
# plt.title(title, fontsize=13, pad=20)
plt.ylim(-0.4, 0.4)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()
```

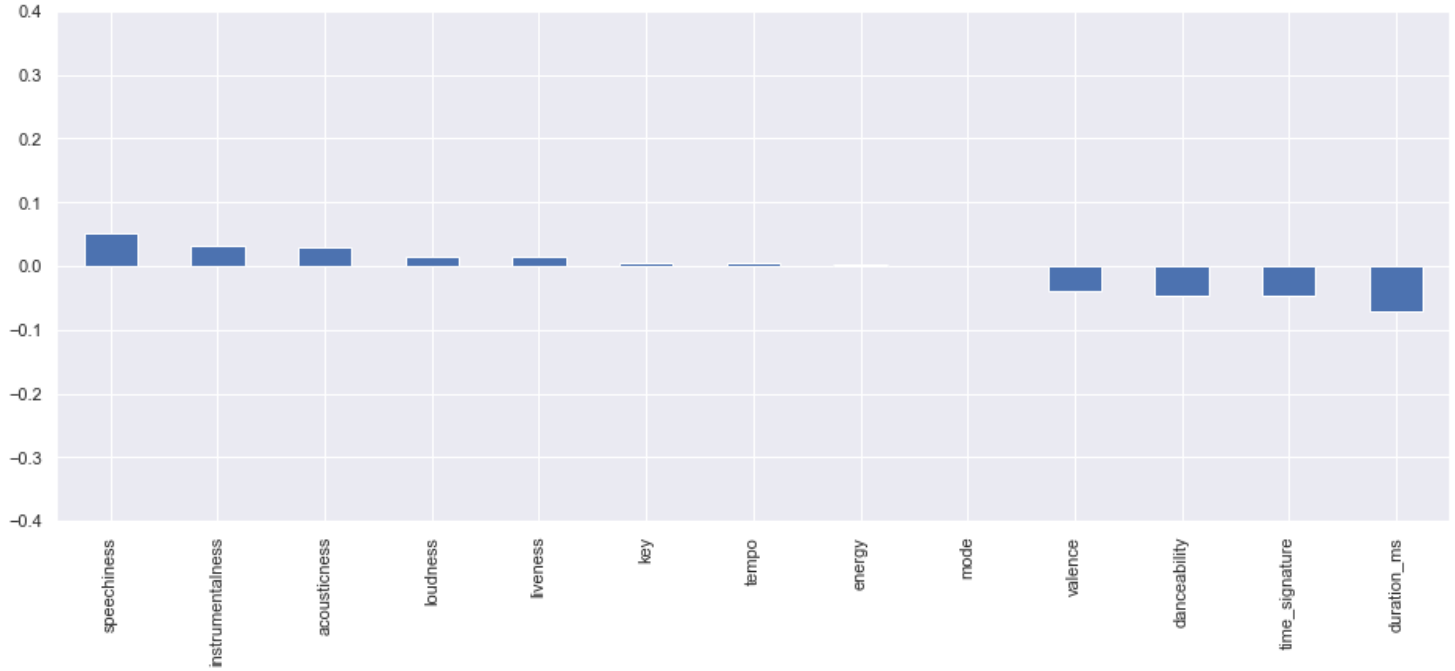
```
Correlation of Weeks on Billboard Hot 100 vs Audio Features
```



In [49]:

```
# correlated with peak rank in Billboard Hot 100
corr_billboard_weeks['peak-rank'].sort_values(ascending=False)[1:-1].plot(kind='bar', figsize=(16,6))
title='Correlation of Peak Rank on Billboard Hot 100 vs Audio Features'
# plt.title(title, fontsize=13, pad=20)
plt.ylim(-0.4, 0.4)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()
```

Correlation of Peak Rank on Billboard Hot 100 vs Audio Features

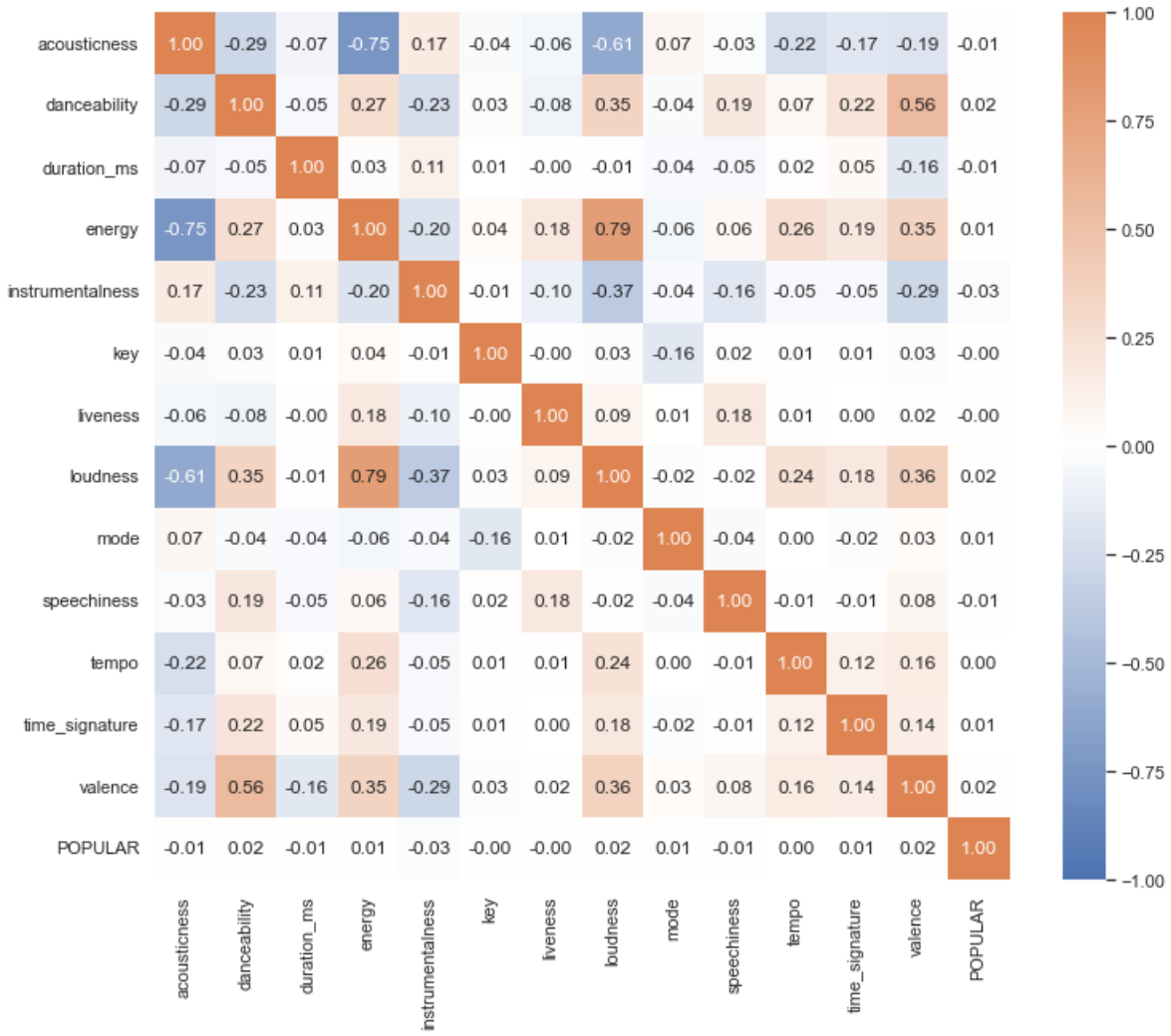


In [50]:

```
# correlation matrix - popular or not
# Note: "POPULAR" simply denotes whether or not a song appeared on the Billboard Hot 100
corr_popular = df_popularity.corr()

# plot
palette = sns.blend_palette([sns.color_palette()[0], '#ffffff', sns.color_palette()[1]], 3, as_cmap=True)
```

```
plt.subplots(figsize=(12, 10))
sns.heatmap(corr_popular, cmap=palette, center=0, vmin=-1, vmax=1, annot=True, fmt='.2f')
title='Correlation of Popularity with Audio Features'
# plt.title(title, fontsize=13, pad=20)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()
```

```
Correlation of Popularity with Audio Features
```

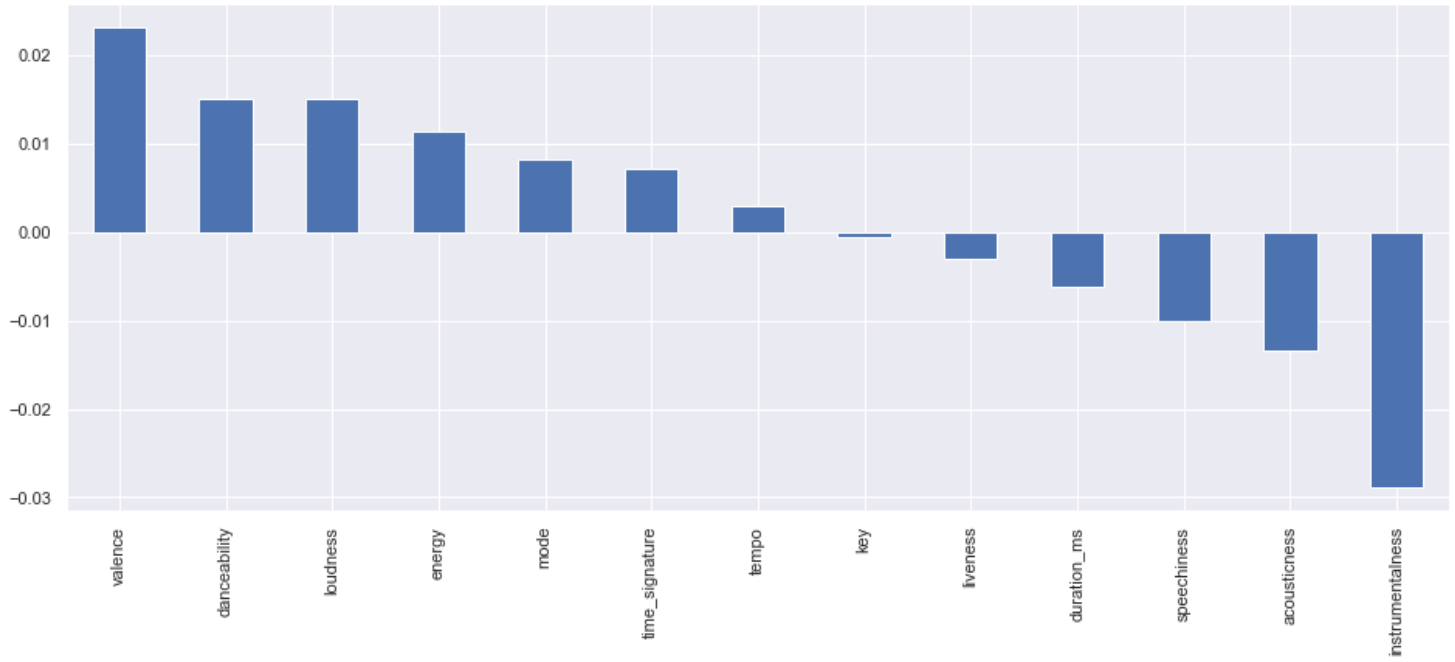


In [51]:

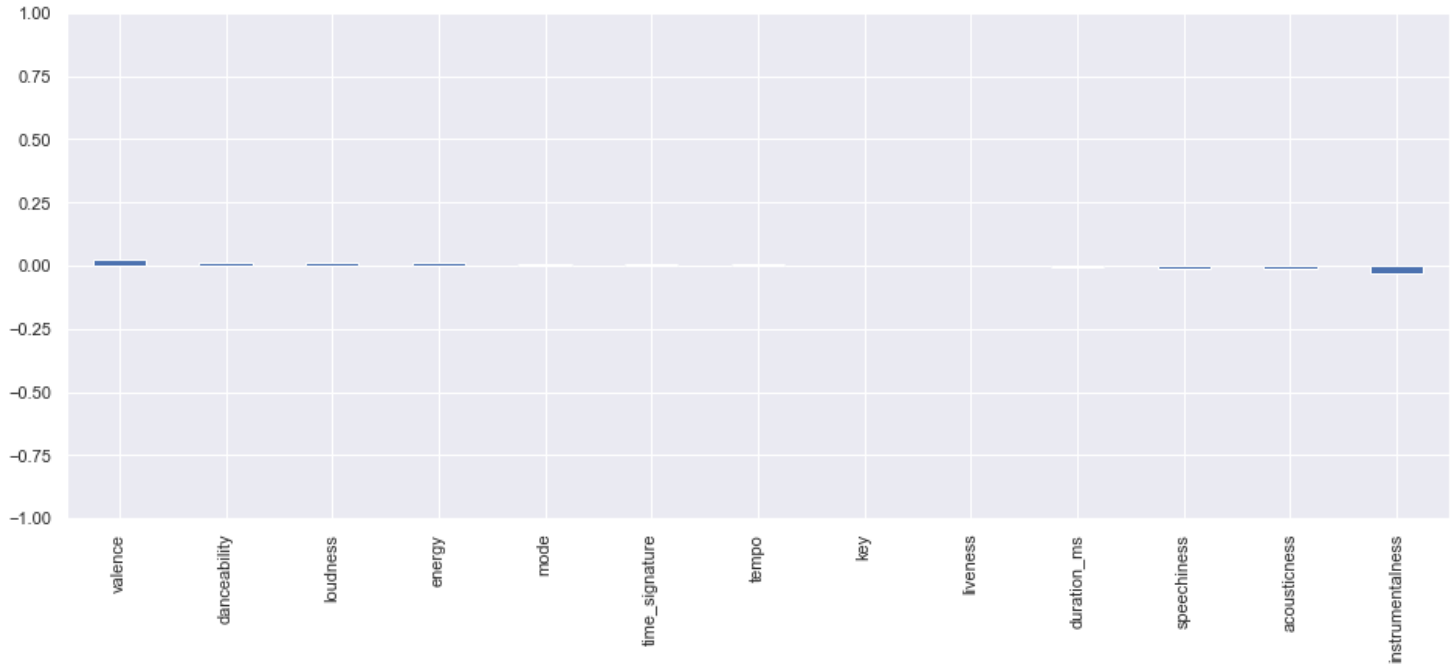
```
# correlated with peak rank in Billboard Hot 100
corr_popular['POPULAR'].sort_values(ascending=False)[1:].plot(kind='bar', figsize=(16,6))
title='Correlation of Popularity with Audio Features'
# plt.title(title, fontsize=13, pad=20)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)

plt.show()

corr_popular['POPULAR'].sort_values(ascending=False)[1:].plot(kind='bar', figsize=(16,6))
title='Correlation of Popularity with Audio Features (Full Scale)'
# plt.title(title, fontsize=13, pad=20)
plt.ylim(-1, 1)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')
```

```
# print the title
print(title)

plt.show()
```

```
Correlation of Popularity with Audio Features
```




```
Correlation of Popularity with Audio Features (Full Scale)
```

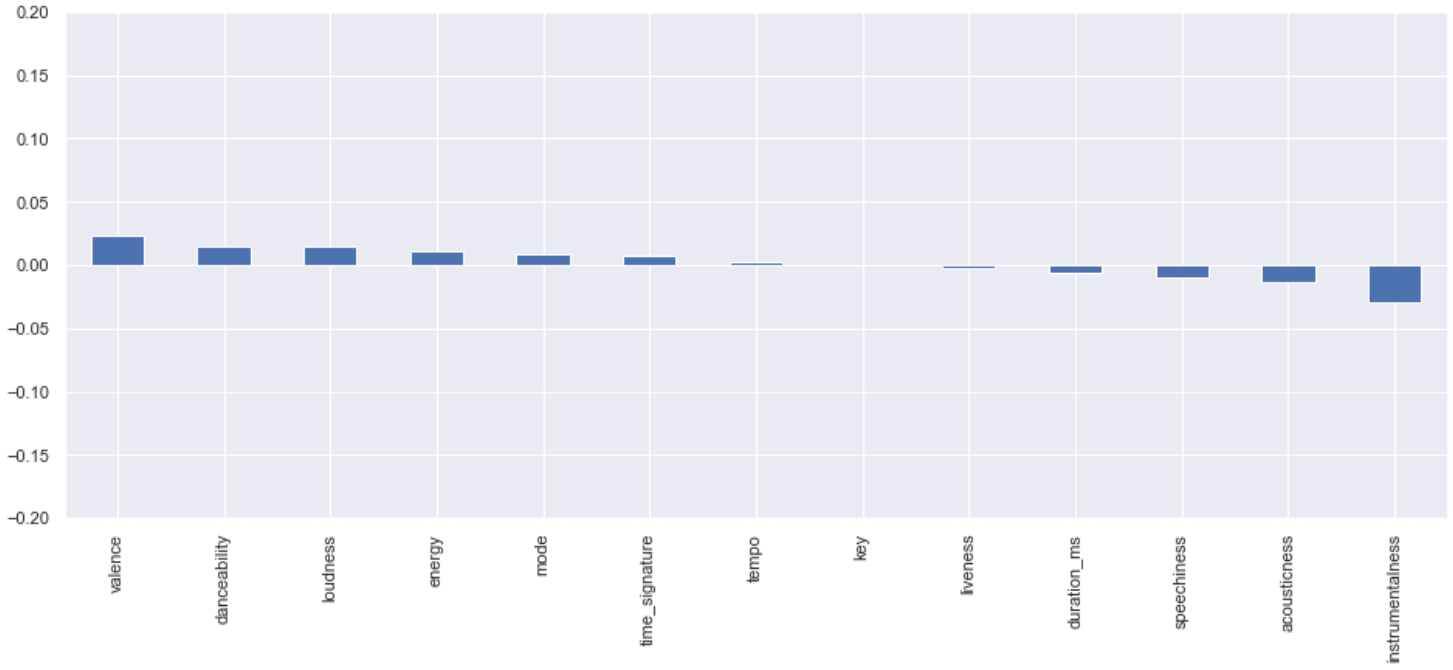


In [52]:

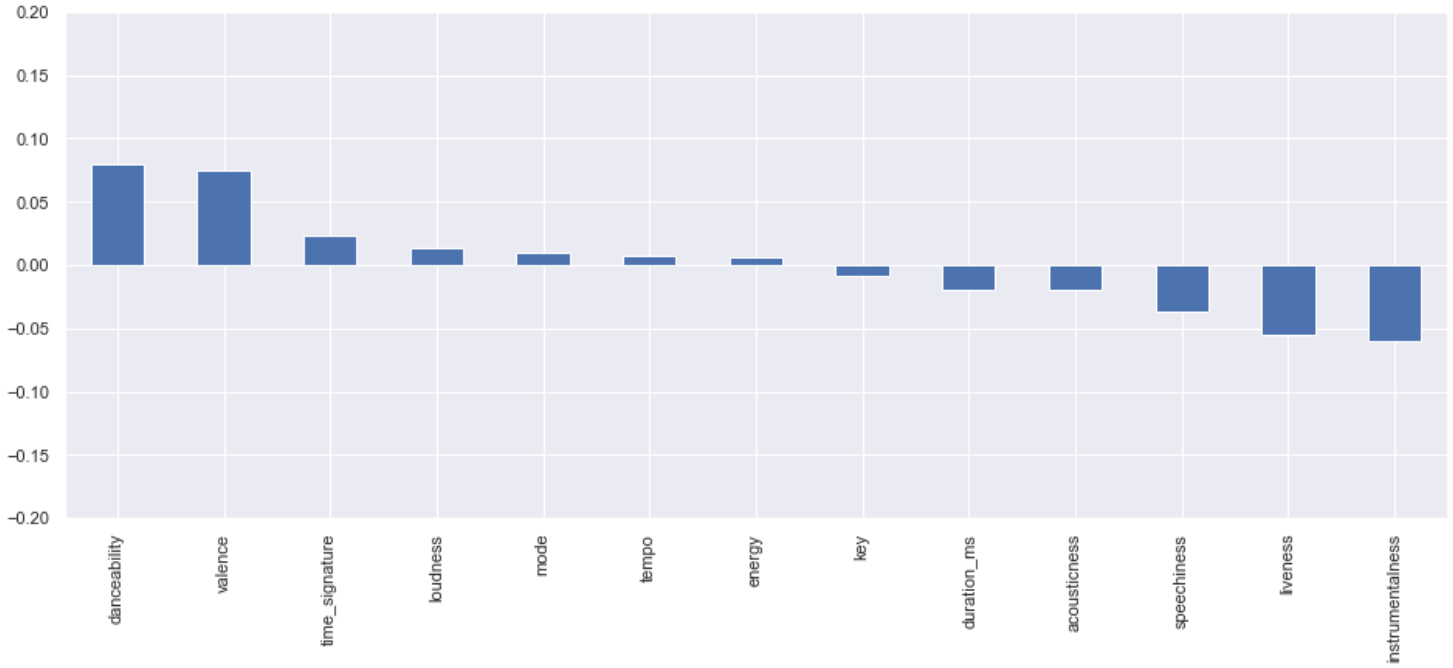
```
# top 10 genres in Billboard Hot 100
genres = list(df_B100_songs_genre.genre.value_counts().head(10).index)

# shared y limits
ylim = -0.2, 0.2

# all genres
df_popularity.corr()['POPULAR'].sort_values(ascending=False)[1:].plot(kind='bar', figsize=(16,6))
title='Correlation of Popularity with Audio Features, All Genres Combined'
# plt.title(title, fontsize=13, pad=20)
plt.ylim(*ylim)

# save the image
plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

# print the title
print(title)
```

```
plt.show()

# separated genres
for genre in genres:
    df_popularity.query(f'genre == "{genre}"').corr()['POPULAR'].sort_values(ascending=False)[1:].plot(kind='bar', figsize=(16
    title = f'Correlation of Popularity with Audio Features, Filtered by Genre - {genre.title()}'
#     plt.title(title, fontsize=13, pad=20)
    plt.ylim(*ylim)

    # save the image
    plt.savefig(f'figures/correlation/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

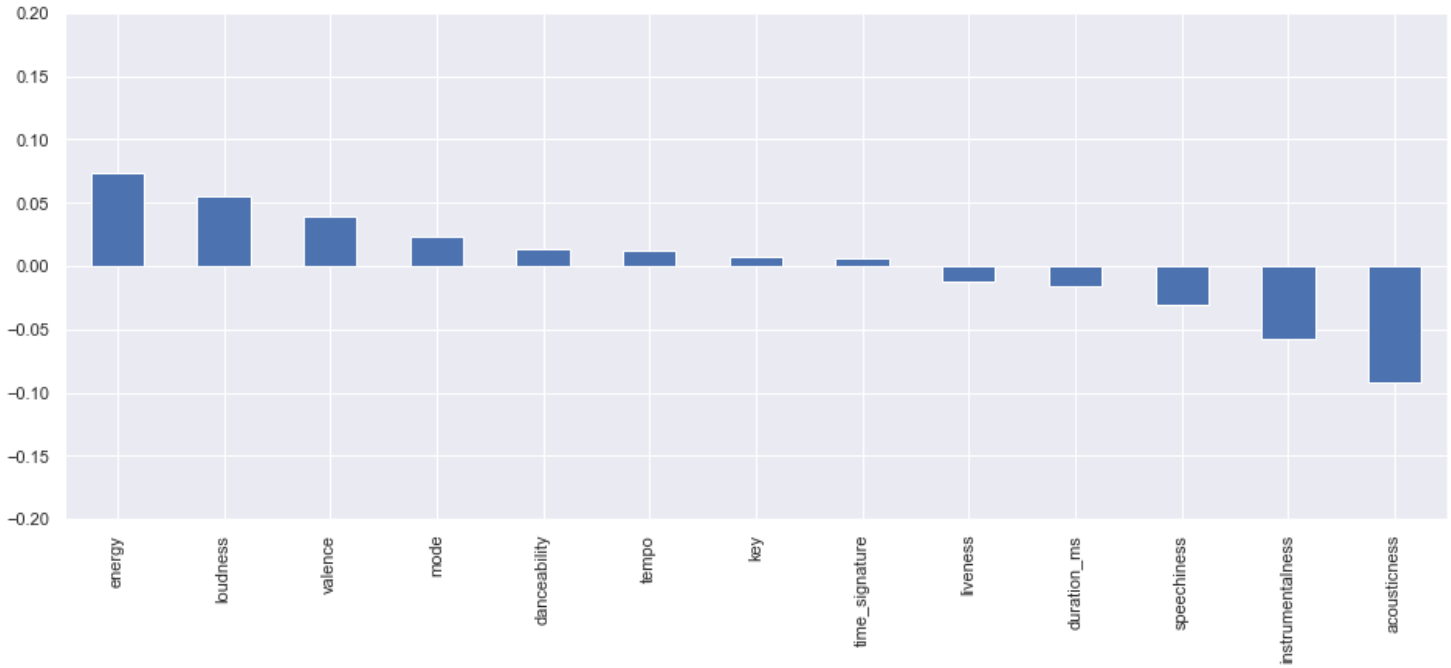
Correlation of Popularity with Audio Features, All Genres Combined



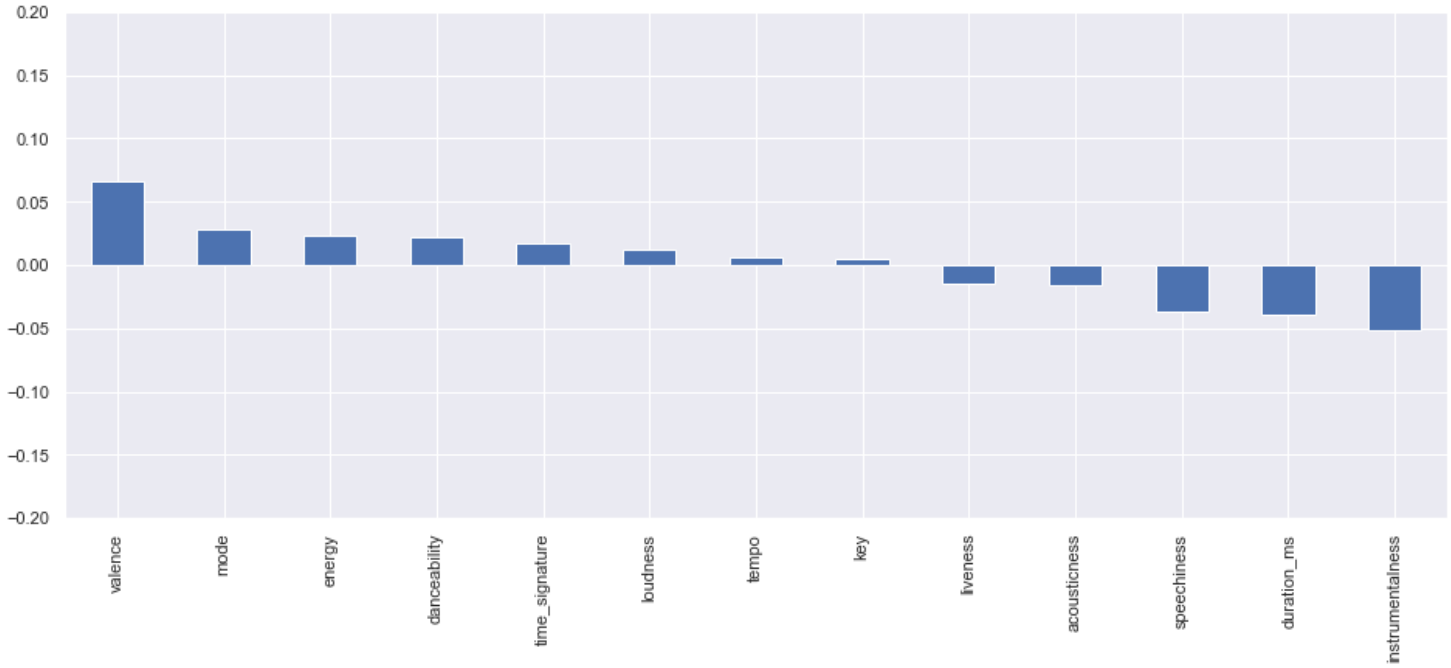
Correlation of Popularity with Audio Features, Filtered by Genre - Rock



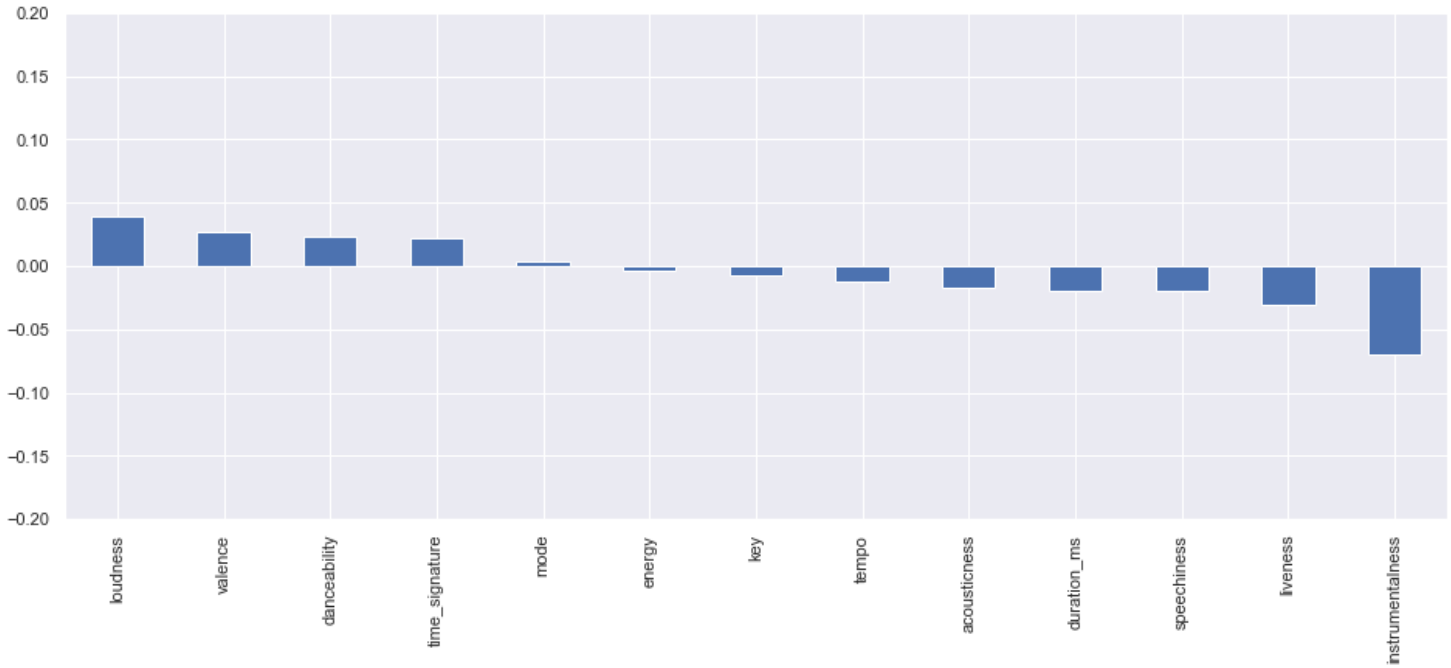
Correlation of Popularity with Audio Features, Filtered by Genre - Adult Standards



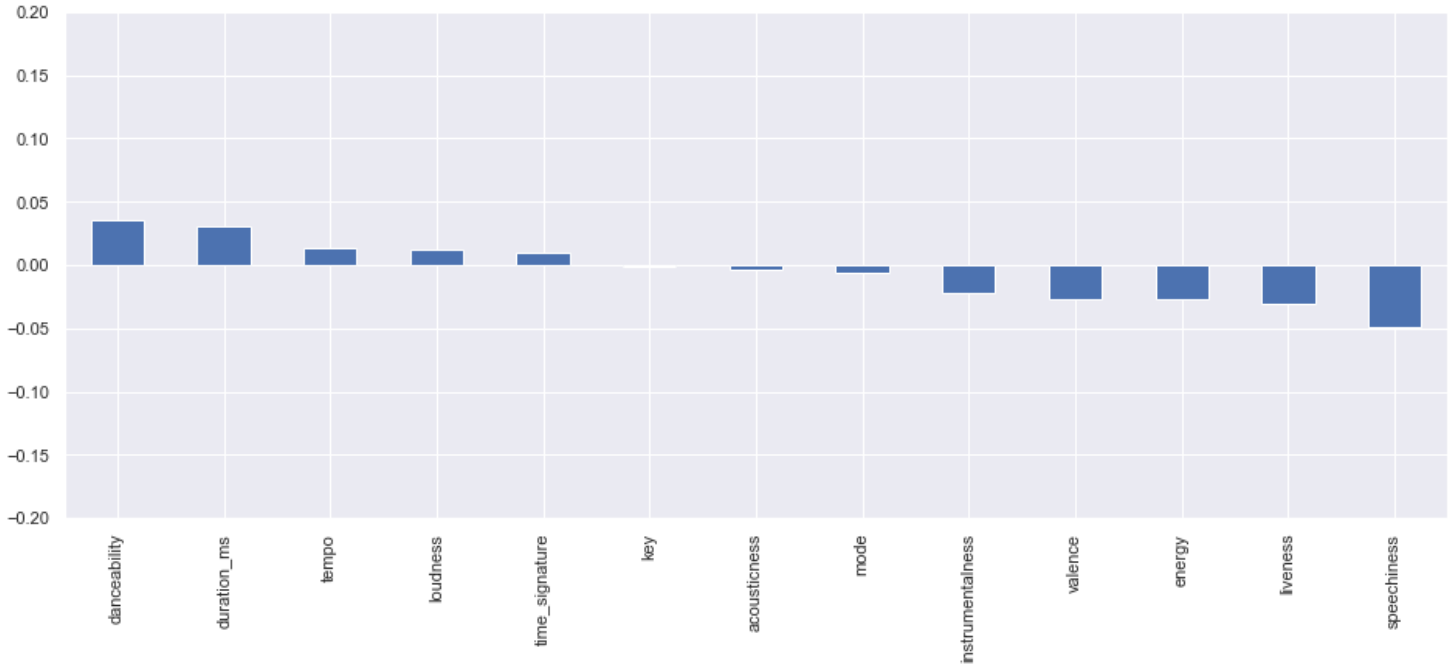
Correlation of Popularity with Audio Features, Filtered by Genre - Soul



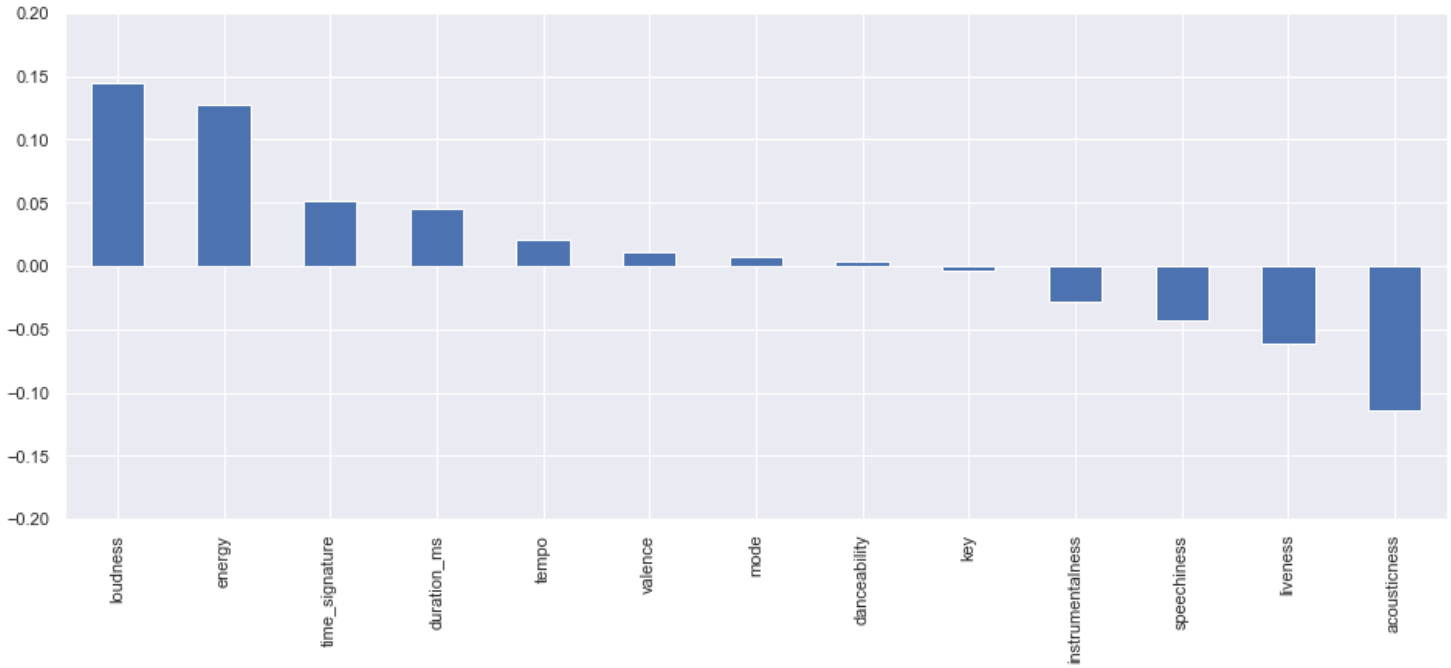
Correlation of Popularity with Audio Features, Filtered by Genre - Dance Pop



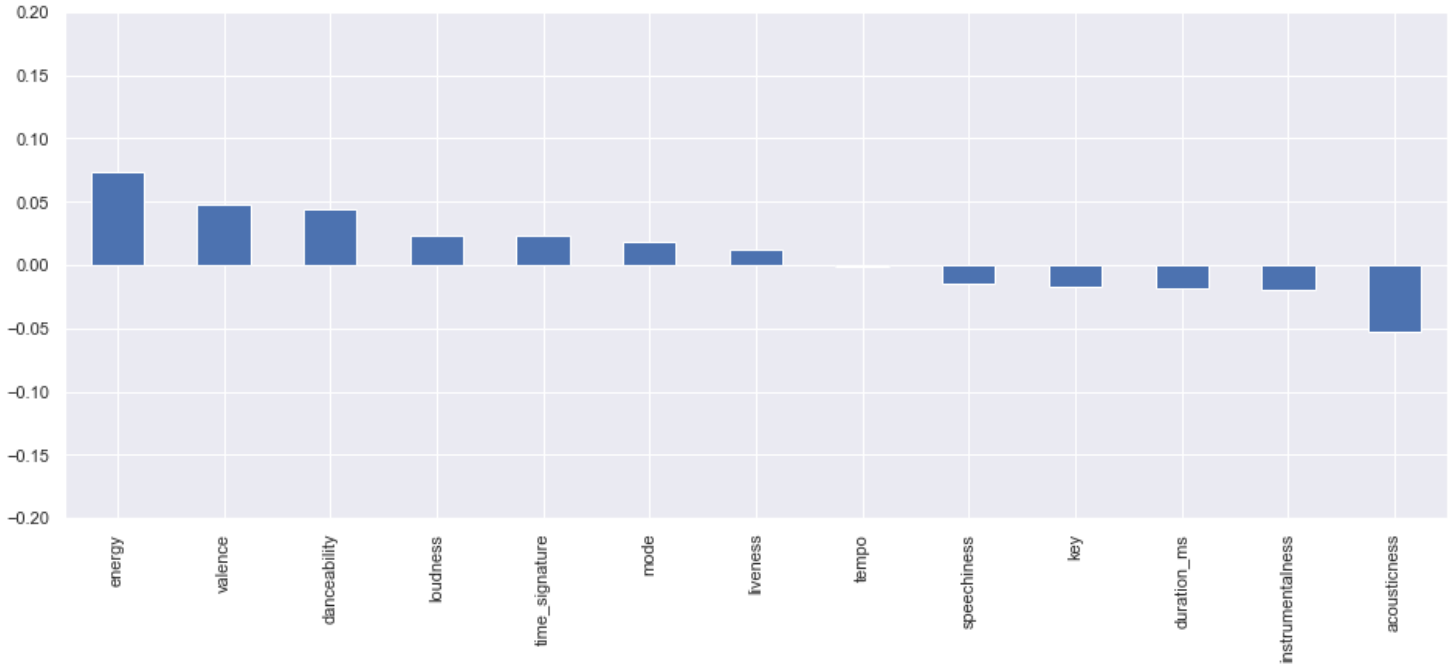
Correlation of Popularity with Audio Features, Filtered by Genre - Hip Hop



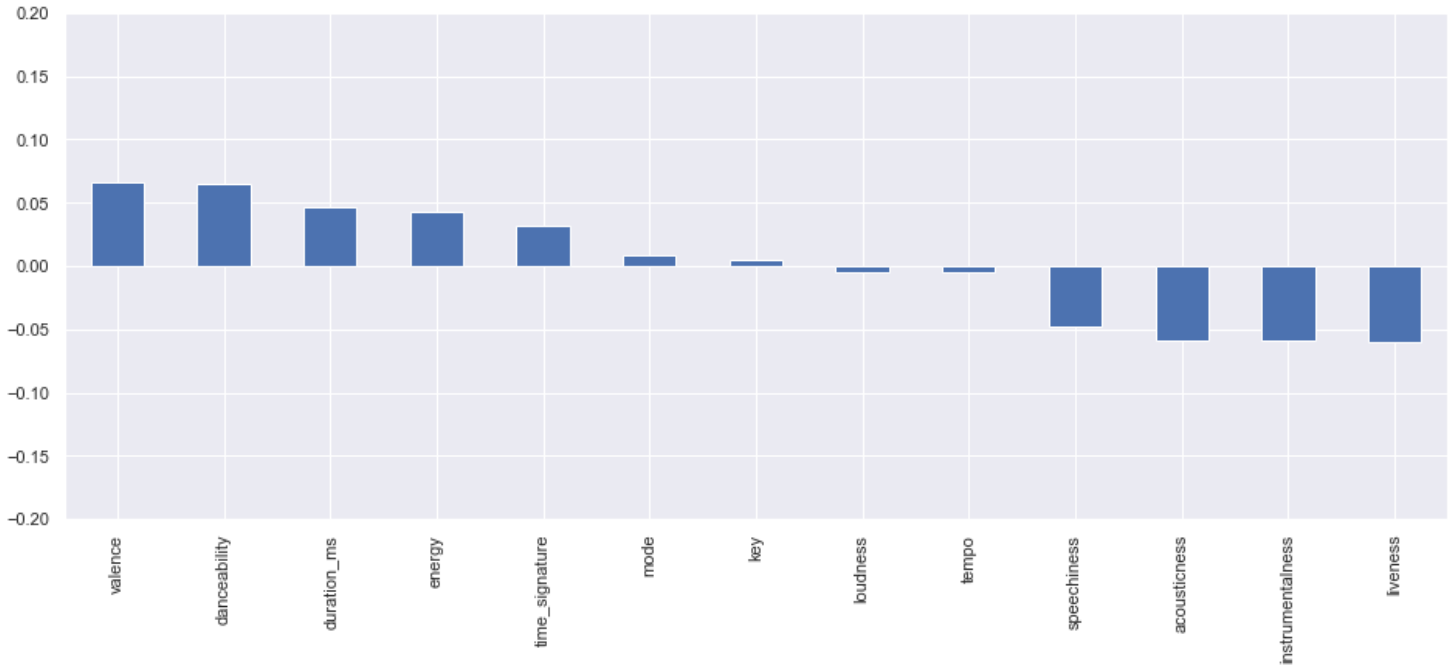
Correlation of Popularity with Audio Features, Filtered by Genre - Country



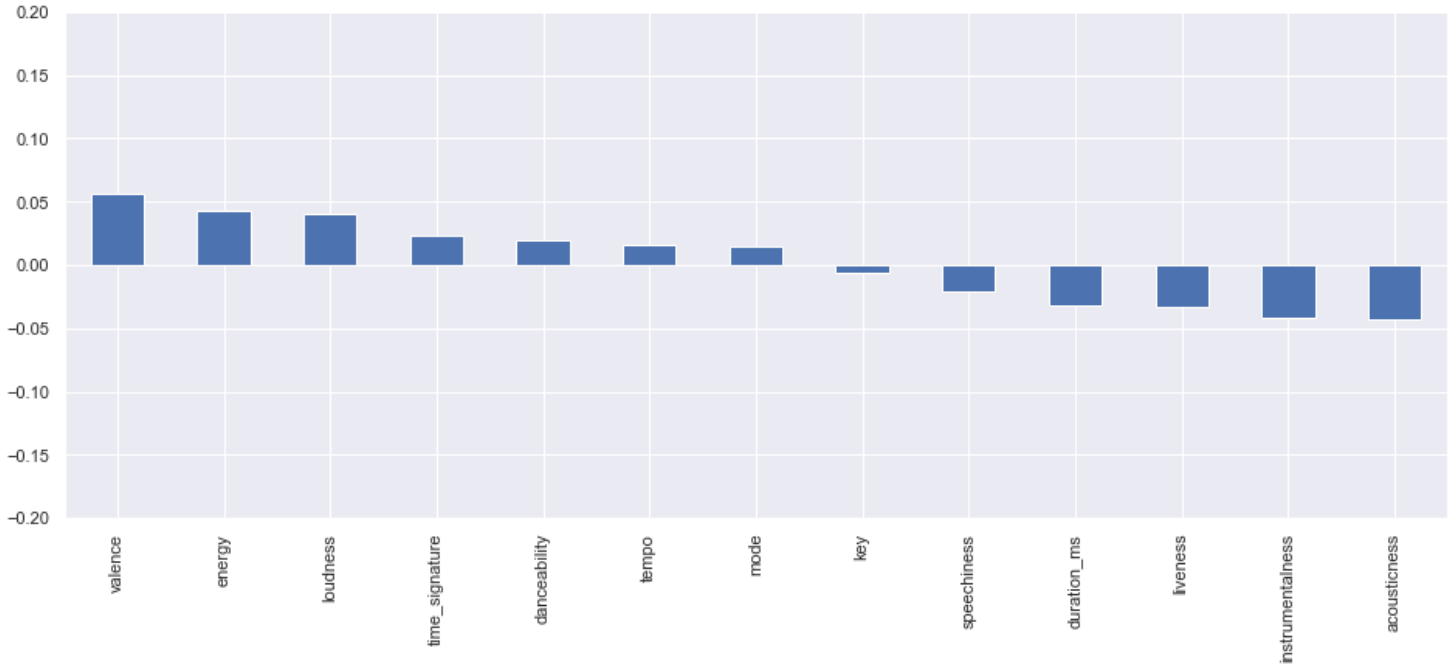
Correlation of Popularity with Audio Features, Filtered by Genre - Brill Building Pop



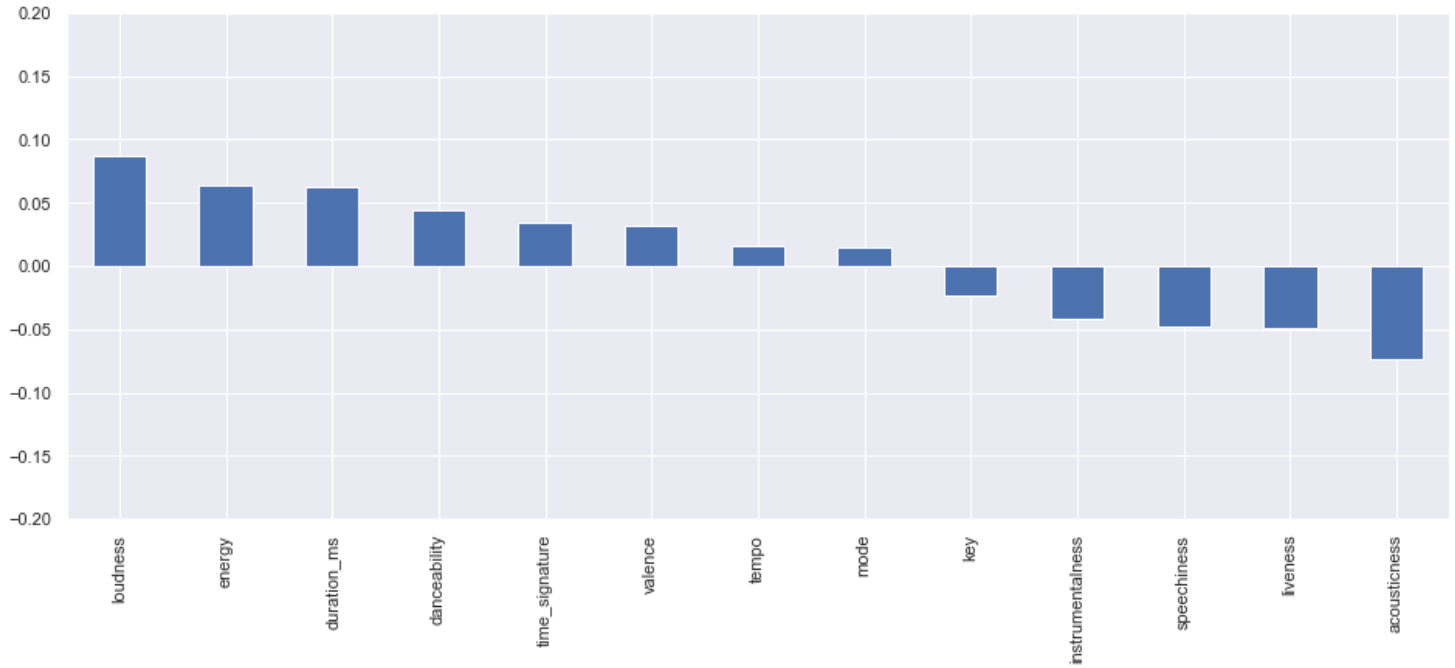
Correlation of Popularity with Audio Features, Filtered by Genre - Mellow Gold



Correlation of Popularity with Audio Features, Filtered by Genre - Folk Rock

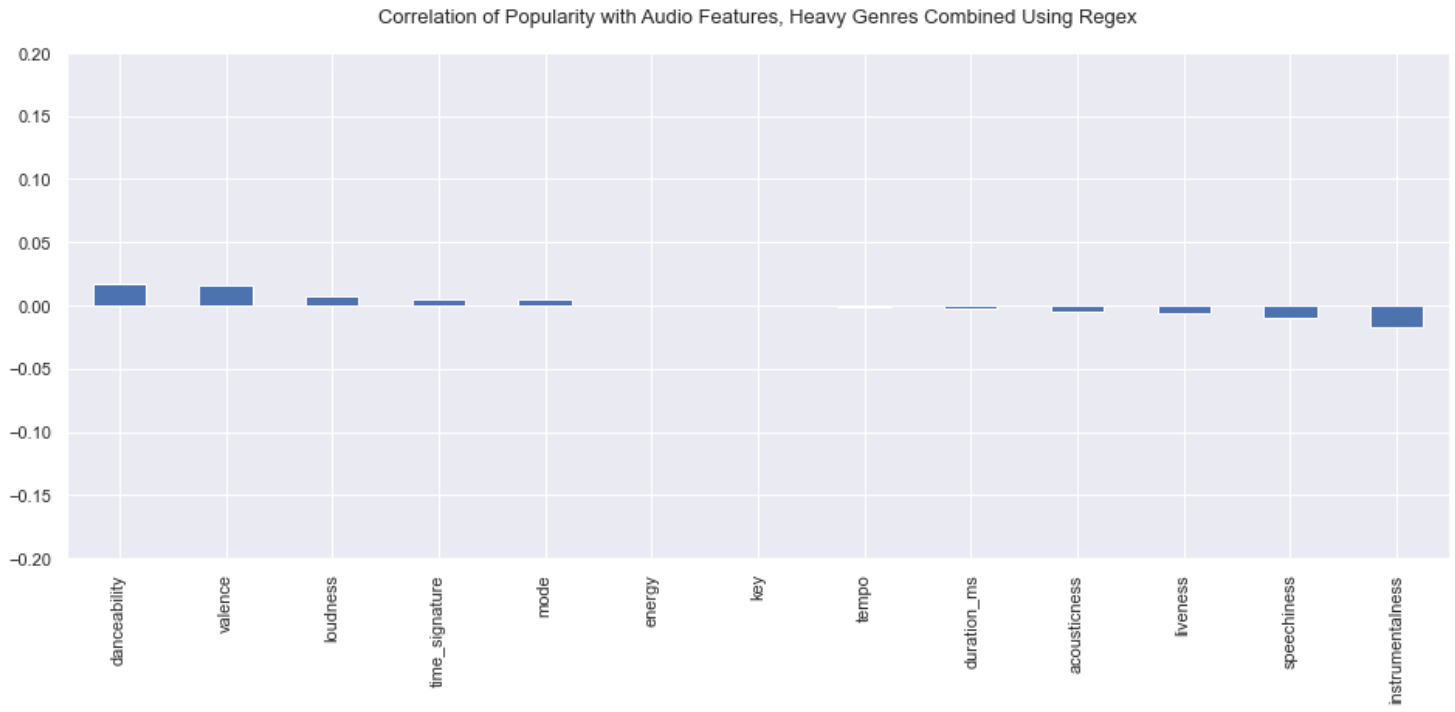


Correlation of Popularity with Audio Features, Filtered by Genre - Pop



In [53]:
```
# testing with regex to look for ability to categorise more broadly

heavy_regex = df_popularity[df_popularity.genre.str.contains(r'(?:metal|heavy|hard|alternative)').fillna(False)]
```

In [54]:
```
# shared y limits
ylim = -0.2, 0.2

# all genres
heavy_regex.corr()['POPULAR'].sort_values(ascending=False)[1:].plot(kind='bar', figsize=(16,6))
title='Correlation of Popularity with Audio Features, Heavy Genres Combined Using Regex'
plt.title(title, fontsize=13, pad=20)
plt.ylim(*ylim)
plt.show()
```



In [55]:
```
# there is some interesting stuff here!
heavy_regex.genre.sample(10)
```

Out[55]:
```
2592639             alternative metal
3641186                         metal
1483120                 turkish metal
9325467           alternative hip hop
2550169    christian alternative rock
7884678        alternative roots rock
521233                  scottish metal
7451822           swedish heavy metal
3519520                 post-hardcore
1921771               alternative r&b
Name: genre, dtype: object
```

Genres

Import Genre Data

In [56]:
```
df_genre_counts.shape[0]
```

Out[56]:
```
5489
```

In [57]:
```
# top 10 genres
df_genre_counts.head(10)
```

Out[57]:

genre	count
classical	635467
classical performance	408792
adult standards	237596
orchestral performance	176051
rock	172423
early music	154337
orchestra	147577
vocal jazz	139082
folk rock	137774
classic rock	136425

In [58]:
```
# 10 random genres
list(df_genre_counts.sample(10).index)
```

Out[58]:
```
['boston electronic',
 'jazz orchestra',
 'deep neo-synthpop',
 'new orleans rap',
 'swing revival',
 'mexican thrash metal',
 'zydeco',
 'viral rap',
 'emo',
 'canadian modern jazz']
```

Too Many Genres!!!

- genre_id seems more like a subgenre than an overarching genre
- it may be worth grouping these subgenres into broader genres

Regex Filtering to Create Subgenre Groups

Test Case 1: Heavy Music (not popular on Billboard Charts)

In [59]:
```
regex_heavy = r'(?:metal|heavy|hard|alternative)'
```

```
heavy_genre_counts = df_popularity[df_popularity.genre.str.contains(regex_heavy).fillna(False)].groupby('genre').count()['id']
```

In [60]:
```
# what about on the Billboard Hot 100?
df_B100_songs_genre[df_B100_songs_genre.genre.str.contains(regex_heavy).fillna(False)].groupby('genre').count()['id'].sort_val
```

Out[60]:
```
genre
alternative metal    131
alternative rock      36
hard rock             34
hardcore hip hop       9
glam metal             5
Name: id, dtype: int64
```

In [61]:
```
# alternative metal is most popular, let's take a look at the top 10 alt metal bands in the Billboard Hot 100
df_B100_songs_genre[df_B100_songs_genre.genre.str.contains('^alternative metal?')].groupby('artist').count()['id'].sort_values
```

Out[61]:
```
artist
Linkin Park          19
Nickelback           18
Three Days Grace      7
Shinedown             7
Good Charlotte        6
Breaking Benjamin     6
Seether               6
Papa Roach            6
Disturbed             6
System Of A Down      5
Name: id, dtype: int64
```

In [62]:
```
# let's compare to the heavy regex category
df_B100_songs_genre[df_B100_songs_genre.genre.str.contains(regex_heavy)].groupby('artist').count()['id'].sort_values(ascending

# same bands, slightly different sorting
```

Out[62]:
```
artist
Linkin Park              19
Nickelback               18
Red Hot Chili Peppers    12
Three Days Grace          7
Shinedown                 7
Breaking Benjamin         6
Disturbed                 6
Seether                   6
Papa Roach                6
Good Charlotte            6
Name: id, dtype: int64
```

In [63]:
```
# even though the top 10 is the same, many less popular bands are missed
(df_B100_songs_genre[df_B100_songs_genre.genre.str.contains('alternative metal')].groupby('artist').count()['id'].shape[0],
df_B100_songs_genre[df_B100_songs_genre.genre.str.contains(regex_heavy)].groupby('artist').count()['id'].shape[0])
```

Out[63]:
```
(34, 97)
```

In [64]:
```
# let's take a look - every heavy song on the Billboard charts that isn't "alternative metal"
df_B100_songs_genre[
    (df_B100_songs_genre.genre.str.contains(regex_heavy)) & (df_B100_songs_genre.genre != 'alternative metal')
].sort_values('release_date')[['artist', 'song', 'release_date', 'genre']]

# not a lot of data to work with, probably shouldn't use this genre
# lets see if this improves specificity for audio features
```

Out[64]:

	artist	song	release_date	genre
6476	Gene Simmons	Radioactive	1978-01-01	hard rock
19110	Vandenberg	Burning Heart	1982-01-01	hard rock
6657	Giuffria	Call To The Heart	1984-01-01	hard rock
6659	Giuffria	Lonely In Love	1984-01-01	hard rock
6658	Giuffria	I Must Be Dreaming	1986-01-01	hard rock

	artist	song	release_date	genre
19188	Vixen	Edge Of A Broken Heart	1988-01-01	glam metal
19187	Vixen	Cryin'	1988-01-01	glam metal
15484	Stryper	Always There For You	1988-01-01	hard rock
15486	Stryper	I Believe In You	1988-01-01	hard rock
3975	Dan Reed Network	Ritual	1988-01-01	glam metal
16430	The Church	Under The Milky Way	1988-02-16	alternative rock
14982	Siouxsie & The Banshees	Peek-A-Boo	1988-09-05	alternative rock
2709	BulletBoys	For The Love Of Money	1988-09-16	hard rock
18721	Tora Tora	Walkin' Shoes	1989-01-01	hard rock
14645	Saraya	Back To The Bullet	1989-01-01	hard rock
14646	Saraya	Love Has Taken Its Toll	1989-01-01	hard rock
15740	Tangier	On The Line	1989-01-01	hard rock
5807	Faster Pussycat	House Of Pain	1989-08-01	hard rock
5546	Enuff Z'Nuff	New Thing	1989-08-18	hard rock
18919	Trixter	Give It To Me Good	1990-01-01	hard rock
10604	Little Caesar	In Your Arms	1990-01-01	hard rock
18921	Trixter	Surrender	1990-01-01	hard rock
18920	Trixter	One In A Million	1990-01-01	hard rock
10603	Little Caesar	Chain Of Fools	1990-01-01	hard rock
19189	Vixen	How Much Love	1990-07-01	glam metal
19190	Vixen	Love Is A Killer	1990-07-01	glam metal
15485	Stryper	Honestly	1991-01-01	hard rock
14981	Siouxsie & The Banshees	Kiss Them For Me	1991-06-10	alternative rock
9428	Kane Roberts	Does Anybody Really Fall In Love Anymore?	1991-07-03	hard rock
13832	Red Hot Chili Peppers	Under The Bridge	1991-09-24	alternative rock
13825	Red Hot Chili Peppers	Give It Away	1991-09-24	alternative rock
14450	Saigon Kick	Love Is On The Way	1992-01-01	hard rock
11875	Mitch Malloy	Nobody Wins In This War	1992-01-01	melodic hard rock
11874	Mitch Malloy	Anything At All	1992-01-01	melodic hard rock
10979	MC Serch	Here It Comes	1992-01-01	hardcore hip hop
17185	The Lemonheads	Into Your Arms	1993-01-01	alternative rock
1421	Belly	Feed The Tree	1993-01-29	alternative rock
7060	Green Jelly	Three Little Pigs	1993-02-25	funk metal
16291	The Breeders	Cannonball	1993-08-30	alternative rock
3901	Da Youngsta's	Hip Hop Ride	1994-01-01	hardcore hip hop
10668	Liz Phair	Supernova	1994-01-01	alternative rock
15239	Sponge	Molly (Sixteen Candles)	1994-08-02	alternative rock
5290	Elastica	Connection	1995-01-01	alternative rock
15203	Spacehog	In The Meantime	1995-01-01	alternative rock
5291	Elastica	Stutter	1995-01-01	alternative rock
3028	Channel Live	Mad Izm	1995-01-01	hardcore hip hop
1485	Better Than Ezra	Good	1995-02-28	alternative rock
1486	Better Than Ezra	Rosealia	1995-02-28	alternative rock

	artist	song	release_date	genre
12444	Nine	Whutcha Want?	1995-03-07	hardcore hip hop
9881	King Just	Warrior's Drum	1995-05-16	hardcore hip hop
14771	Seven Mary Three	Cumbersome	1995-09-05	alternative rock
17653	The Rentals	Friends Of P.	1995-10-20	alternative pop
7258	Heather B.	Do You	1996-01-01	hardcore hip hop
14433	Sadat X	Hang 'Em High	1996-03-01	alternative hip hop
1484	Better Than Ezra	Desperately Wanting	1996-08-02	alternative rock
11998	Mr. Big	To Be With You	1996-11-25	hard rock
11999	Mr. Big	Wild World	1996-11-25	hard rock
11996	Mr. Big	Just Take My Heart	1996-11-25	hard rock
5697	Eve 6	Inside Out	1998-01-01	alternative rock
1483	Better Than Ezra	At The Stars	1998-08-14	alternative rock
13822	Red Hot Chili Peppers	Californication	1999-06-08	alternative rock
15062	Smash Mouth	Then The Morning Comes	1999-06-08	alternative rock
13826	Red Hot Chili Peppers	Otherside	1999-06-08	alternative rock
13827	Red Hot Chili Peppers	Scar Tissue	1999-06-08	alternative rock
15061	Smash Mouth	All Star	1999-06-08	alternative rock
5696	Eve 6	Here's To The Night	2000-01-01	alternative rock
6998	Gorillaz	Clint Eastwood	2001-01-01	alternative hip hop
13831	Red Hot Chili Peppers	The Zephyr Song	2002-07-09	alternative rock
13821	Red Hot Chili Peppers	By The Way	2002-07-09	alternative rock
10669	Liz Phair	Why Can't I?	2003-01-01	alternative rock
6058	Fountains Of Wayne	Stacy's Mom	2003-01-01	alternative rock
15952	Terror Squad	Lean Back	2004-01-01	hardcore hip hop
15953	Terror Squad	Take Me Home	2004-01-01	hardcore hip hop
12303	Nelson	Only Time Will Tell	2004-01-01	hard rock
13866	Rhinoceros	Apricot Brandy	2005-02-08	buffalo ny metal
17277	The Mars Volta	The Widow	2005-02-10	alternative rock
6999	Gorillaz	Feel Good Inc	2005-05-23	alternative hip hop
7136	HIM	Wings Of A Butterfly	2005-09-26	gothic metal
13829	Red Hot Chili Peppers	Tell Me Baby	2006-05-09	alternative rock
13828	Red Hot Chili Peppers	Snow ((Hey Oh))	2006-05-09	alternative rock
13823	Red Hot Chili Peppers	Dani California	2006-05-09	alternative rock
4865	DragonForce	Through The Fire And Flames	2006-05-23	metal
5545	Enuff Z'Nuff	Fly High Michelle	2006-09-26	hard rock
17829	The Shins	Phantom Limb	2007-01-23	alternative rock
11997	Mr. Big	Romeo	2007-07-17	hard rock
2506	Britny Fox	Long Way To Love	2007-08-07	hard rock
2710	BulletBoys	Smooth Up	2008-02-05	hard rock
14906	Shiny Toy Guns	Major Tom	2009-01-01	alternative dance
12302	Nelson	More Than Ever	2010-12-03	hard rock
12301	Nelson	After The Rain	2010-12-03	hard rock
479	Alex Clare	Too Close	2011-01-01	modern alternative rock

	artist	song	release_date	genre
13830	Red Hot Chili Peppers	The Adventures Of Rain Dance Maggie	2011-08-29	alternative rock
12757	PMD	I Saw It Cummin'	2013-02-05	hardcore hip hop
13003	Paul Stanley	Hold Me, Touch Me	2014-01-01	hard rock
15060	Smart E's	Sesame's Treet	2014-03-02	hardcore techno
7483	I Prevail	Blank Space	2014-12-16	nu-metalcore
9360	KALEO	Way Down We Go	2016-06-10	modern alternative rock
13824	Red Hot Chili Peppers	Dark Necessities	2016-06-17	alternative rock
5726	FRENTE!	Bizarre Love Triangle	2017-12-21	australian alternative rock
1182	Bad Wolves	Zombie	2018-01-19	metal
5292	Electric Boys	All Lips N' Hips	2019-04-12	hard rock
1414	Bella Poarch	Build A Bitch	2021-05-14	modern alternative pop
12556	Oliver Tree	Life Goes On	2021-05-28	alternative hip hop

Boxplots

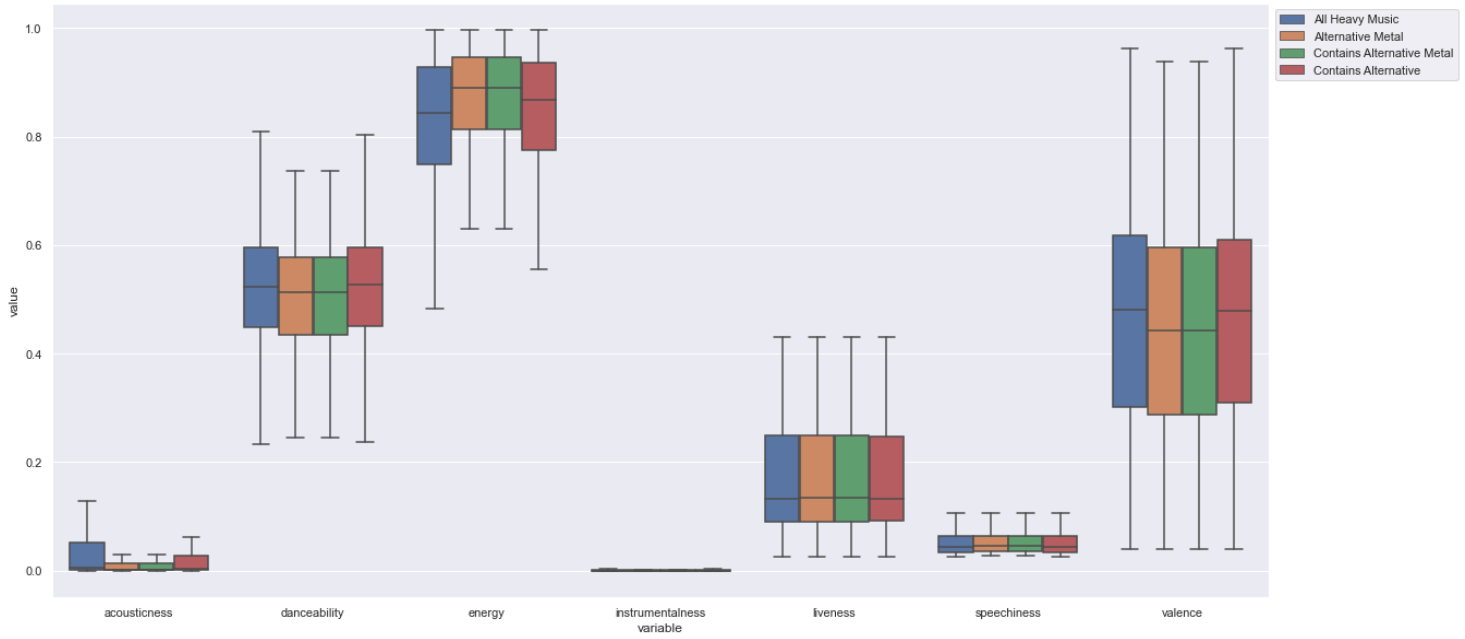
In [65]:

```
# helper function with variable number of dataframes would be useful to compare more data

def boxplot_genre_comparison(dataframe, genre_dict, title, figsize=(20,10), showfliers=False):

    boxplot_features = ['acousticness', 'danceability', 'energy', 'instrumentalness', 'liveness', 'speechiness', 'valence']
    plt.figure(figsize=figsize)

    list_of_genre_df = []

    for genre in genre_dict:
        list_of_genre_df.append(dataframe[dataframe.genre.str.contains(genre_dict[genre]).fillna(False)][boxplot_features].ass

    sns.boxplot(pd.melt(pd.concat(list_of_genre_df), id_vars=['genre_name']), x='variable', y='value', hue='genre_name', showf
#     plt.title(title, fontsize=13, pad=10)
    plt.legend(loc='upper left', bbox_to_anchor=(1, 1))

    # save the figure
    plt.savefig(f'figures/genres/{title}.png', facecolor='w', dpi=150, bbox_inches='tight')

    # print the title
    print(title)

    plt.show()
```

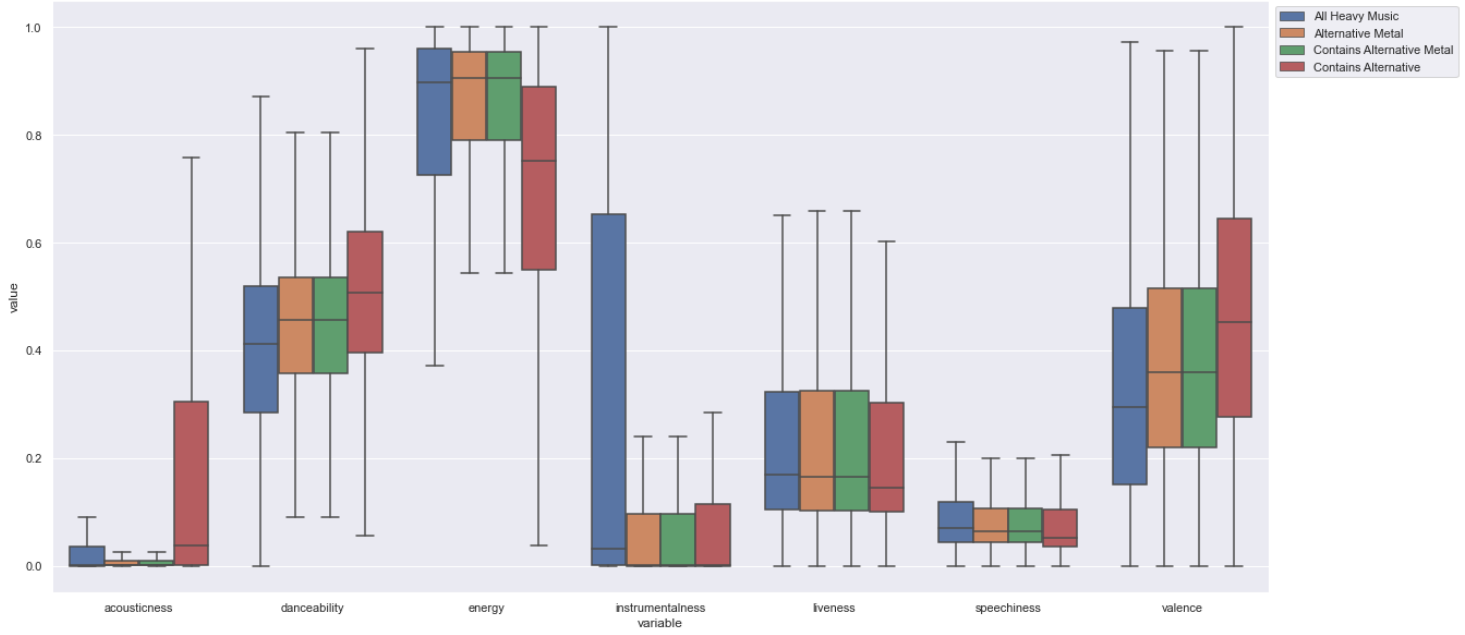
In [66]:

```
genre_dict = {
    'All Heavy Music': r'(?:metal|heavy|hard|alternative)',
    'Alternative Metal': '^alternative metal?',
    'Contains Alternative Metal': 'alternative metal',
    'Contains Alternative': 'alternative',
}

title = 'Comparison of Genrealised Genre with Top Subgenre - Heavy Music ({})'

boxplot_genre_comparison(df_B100_songs_genre, genre_dict, title.format('Billboard Charts'))
boxplot_genre_comparison(df_10M, genre_dict, title.format('All Songs'))
```

```
Comparison of Genrealised Genre with Top Subgenre - Heavy Music (Billboard Charts)
```



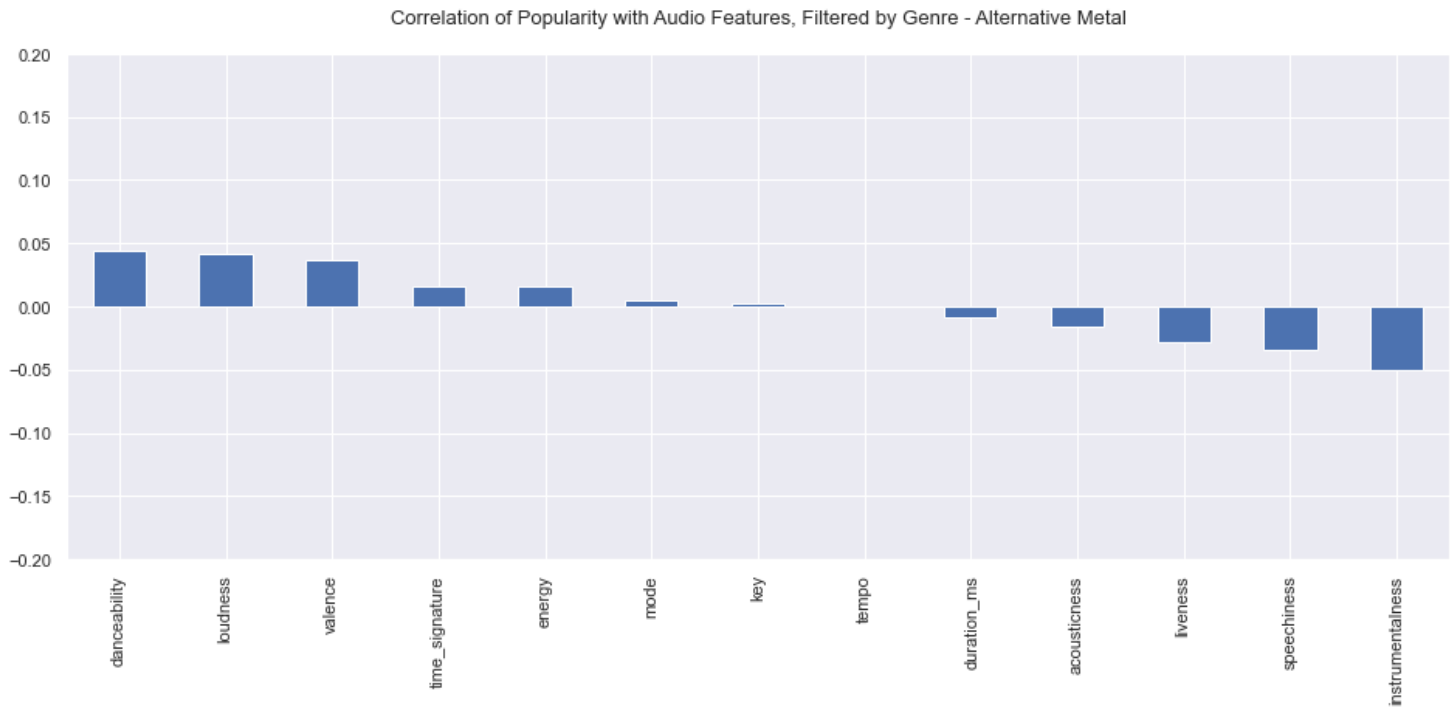
Comparison of Genrealised Genre with Top Subgenre - Heavy Music (All Songs)



Conclusion: It looks like adding more genrealised genre data increases variability

In [67]:

```
# let's see if "alternative metal is any better behaved than heavy music in general"

genre = 'alternative metal'

df_popularity.query(f'genre == "{genre}"').corr()['POPULAR'].sort_values(ascending=False)[1:].plot(kind='bar', figsize=(16,6))
title = f'Correlation of Popularity with Audio Features, Filtered by Genre - {genre.title()}'
plt.title(title, fontsize=13, pad=20)
plt.ylim(*ylim)

plt.show()
```



- This looks a little better, not as strong as some genres, and weak correlation overall
- It may be difficult to make any popularity predictions using these features for this genre
- Interesting non-intuitive results. Popular Alternative Metal is:
  - Billboard charts have more consistent songs (not surprising)
  - danceable
  - loud (not surprising)
  - happier than typically in the genre (positively correlated with higher valence)
  - lower instrumentalness correlated with higher "popularity"

Test Case 2: Pop Music

In [68]:
```
regex_pop = r'(?:pop|adult standards|contemporary)'

# top 10 regex_pop genres (all songs)
df_popularity[df_popularity.genre.str.contains(regex_pop).fillna(False)].groupby('genre').count()['id'].sort_values(ascending=
```

Out[68]:
```
genre
adult standards      166612
dance pop             29033
afropop               15853
c-pop                 15568
pop rock              15377
pop rap               14520
italian adult pop     14267
europop               13013
new wave pop          11763
russian pop           11404
Name: id, dtype: int64
```

In [69]:
```
# top 10 regex_pop genres (billboard)
df_B100_songs_genre[df_B100_songs_genre.genre.str.contains(regex_pop).fillna(False)].groupby('genre').count()['id'].sort_value
```

Out[69]:
```
genre
adult standards       2640
dance pop             1249
brill building pop     561
pop                    491
pop rock               327
new wave pop           273
bubblegum pop          241
post-teen pop          196
pop rap                178
urban contemporary     132
Name: id, dtype: int64
```

In [70]:

```
# confirming that "adult standards" are indeed Pop Music - looks like it to me!
df_B100_songs_genre.query('genre == "adult standards"').artist.sample(10)
```

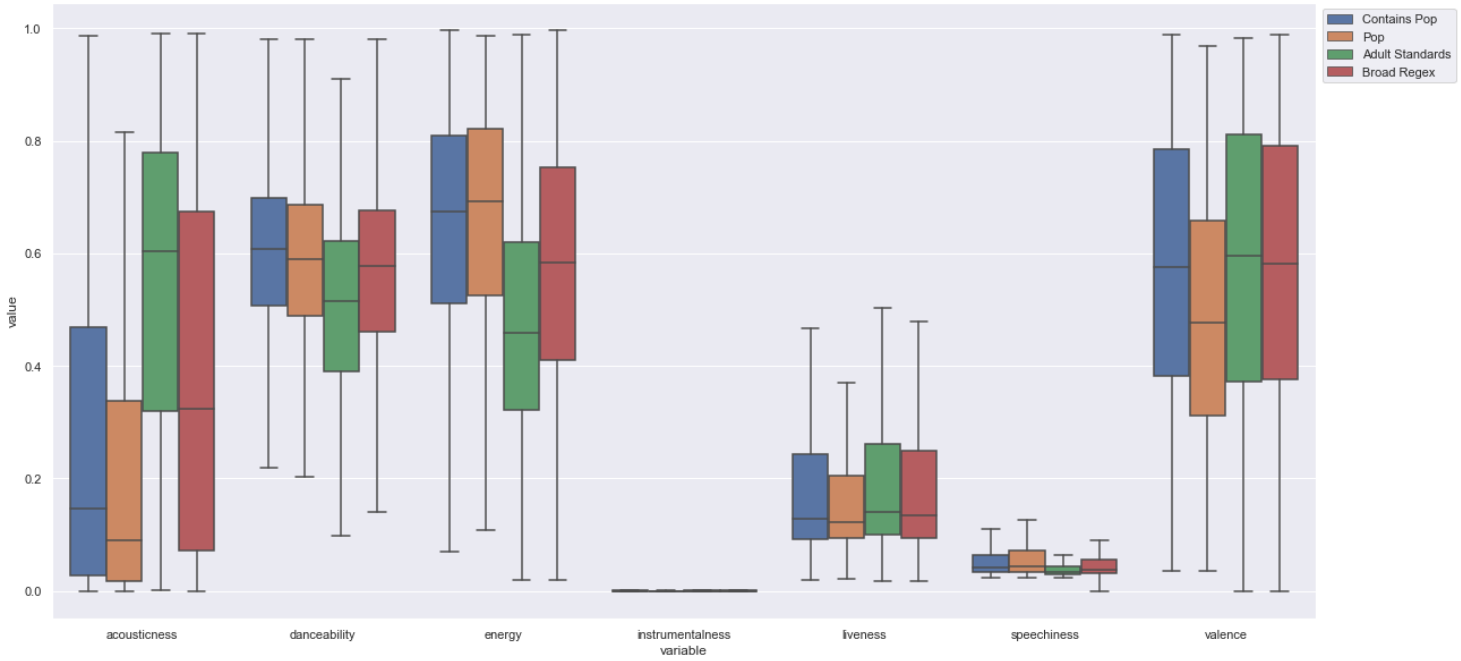
Out[70]:

```
2410              Brenda Lee
8132     Jay & The Americans
3152                 Chicago
10689                   Lobo
4845                Dr. Hook
6178           Frankie Avalon
16510           The Crystals
19276           Wayne Newton
14644          Sarah Vaughan
8455           Jimmy Clanton
Name: artist, dtype: object
```

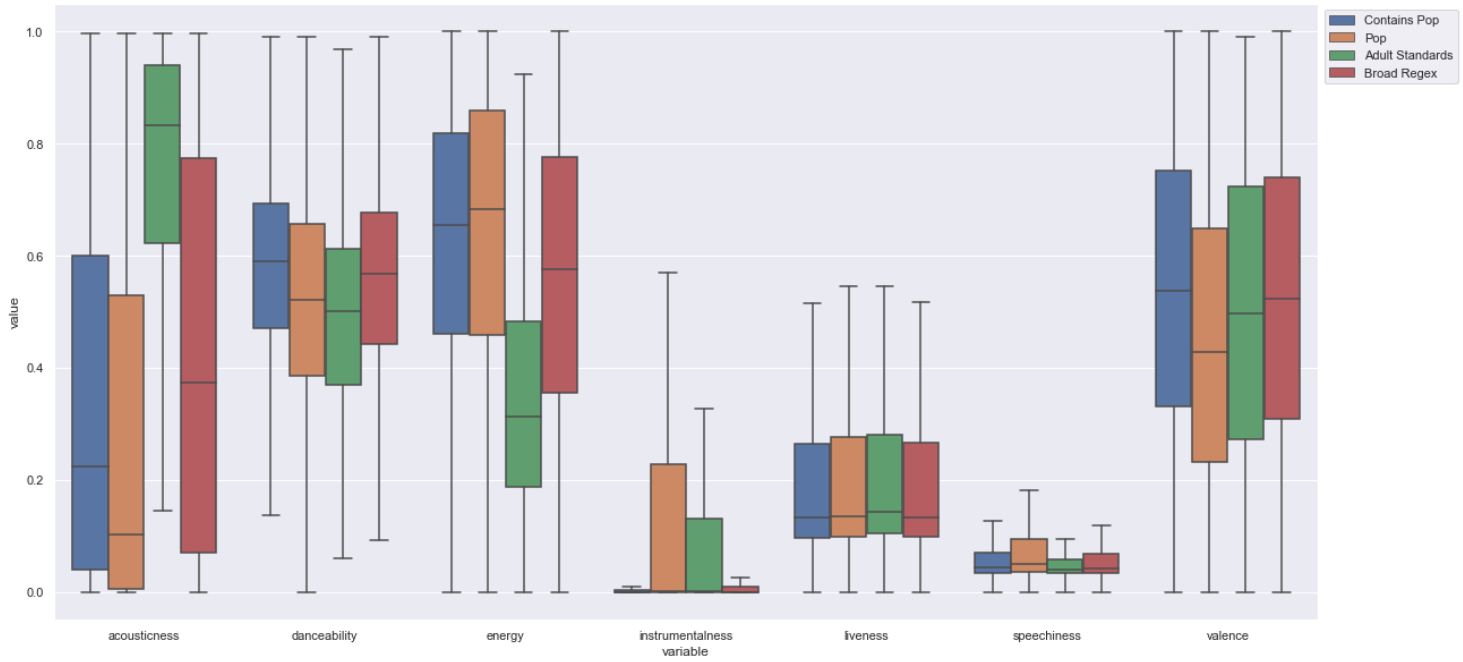
In [71]:

```
# boxplot comparison of Pop genres

genre_dict = {
    'Contains Pop': r'pop',
    'Pop': r'^pop?',
    'Adult Standards': r'^adult standards?',
    'Broad Regex':  r'(?:pop|adult standards|contemporary)'
}

title = 'Comparison of Pop Genre Groupings - {}'

boxplot_genre_comparison(df_B100_songs_genre, genre_dict, title.format('Billboard Charts'))
boxplot_genre_comparison(df_10M, genre_dict, title.format('All Songs'))
```

Comparison of Pop Genre Groupings - Billboard Charts



Comparison of Pop Genre Groupings - All Songs



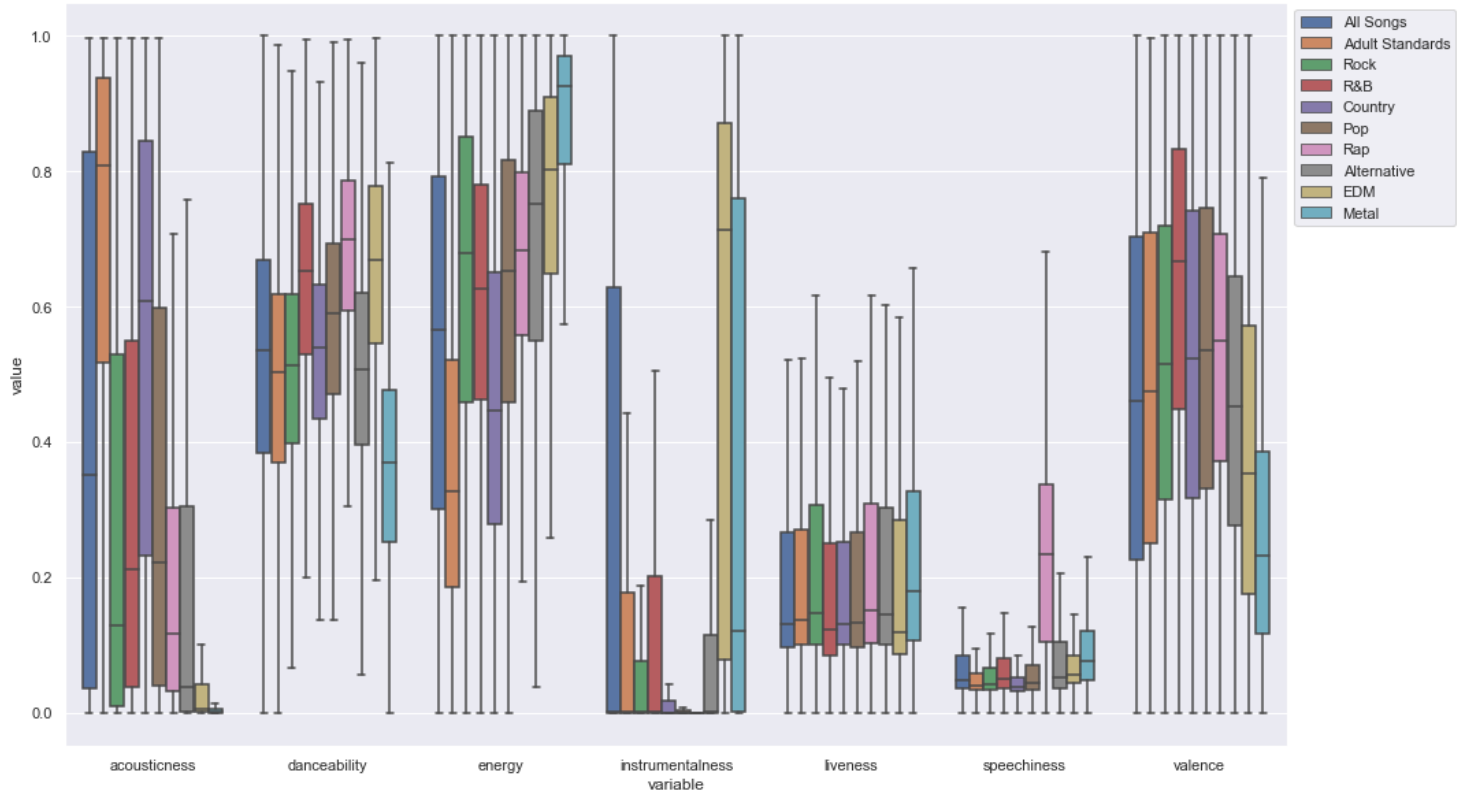
Boxplots for Large Genre Groups

In [72]:
```
# df_B100_songs_genre.groupby('genre').count()['id'].sort_values(ascending=False).head(50)
```

In [73]:
```
# approximating genre categories
genre_dict = {
    'All Songs': r'.+',
    'Adult Standards': r'(?:adult standards|mellow gold|soft rock|contemporary)',
    'Rock': r'(?:rock|^country|^soft)',
    'R&B':  r'(?:soul|rhythm and blues|motown|funk|disco|quiet storm)',  # this one is a stretch
    'Country': r'(?:country|folk|nashville sound)',
    'Pop': r'\b(?:pop|europop)\b',  # whole word only
    'Rap': r'\b(?:rap|trap|hip hop)\b',  # whole word only
    'Alternative': r'(?:alternative)',  # not sure if grunge or punk should be here
    'EDM': r'(?:house|trance|techno|drum and bass|dubstep|synth|edm|breakbeat)', # not very popular on Bilboard
    'Metal': r'(?:heavy|metal|thrash|djent|deathcore|mathcore|grindcore)' # not very popular on Bilboard
}
```

In [74]:
```
# plot it
title = 'Comparison of Audio Featuers for Large Genre Groupings (as defined by Kevin) - {}'
figsize = (16, 10)
# boxplot_genre_comparison(df_B100_songs_genre, genre_dict, title.format('Billboard Charts'), figsize=figsize)
boxplot_genre_comparison(df_10M, genre_dict, title.format('All Songs'), figsize=figsize)
```

```
Comparison of Audio Featuers for Large Genre Groupings (as defined by Kevin) - All Songs
```



In [75]:

```
# genre statistics vs relative popularity

df_genres = pd.DataFrame()

for genre in list(genre_dict.keys()):
    df_genres.loc[genre, 'billboard_counts'] = df_B100_songs_genre[(df_B100_songs_genre.genre.str.contains(genre_dict[genre]))
    df_genres.loc[genre, 'all_counts'] = df_10M[(df_10M.genre.str.contains(genre_dict[genre])).fillna(False)].count()['id']
```

In [76]:

```
# new columns
df_genres['proportion_billboard'] = df_genres['billboard_counts'] / df_genres['billboard_counts'].sum()
df_genres['proportion_allsongs'] = df_genres['all_counts'] / df_genres['all_counts'].sum()
df_genres['relative_popularity'] = df_genres['proportion_billboard'] / df_genres['proportion_allsongs']
df_genres['songs_per_popular_song'] = df_genres['all_counts'] / df_genres['billboard_counts']

# transpose and format
dft = df_genres.T
dft.iloc[0:2] = dft.iloc[0:2].applymap('{0:,.0f}'.format)
dft.iloc[2:-1] = dft.iloc[2:-1].applymap('{0:,.3f}'.format)
dft.iloc[-1:] = dft.iloc[-1:].applymap('{0:,.0f}'.format)
dft
```

Out[76]:

	All Songs	Adult Standards	Rock	R&B	Country	Pop	Rap	Alternative	EDM	Metal
billboard_counts	19,750	3,680	6,321	2,981	2,449	3,908	2,087	177	104	142
all_counts	6,320,843	214,324	675,340	139,178	275,228	548,402	428,865	87,200	259,091	256,785
proportion_billboard	0.475	0.088	0.152	0.072	0.059	0.094	0.050	0.004	0.003	0.003
proportion_allsongs	0.687	0.023	0.073	0.015	0.030	0.060	0.047	0.009	0.028	0.028
relative_popularity	0.691	3.800	2.071	4.740	1.969	1.577	1.077	0.449	0.089	0.122
songs_per_popular_song	320	58	107	47	112	140	205	493	2,491	1,808

In [77]:

```
# best genres to write music in if you want onto the Billboad Hot 100
print('Total Songs in each Genre per Song on the Billboard Hot 100')
print(df_genres['songs_per_popular_song'].astype('int32').sort_values())
```

```
Total Songs in each Genre per Song on the Billboard Hot 100
R&B                   46
Adult Standards       58
```

```
Rock                106
Country             112
Pop                 140
Rap                 205
All Songs           320
Alternative         492
Metal              1808
EDM                2491
Name: songs_per_popular_song, dtype: int32
```

In [ ]:

Attachment 3

Summary of Findings

- exclude tracks based on non-musical genres:
  - e.g., 'sleep', 'ringtone', etc.
- exclude tracks based on domain knowledge:
  - songs <1min or >10min are extreme outliers (less likely to be songs than misc audio)
    - duration_ms < 60_000
    - duration_ms > 600_000
- exclude select extrema for audio features based on inspection:
  - 99% range (from 0.5% to 99.5% was used)
  - some outlier regions appear to contain songs, including hit songs
  - some outlier regions include mostly non-music tracks
    - valence == 0
    - speechiness > 0.947000
    - tempo == 0
    - loudness < -34.668999
    - danceability < 0.064400
- other outlier exclusion methods didn't work effectively
  - IQR and Z values excluded very little
  - LocalOutlierFactor() from sklearn not effective by default and very slow

Imports

In [1]:
```
import spotipy
import re

# math and dataframes
import pandas as pd
import numpy as np
from scipy.stats import zscore

# plotting
import matplotlib.pyplot as plt
import matplotlib as mpl
import time
import seaborn as sns
sns.set_theme()

# jupyter notebook full-width display
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

# pandas formatting
pd.set_option('display.float_format', '{:.3f}'.format)
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 200)
```

In [2]:
```
# all songs with audio features (combined from 3 sources)
df_10M = pd.read_pickle('df_10M.pickle')

# all Billboard 100 lists, audio features included where possible
df_B100 = pd.read_pickle('df_B100.pickle')

# all unique songs from the Billboard 100 lists, audio features included where possible
df_B100_songs = pd.read_pickle('df_B100_songs.pickle')
```

In [3]:
```
# confirm import worked (B100 songs are all in df_10M, no duplicates)
id_B100 = set(df_B100_songs[~df_B100_songs.id.isnull()].id)
len(id_B100), df_10M[df_10M.id.isin(id_B100)].shape[0]
# GOOD!
```

Out[3]: (21274, 21274)

OUTLIERS

METHOD 1: IQR

(didn't work)

In [4]:
```
df_B100_range = df_B100_songs.describe().T[['min', 'max']]
df_outliers = df_10M.describe()['min':'max'].T
```

In [5]:
```
df_B100_range
```

Out[5]:

	min	max
acousticness	0.000	0.996
danceability	0.000	0.988
duration_ms	12507.000	2902827.000
energy	0.000	0.999
instrumentalness	0.000	0.996
key	0.000	11.000
liveness	0.000	1.000
loudness	-60.000	2.291
mode	0.000	1.000
speechiness	0.000	0.949
tempo	0.000	241.009
time_signature	0.000	5.000
valence	0.000	0.989

In [6]:
```
df_outliers
```

Out[6]:

	min	25%	50%	75%	max
acousticness	0.000	0.034	0.337	0.817	0.996
danceability	0.000	0.396	0.545	0.676	1.000
duration_ms	1000.000	169507.000	216851.000	275053.000	6072187.000
energy	0.000	0.311	0.567	0.789	1.000
instrumentalness	0.000	0.000	0.002	0.647	1.000
key	0.000	2.000	5.000	8.000	11.000
liveness	0.000	0.096	0.129	0.262	1.000
loudness	-60.000	-13.674	-9.195	-6.397	7.234
mode	0.000	0.000	1.000	1.000	1.000
speechiness	0.000	0.036	0.047	0.082	0.974
tempo	0.000	95.077	118.959	137.453	249.987
time_signature	0.000	4.000	4.000	4.000	5.000
valence	0.000	0.234	0.468	0.708	1.000

In [7]:
```
df_outliers['IQR'] = df_outliers['75%'] - df_outliers['25%']
df_outliers['out_low'] = df_outliers['25%'] - 1.5 * df_outliers['IQR']
df_outliers['out_high'] = df_outliers['75%'] + 1.5 * df_outliers['IQR']
df_outliers
```

Out[7]:

	min	25%	50%	75%	max	IQR	out_low	out_high
acousticness	0.000	0.034	0.337	0.817	0.996	0.783	-1.142	1.992
danceability	0.000	0.396	0.545	0.676	1.000	0.280	-0.024	1.096
duration_ms	1000.000	169507.000	216851.000	275053.000	6072187.000	105546.000	11188.000	433372.000
energy	0.000	0.311	0.567	0.789	1.000	0.478	-0.406	1.506
instrumentalness	0.000	0.000	0.002	0.647	1.000	0.647	-0.970	1.617
key	0.000	2.000	5.000	8.000	11.000	6.000	-7.000	17.000
liveness	0.000	0.096	0.129	0.262	1.000	0.166	-0.152	0.510
loudness	-60.000	-13.674	-9.195	-6.397	7.234	7.277	-24.589	4.519
mode	0.000	0.000	1.000	1.000	1.000	1.000	-1.500	2.500
speechiness	0.000	0.036	0.047	0.082	0.974	0.047	-0.034	0.152
tempo	0.000	95.077	118.959	137.453	249.987	42.376	31.513	201.017
time_signature	0.000	4.000	4.000	4.000	5.000	0.000	4.000	4.000
valence	0.000	0.234	0.468	0.708	1.000	0.474	-0.477	1.419

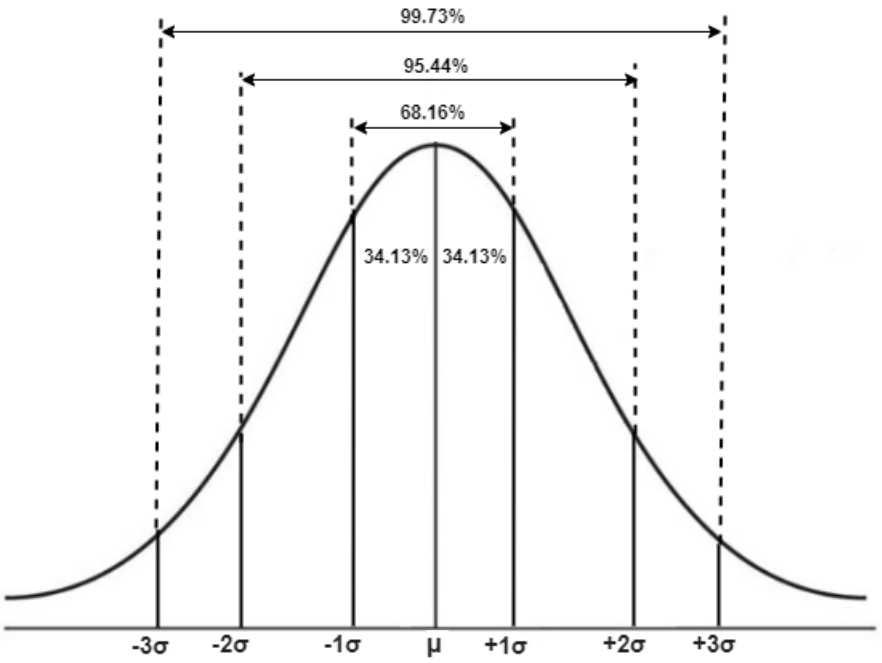
In [8]:
```
# this didn't exclude anything except liveliness > 0.51, speechiness > 0.152 and duration_ms
# let's check duration, to see if it's useful
df_outliers.loc['duration_ms', 'out_low'] / (60 * 1000), df_outliers.loc['duration_ms', 'out_high'] / (60 * 1000)

# 12 second songs, nope
# 7 minute songs, maybe
```

Out[8]:
```
(0.18646666666666667, 7.2228666666666665)
```

METHOD 2: Z-scores

(didn't work)

ESTIMATE (not exactly correct unless Gaussian):



In [9]:
```
# helper function, check z scores
def check_z_score(z_exclude=3):
    z_scores = {1: 0.6816, 2: 0.9544, 3: 0.9973}
    z_exclude = z_exclude
    low_quartile = round((1 - z_scores[z_exclude]) / 2, 5)
```

```
    high_quartile = round(1 - (1 - z_scores[z_exclude]) / 2, 5)

    quartiles = [low_quartile, high_quartile]

    df_B100_range = df_B100_songs.describe().T
    df_10M_range = df_10M.describe(quartiles)['min':'max'].T

    output_df = pd.concat(
        [df_B100_range.rename({'min': 'B100_min', 'max': 'B100_max'}, axis=1), df_10M_range],
        axis=1)

    output_df['low_error'] = output_df['B100_min'] - output_df[df_10M_range.columns[1]]
    output_df['high_error'] = output_df[df_10M_range.columns[3]] - output_df['B100_max']

    output_df.loc[['duration_ms', ]] = output_df.loc[['duration_ms', ]] / 60_000  # convert ms to minutes
    output_df = output_df.rename({'duration_ms': 'minutes'})

    return output_df[['B100_min', df_10M_range.columns[1], 'low_error', 'B100_max', df_10M_range.columns[3], 'high_error']]
```

In [10]:

```
# z-score > 3
z3 = check_z_score(3)

# z-score > 2
z2 = check_z_score(2)
```

In [11]:

```
z3
```

Out[11]:

	B100_min	0.1%	low_error	B100_max	99.9%	high_error
acousticness	0.000	0.000	-0.000	0.996	0.996	0.000
danceability	0.000	0.000	0.000	0.988	0.950	-0.038
minutes	0.208	0.137	0.072	48.380	25.059	-23.321
energy	0.000	0.001	-0.001	0.999	0.999	0.000
instrumentalness	0.000	0.000	0.000	0.996	0.989	-0.007
key	0.000	0.000	0.000	11.000	11.000	0.000
liveness	0.000	0.000	0.000	1.000	0.981	-0.019
loudness	-60.000	-40.092	-19.908	2.291	-0.811	-3.102
mode	0.000	0.000	0.000	1.000	1.000	0.000
speechiness	0.000	0.000	0.000	0.949	0.960	0.011
tempo	0.000	0.000	0.000	241.009	207.734	-33.275
time_signature	0.000	0.000	0.000	5.000	5.000	0.000
valence	0.000	0.000	0.000	0.989	0.981	-0.008

In [12]:

```
z2
```

Out[12]:

	B100_min	2.3%	low_error	B100_max	97.7%	high_error
acousticness	0.000	0.000	-0.000	0.996	0.993	-0.003
danceability	0.000	0.139	-0.139	0.988	0.859	-0.129
minutes	0.208	0.931	-0.723	48.380	8.884	-39.496
energy	0.000	0.027	-0.027	0.999	0.978	-0.021
instrumentalness	0.000	0.000	0.000	0.996	0.948	-0.048
key	0.000	0.000	0.000	11.000	11.000	0.000
liveness	0.000	0.048	-0.048	1.000	0.803	-0.197
loudness	-60.000	-27.790	-32.210	2.291	-3.046	-5.337
mode	0.000	0.000	0.000	1.000	1.000	0.000
speechiness	0.000	0.027	-0.027	0.949	0.508	-0.441

	B100_min	2.3%	low_error	B100_max	97.7%	high_error
tempo	0.000	67.580	-67.580	241.009	182.926	-58.083
time_signature	0.000	3.000	-3.000	5.000	5.000	0.000
valence	0.000	0.036	-0.036	0.989	0.962	-0.027

In [13]:
```
Z_errors = pd.concat([
    z3[[z3.columns[2], z3.columns[5]]].rename({z3.columns[2]: 'Z_error_low_3', z3.columns[5]: 'Z_error_high_3'}, axis=1),
    z2[[z2.columns[2], z2.columns[5]]].rename({z2.columns[2]: 'Z_error_low_2', z2.columns[5]: 'Z_error_high_2'}, axis=1)], axi
Z_errors
```

Out[13]:

	Z_error_low_3	Z_error_high_3	Z_error_low_2	Z_error_high_2
acousticness	-0.000	0.000	-0.000	-0.003
danceability	0.000	-0.038	-0.139	-0.129
minutes	0.072	-23.321	-0.723	-39.496
energy	-0.001	0.000	-0.027	-0.021
instrumentalness	0.000	-0.007	0.000	-0.048
key	0.000	0.000	0.000	0.000
liveness	0.000	-0.019	-0.048	-0.197
loudness	-19.908	-3.102	-32.210	-5.337
mode	0.000	0.000	0.000	0.000
speechiness	0.000	0.011	-0.027	-0.441
tempo	0.000	-33.275	-67.580	-58.083
time_signature	0.000	0.000	-3.000	0.000
valence	0.000	-0.008	-0.036	-0.027

In [14]:
```
df_sqerr = pd.DataFrame()
df_sqerr['Z3'] = Z_errors['Z_error_low_3'] * Z_errors['Z_error_low_3'] + Z_errors['Z_error_high_3'] * Z_errors['Z_error_high_3
df_sqerr['Z2'] = Z_errors['Z_error_low_2'] * Z_errors['Z_error_low_2'] + Z_errors['Z_error_high_2'] * Z_errors['Z_error_high_2
df_sqerr['Z3_is_best'] = df_sqerr['Z2'] > df_sqerr['Z3']
df_sqerr

# therefore Zscore of 3 is best for outlier removal
```

Out[14]:

	Z3	Z2	Z3_is_best
acousticness	0.000	0.000	True
danceability	0.001	0.036	True
minutes	543.894	1560.456	True
energy	0.000	0.001	True
instrumentalness	0.000	0.002	True
key	0.000	0.000	False
liveness	0.000	0.041	True
loudness	405.951	1065.968	True
mode	0.000	0.000	False
speechiness	0.000	0.195	True
tempo	1107.226	7940.692	True
time_signature	0.000	9.000	True
valence	0.000	0.002	True

In [15]:
```
z3[[z3.columns[1], z3.columns[4]]]
```

Out[15]:

	0.1%	99.9%
acousticness	0.000	0.996
danceability	0.000	0.950
minutes	0.137	25.059
energy	0.001	0.999
instrumentalness	0.000	0.989
key	0.000	11.000
liveness	0.000	0.981
loudness	-40.092	-0.811
mode	0.000	1.000
speechiness	0.000	0.960
tempo	0.000	207.734
time_signature	0.000	5.000
valence	0.000	0.981

In [16]:
```
df_B100_songs.describe().T[['min', 'max']]
```

Out[16]:

	min	max
acousticness	0.000	0.996
danceability	0.000	0.988
duration_ms	12507.000	2902827.000
energy	0.000	0.999
instrumentalness	0.000	0.996
key	0.000	11.000
liveness	0.000	1.000
loudness	-60.000	2.291
mode	0.000	1.000
speechiness	0.000	0.949
tempo	0.000	241.009
time_signature	0.000	5.000
valence	0.000	0.989

In [17]:
```
# longest and shortest songs on B100
df_B100_songs.describe().T[['min', 'max']].rename({'duration_ms': 'minutes'}, axis=1).loc[['duration_ms']] / 60_000
```

Out[17]:

	min	max
duration_ms	0.208	48.380

In [18]:
```
# yep, actually 48 minutes, a live version, correct song, but probalby an outlier
df_B100_songs[df_B100_songs.duration_ms == df_B100_songs.duration_ms.max()]
```

Out[18]:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness
18120	0mEyKJpqUP6lnNvySIUhsF	Electric Barbarella	Duran Duran	rock	2017-05-03	0.004	0.364	2902827	0.853	0.001	2	0.68

In [20]:
```
# yep, not the normal version of the song
# also, API doesn't use '- callout' in the name so can get a false positive
# outlier
df_B100_songs[df_B100_songs.duration_ms == df_B100_songs.duration_ms.min()]
```

Out[20]:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	l
15906	4EIMnC6rzLlYvvIOq1tHqS	In Love With A Girl	Gavin DeGraw	pop rock	2007-10-25	0.058	0.000	12507	0.798	0.000	3	0.235	

check actual zscores for duration

In [48]:
```
# not an outlier, (no songs are short enough to be considered outliers)
zscore(df_B100_songs.duration_ms.to_list())[15906]
```

Out[48]:
```
-2.9370678974825735
```

METHOD 3: Percentile

works pretty good, but may exclude some songs

In [59]:
```
# intervals:  99.9%, 99%, and 95%
percentile = [0.0005, 0.005, 0.025, 0.975, 0.995, 0.9995]
TEMP = df_B100_songs.copy()
TEMP['minutes'] = TEMP['duration_ms'].apply(lambda x: x / 60_000)
TEMP.drop('duration_ms', axis=1).describe(percentile)['min':'max'].T
```

Out[59]:

	min	0.05%	0.5%	2.5%	50%	97.5%	99.5%	99.95%	max
acousticness	0.000	0.000	0.000	0.001	0.204	0.898	0.961	0.991	0.996
danceability	0.000	0.113	0.192	0.273	0.605	0.881	0.935	0.968	0.988
energy	0.000	0.024	0.107	0.204	0.630	0.941	0.975	0.994	0.999
instrumentalness	0.000	0.000	0.000	0.000	0.000	0.656	0.905	0.961	0.996
key	0.000	0.000	0.000	0.000	5.000	11.000	11.000	11.000	11.000
liveness	0.000	0.017	0.029	0.043	0.134	0.784	0.970	0.991	1.000
loudness	-60.000	-27.614	-20.190	-16.650	-8.369	-3.232	-2.278	-0.727	2.291
mode	0.000	0.000	0.000	0.000	1.000	1.000	1.000	1.000	1.000
speechiness	0.000	0.022	0.024	0.026	0.042	0.350	0.485	0.900	0.949
tempo	0.000	50.764	65.949	75.072	119.230	182.752	202.655	209.992	241.009
time_signature	0.000	1.000	3.000	3.000	4.000	4.000	5.000	5.000	5.000
valence	0.000	0.035	0.071	0.141	0.621	0.964	0.973	0.983	0.989
minutes	0.208	0.562	1.738	2.067	3.550	6.321	8.372	11.883	48.380

In [62]:
```
# 99% seems like the right choice for duration (based on my expertise)
TEMP.drop('duration_ms', axis=1).describe(percentile)['min':'max'][['minutes']]
```

Out[62]:

	minutes
min	0.208
0.05%	0.562
0.5%	1.738
2.5%	2.067
50%	3.550

	minutes
97.5%	6.321
99.5%	8.372
99.95%	11.883
max	48.380

In [77]:
```
df_outlier_limits_99 = (
    df_B100_songs
    .describe([0.005, 0.995])['0.5%':'99.5%':2].T
    .drop(['key', 'mode', 'time_signature'])
)
df_outlier_limits_99
```

Out[77]:

	0.5%	99.5%
acousticness	0.000	0.961
danceability	0.192	0.935
duration_ms	104272.530	502313.490
energy	0.107	0.975
instrumentalness	0.000	0.905
liveness	0.029	0.970
loudness	-20.190	-2.278
speechiness	0.024	0.485
tempo	65.949	202.655
valence	0.071	0.973

In [79]:
```
df_outlier_limits_99.columns = ['lower_limit', 'upper_limit']
df_outlier_limits_99
```

Out[79]:

	lower_limit	upper_limit
acousticness	0.000	0.961
danceability	0.192	0.935
duration_ms	104272.530	502313.490
energy	0.107	0.975
instrumentalness	0.000	0.905
liveness	0.029	0.970
loudness	-20.190	-2.278
speechiness	0.024	0.485
tempo	65.949	202.655
valence	0.071	0.973

In [80]:
```
# these could be good
df_outlier_limits_99.to_pickle('df_outlier_limits_99.pickle')
```

In [81]:
```
# let's compart to df_10M
# there are WAY more extreme songs in df_10M
(
    df_10M
    .describe([0.005, 0.995])['0.5%':'99.5%':2].T
    .drop(['key', 'mode', 'time_signature'])
)
```

Out[81]:

	0.5%	99.5%
acousticness	0.000	0.995
danceability	0.065	0.918
duration_ms	22107.000	868733.000
energy	0.005	0.995
instrumentalness	0.000	0.975
liveness	0.030	0.962
loudness	-34.602	-1.826
speechiness	0.024	0.947
tempo	48.654	201.512
valence	0.000	0.973

In [83]:

```
# maybe 95% would be better
(
    df_10M
    .describe([0.025, 0.975])['2.5%':'97.5%':2].T
    .drop(['key', 'mode', 'time_signature'])
)
```

Out[83]:

	2.5%	97.5%
acousticness	0.000	0.992
danceability	0.144	0.855
duration_ms	59000.000	520160.000
energy	0.030	0.977
instrumentalness	0.000	0.946
liveness	0.049	0.780
loudness	-27.361	-3.129
speechiness	0.027	0.479
tempo	68.292	181.613
valence	0.036	0.962

In [ ]:

```
# not sure how to deal with information leakage
# if outlier limits are determined from B100, they are used to predict themselves
# may need a crossvalidate pipeline with outlier removal as part of the pipe
```

In [84]:

```
# but is 95% too much for B100?
(
    df_B100_songs
    .describe([0.025, 0.975])['2.5%':'97.5%':2].T
    .drop(['key', 'mode', 'time_signature'])
)

# do they need to be processed the same
# (I think so, but there are a lot of non-music or obscure things on Spotify)
```

Out[84]:

	2.5%	97.5%
acousticness	0.001	0.898
danceability	0.273	0.881
duration_ms	124000.000	379231.550
energy	0.204	0.941
instrumentalness	0.000	0.656
liveness	0.043	0.784

	2.5%	97.5%
loudness	-16.650	-3.232
speechiness	0.026	0.350
tempo	75.072	182.752
valence	0.141	0.964

METHOD 4: Domain Knowledge

good: include this

- source = me:
  - songs are at least 1 minute long and less than 10 minutes

In [52]:
```
# 82 song length outliers (0.32%)
too_short, too_long = sum(df_B100_songs.duration_ms < 60_000), sum(df_B100_songs.duration_ms > 600_000)
too_short, too_long, too_short + too_long, (too_short + too_long) / df_B100_songs.duration_ms.count()
```

Out[52]:
```
(35, 47, 82, 0.003214803779354687)
```

In [ ]:
```
# seems OK, and close to 99% interval
```

Notes on Importing and Avoiding Outliers / Errors:

- maybe a better idea for how to merge B100 and 10M
  - exact matches only
  - API
  - then approx matches for missing
- ALT (slow version)
  - API first
  - then match exact, then missing
- Neither will be 100%, probably best to just exclude outliers for predictions
- Both of the extreme outlier songs examined were mostly correct, but a sub-optimal version
  - audio feature data would likely be mostly correct for these songs
- Other than duration_ms, not sure if other audio features should be considered for outliers

METHOD 5: sklearn.neighbors.LocalOutlierFactor()

might be good, but would take weeks to tune hyperparameters

In [191…
```
from sklearn.preprocessing import QuantileTransformer
from sklearn.neighbors import LocalOutlierFactor
```

In [192…
```
X = df_10M[['acousticness','danceability', 'duration_ms', 'energy', 'instrumentalness',
            'liveness', 'loudness', 'speechiness', 'tempo', 'valence']]  # exclude key, time_signature, mode
X.index = df_10M['id']
```

In [197…
```
# do less rows to check speed
X_1k = X.head(1_000)
X_10k = X.head(10_000)
X_100k = X.head(100_000)
```

In [198…
```
%%time

qt = QuantileTransformer()
X_qt = qt.fit_transform(X_1k)
```

```
clf_out = LocalOutlierFactor()
X_out = clf_out.fit_predict(X_qt)
```

```
Wall time: 38.9 ms
```

In [194…

```
%%time

qt = QuantileTransformer()
X_qt = qt.fit_transform(X_10k)

clf_out = LocalOutlierFactor()
X_out = clf_out.fit_predict(X_qt)
```

```
Wall time: 870 ms
```

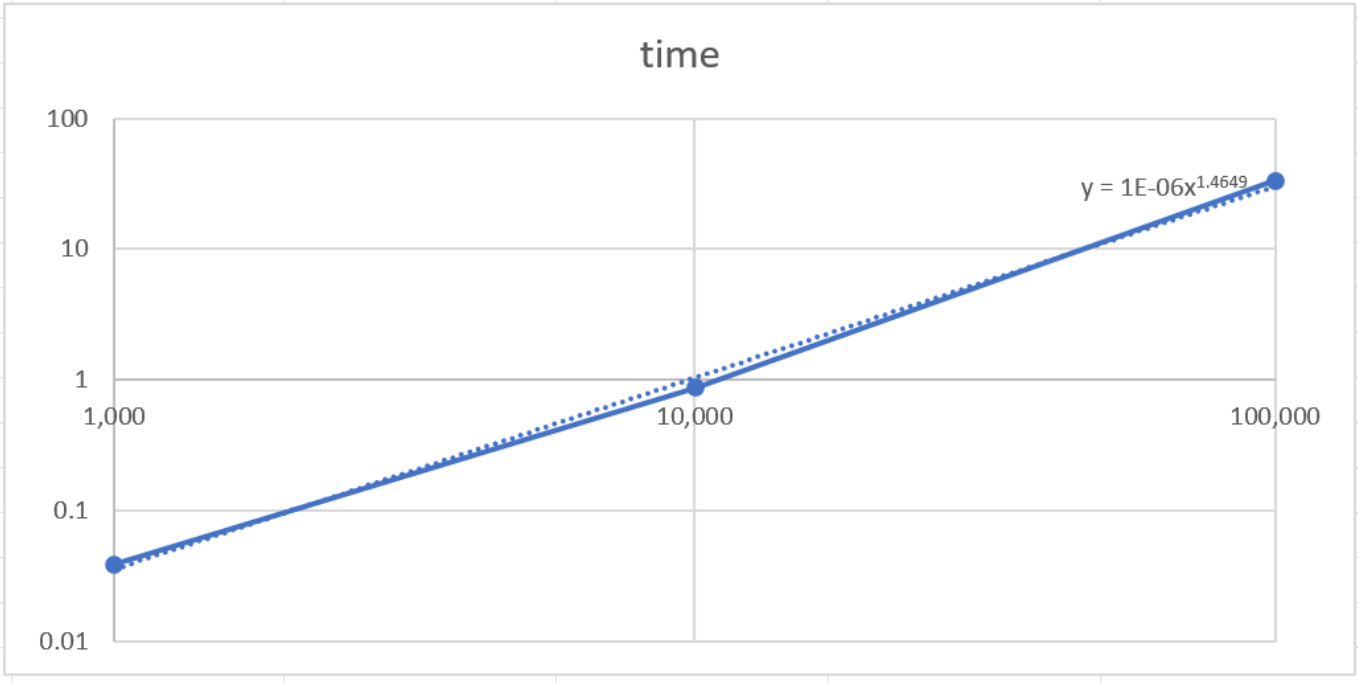
In [195…

```
%%time

qt = QuantileTransformer()
X_qt = qt.fit_transform(X_100k)

clf_out = LocalOutlierFactor()
X_out = clf_out.fit_predict(X_qt)
```

```
Wall time: 33.1 s
```

~5 hours per scenario

entries	time	predicted	minutes	hours
1,000	0.0389	0.025	0.0004	
10,000	0.87	0.724	0.0121	
100,000	33.1	21.11	0.3518	
10,000,000		17960	299.3	4.99



In [18]:

```
# check results - these look like outliers for sure
df_10M[df_10M.id.isin(set(X[X_out == -1].index))]

# the only songs on here are
"""
Jemand Lief Amok Auf Der Mayday ...But Alive  # very intense punk music, poor audio quality
```

```
Who Knows Where the Time Goes?  10,000 Maniacs  # sounds like a normal song, but has 2 min dead air at the end
"""
```

Out[18]:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	k
1708	1FsVUf5Uzhy7yAx0D7Q3cm	Nature Trails	#Sleep	sleep	2020-02-10	0.929	0.072	150000	0.484	0.918	
2325	1RqvQPm5ynqb8Py10SgIM3	Intermission with the Gods (Interlude)	tareed	NaN	NaT	0.972	0.300	170344	0.659	0.903	
4865	7xUthB9jFcA5MXMRWPqMdN	Jemand Lief Amok Auf Der Mayday	...But Alive	german punk	1998-05-01	0.000	0.409	251093	0.001	0.078	
4959	5AGk0U6FYY39FZcaCCnUo2	Thank You For Your Time	...music video?	tucson indie	2004-01-01	0.991	0.000	7000	0.353	0.000	
5105	3yY2ExH2Awk253MneKmOr8	.	.bipolar.	<NA>	2012-11-02	0.000	0.000	182000	0.000	0.000	
6013	1sTVvE61TBFJqifntliGWa	Pedo (おなら Furz Fart Pet)	1	sound effects	2012-02-20	0.000	0.000	5319	0.439	0.002	
6014	6xYDFftQhuMG1B5sT3g9si	Risas De Mujer (Frauen Lachen)	1	sound effects	2012-02-20	0.859	0.000	12125	0.924	0.025	
6015	4XrHRpovOtz1fJpBNHu4AY	Gran Pedo (おなら Furz Fart Pet)	1	sound effects	2012-02-20	0.000	0.000	3722	0.060	0.000	
6019	3tCUJaelXa5qtWWbyU39eb	Pedo 2 (おなら Furz Fart Pet)	1	sound effects	2012-02-20	0.867	0.000	3650	0.559	1.000	
6021	0Wwtqba02lFC7YC2A23gX4	Pedo Largo (おなら Furz Fart Pet)	1	sound effects	2012-02-20	0.000	0.000	4785	0.015	0.000	
6024	18CAPCgZDNtb3zeXrh0bkM	Orgasmos Hombre Y Mujer (orgasms オーガズム Orgasmen)	1	sound effects	2012-02-20	0.000	0.000	1757	0.943	1.000	
6406	5RawKDhVP4oH4AcR2ygkLf	Weg Weg - Weg Weg	1. FC Nuremberg FanChants	football	2006-01-01	0.994	0.000	14786	0.834	0.942	
6591	1gpzF6NlidDY8ufHUxlxOy	Weg Weg	1.FCN Fans Fangesänge	football	2016-03-23	0.994	0.000	14786	0.796	0.837	
7139	0o0GjGBZarwihzgIOxCuOz	Who Knows Where the Time Goes?	10,000 Maniacs	<NA>	1999-01-01	0.037	0.455	400027	0.007	0.013	
7288	34aOR4x0XqJN43HdBdv2Fm	Who Knows Where the Time Goes?	10,000 Maniacs	mellow gold	1999-01-01	0.037	0.454	400027	0.007	0.013	
7622	4BTlXQHCxrGFQ2QgB0pbL3	Lost - SMS	100 High Quality Ringtones	NaN	NaT	0.667	0.000	7784	0.118	0.999	
7633	207xn4qJzcsSxNIZa6vQOR	Boing and Chirp - SMS	100 High Quality Ringtones	NaN	NaT	0.212	0.000	5590	0.598	0.746	

In [25]:
```
# are these all songs?
df_10M[df_10M.id.isin(set(X[X_out == 1].index))].sample(20)
# no. there are 'football' and 'ringtone' genres included, some very short songs
```

Out[25]:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumenta
7816	1wSDy4kNfqyJixNgDrWsEL	071 Eloping With a Millionaire	100 Instrumentals	beats	2010-09-25	0.003	0.727	244253	0.566	
2651	3OPynmNaSuaOjPTdkmQW00	Okie Dokie Stomp (1954)	''Gatemouth'' Brown	NaN	NaT	0.642	0.480	153267	0.743	
4507	1YlhmgGDDah1cfadrcafWI	Wait a Minute	-M-	chanson	2016-01-22	0.045	0.767	192679	0.603	
6586	2Fl6pn7l998h2mW2wJJkWH	Meine Stadt	1.FCN Fans Fangesänge	football	2016-03-23	0.739	0.556	22700	0.902	
2546	1S4MM1ukLm3ourrBJGHniC	[whispers Indistinctly]	*uicideboy*	underground hip hop	2020-02-14	0.182	0.899	164624	0.716	
5982	71LXYDPiZ7OLTSPdFFGn17	Dotdotdotsix	0N4B	african experimental	2018-11-28	0.011	0.774	335018	0.402	
6329	2oEltWv1p5YgETvzjtgmCm	Graveyard Paradise	1-2-6	norwegian prog	1968-06-18	0.148	0.576	177072	0.527	
6322	0ATGklmLJOBcBqEhEb1Bgg	Who's Been Sleeping in My Bed	1-2-6	norwegian prog	1968-06-18	0.003	0.693	125907	0.763	
4519	34jpGj9jQKWi2oU9vL9wae	Close to Me	-M-	chanson	1999-10-05	0.010	0.665	239973	0.831	
1835	73KwtRxljSDxlLvSByXkva	Top	$Ha Hef	NaN	NaT	0.219	0.702	204356	0.947	
2895	4XCP2gVRk858dlWPXuhIpB	Jason's Bar Mitzvah	'Falsettos' 2016 Broadway Company	NaN	NaT	0.817	0.384	308133	0.309	
6846	7hIjqPjBWMIZBjrhPJLKi0	La pression	10 Kret	rap marseille	2010-12-02	0.210	0.669	265520	0.799	
2052	4IfMkREYOP27mLyGvdGbND	Saucy	$b	kenyan hip hop	2020-04-07	0.426	0.634	171207	0.328	
2049	1nmDspbx8RJLQkXJmFo5GS	Who TF Is You	$avage	trap	2017-02-10	0.118	0.812	318109	0.531	
831	5aUTSJltF24eevgzM2j7gE	My Bologna	"Weird Al" Yankovic	<NA>	2019-07-05	0.083	0.687	119147	0.906	
991	0jfsW09C07H0jEftKVFb2v	Alimony	"Weird Al" Yankovic	comic	1988-01-01	0.146	0.575	196347	0.941	
810	3A3BVdhNfl6EjWzxFSJRxt	Lasilasi	"Trojan"	NaN	NaT	0.415	0.628	230609	0.552	
4888	6wXzhHszTGEZsTcKHXVOt2	Instinto (...Nuevas Fronteras)	...Por Hablar	peruvian metal	2001-01-01	0.001	0.435	287333	0.889	
367	7Lo7KR81hqbEUheJ80v1wT	It Is Worthy To Praise Thee	"Drevnerussky Rospev" Male Choir	orthodox chant	1991-01-01	0.994	0.116	153365	0.096	
8354	5QJJ8EsBBEJ30VrgMPTGUa	Teeth Fletcher (Part 2)	100000 TONNEN KRUPPSTAHL	epic doom	2019-06-28	0.000	0.305	912288	0.793	

not the best method:

- may need multiple hours to calculate the best hyperparameters
- no way to quantify how good these results are
- NOTE: should exclude some genres entirely (examples: 'ringtone', 'football', 'sound effects', 'sleep')

In [ ]:
```
# def print_outlier_results():
#     pass


# n_quantiles = [100, 1000, 10000]
# n_neighbors = [1, 2, 4, 8, 20, 50, 100]

# for i in n_quantiles:
#     qt = QuantileTransformer(n_quantiles=i)
#     X_qt = qt.fit_transform(X)

#     for j in n_neighbors:
#         clf_out = LocalOutlierFactor(n_neighbors=j)
#         clf_out.fit_predict(X_qt)

#         print_outlier_results()
```

METHOD 6: Genres to Exclude

SUCCESS, include this method

In [7]:
```
set_genre = set(df_10M.genre.unique())
```

In [97]:
```
# check some genres based on outlier song length
song_length = 2_400_000
df_10M[df_10M.duration_ms > song_length].sample(10)[['song', 'artist', 'duration_ms', 'genre']]
```

Out[97]:

	song	artist	duration_ms	genre
454956	Composition 340	Ann Rhodes	2789347	<NA>
6060163	Nukleuz Hard Dance Classics 2011 - DJ Mix 1	Nukleuz DJs	3878972	NaN
825676	Late Night Tales: Belle and Sebastian - Contin...	Belle & Sebastian	4738853	alternative rock
2699131	Awel Hamsa - Live	Farid al-Atrash	3323797	NaN
8814816	Alf Leila We Leila	Umm Kulthum	2497280	classic arab pop
2554248	Psychologically Ultimate Seashore	Environments	3603400	bulgarian experimental
6594774	Digital Drugs 3: Shape The Machine DJ Mix (fea...	Psychedelic Mushrooms Infected With Hallucinogens	3431352	NaN
2888005	Continuous mix	Freaks	4658000	NaN
3547124	Hard Kryptic Records Yearmix 2018 - Continuous...	How Hard	2818399	dark hardcore
5650095	Imagine the World, vol 1 - Mixed by Misja Hels...	Misja Helsloot	4648952	uplifting trance

In [100...
```
# sets that aren't music
# by inspection using songs less than 1min or more than 10min (+ more extreme)
genres_to_exclude = set([
    'sleep', 'football', 'halloween', 'birthday', 'lullaby', 'ringtone', 'fan chant',
    'sound effects', 'spoken word', 'bible', 'prank', 'wrestling', 'language', 'oratory',
    'erotica', 'tone', 'vintage radio show', 'sound', 'quran', 'islamic recitation',
    'reading', 'asmr', 'mindfulness', 'meditation', 'guided meditation', 'workout product',
    'theme', 'environmental', 'motivation'
])
```

In [114...
```
# inspect some genres -  this looks like it's good
df_10M[df_10M.genre == 'theme'].sample(10)[['song', 'artist', 'duration_ms', 'genre']]
```

Out[114...

	song	artist	duration_ms	genre
6171773	Andy Reynold's Reel	Orlando Pops Orchestra	82587	theme
5743792	Once Upon a Time (Theme)	Mount Royal Orchestra	86831	theme
1343728	Magilla Gorilla	Cartoon Theme Ensemble	68667	theme
1893187	Theme (From "Dream On")	Daniel Caine Orchestra	178880	theme
7861388	Let's Make a Deal	TV Tunesters	90022	theme

	song	artist	duration_ms	genre
5313382	Tubular Bells (From "The Exorcist")	Mark Ayres	210387	theme
1618937	Spiderman & his amazing friends	Coded Channel	63190	theme
1343683	Heigh Ho (From Snow White)	Cartoon Theme Ensemble	112133	theme
1429864	Max Headroom	Charlie's Angels	66080	theme
1343716	Someday (From The Hunchback Of Notre Dame)	Cartoon Theme Ensemble	271747	theme

In [112…

```
# inspect some genres -  this looks like it's good
df_10M[df_10M.genre.isin(genres_to_exclude)].sample(10)[['song', 'artist', 'duration_ms', 'genre']]
```

Out[112…

	song	artist	duration_ms	genre
950901	Numbness	Binaural Beats	141196	sleep
5887853	Flamboyant Sea	Nature Sounds	147456	sleep
6686786	Rain Gradual	Rain Sounds	180350	sleep
8689430	Les Haubans	Traditional	164440	sleep
607615	Tape Play	Asmr Aston	149302	asmr
5743768	Home Alone Theme	Mount Royal Orchestra	295471	theme
6085920	Fantastic Bayside Ocean Waves	Ocean Sounds	254563	sleep
2229261	Legend of the Twelve Robbers	Don Cossack Choir Serge Jaroff	290437	sleep
9240090	Core	Yoga Music	316526	sleep
4357617	Kapitel 36: Reise zum Mittelpunkt der Erde - T...	Karlheinz Gabor	120647	reading

In [117…

```
# there are 2 songs in B100 that are excluded (maybe 2 versions of the same song)
df_10M[(df_10M.genre.isin(genres_to_exclude)) & (df_10M.in_B100)].head()
```

Out[117…

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
1046771	40lzYjHgIw0r55twhZI3SK	Monster Mash	Bobby "Boris" Pickett And The Crypt-Kickers	halloween	1900-01-23	0.924	0.574	187922	0.612	0.200	2
1046772	3XYbmvGhgbfvlvX8xdCG9u	Monsters' Holiday	Bobby "Boris" Pickett And The Crypt-Kickers	halloween	2013-03-08	0.526	0.714	189000	0.406	0.000	0

In [118…

```
# 2 out of 10k songs
sum(df_10M.genre == 'halloween')
```

Out[118…

9869

In [119…

```
# how many songs can be excluded based on genre
sum(df_10M.genre.isin(genres_to_exclude))
```

Out[119…

182474

METHOD 6b: exclude other AF extremes

PARTIAL SUCCESS, include parts of this method

In [130…

```
AF = ['acousticness','danceability', 'duration_ms', 'energy', 'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo

# 99% interval (0.5% to 99.5%)
AF_min, AF_max = 0.005, 0.995

# create df for 99% interval
df_99 = df_10M[AF].describe([AF_min, AF_max]).T[['0.5%', '99.5%']]
df_99
```

Out[130…

	0.5%	99.5%
acousticness	0.000	0.995
danceability	0.064	0.918
duration_ms	21960.000	871749.990
energy	0.005	0.995
instrumentalness	0.000	0.975
liveness	0.030	0.961
loudness	-34.669	-1.825
speechiness	0.024	0.947
tempo	48.370	201.457
valence	0.000	0.973

In [136…

```
AF_min, AF_max = df_99.loc['tempo']
AF_min, AF_max
```

Out[136…

```
(48.369998931884766, 201.45700073242188)
```

In [165…

```
# check some genres based on outlier song length
for feature in AF:
    AF_current = feature
    AF_min, AF_max = df_99.loc[AF_current]

    print('--------------------------------------------------------------------------------------------------------------')
    print(AF_current)
    print()
    print(f'min: {AF_min:.3f}, max: {AF_max:.3f}')
    print()

    # random sample
    print('random sample of outliers:')
    display(df_10M[(df_10M[AF_current] < AF_min) | (df_10M[AF_current] > AF_max)].sample(5)[['song', 'artist', 'duration_ms',

    # how many are in the B100
    print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] < AF_min) | (df_10M[AF_current] > AF_max)].in_B10
    print()
    print('first 5 on B100:')
    display(df_10M[((df_10M[AF_current] < AF_min) | (df_10M[AF_current] > AF_max)) & (df_10M.in_B100)].head())

    print('\n')
```

```
--------------------------------------------------------------------------------------------------------------
acousticness

min: 0.000, max: 0.995

random sample of outliers:
```

	song	artist	duration_ms	genre	acousticness
4327118	Me Khosk Unim - Duduk	Kamo Nazarian	193227	duduk	0.996
9276796	Cluster Of Minds	Yuri Gagarin (SWE)	342491	space rock	0.000
7137678	Rain	Samael	240880	metal	0.000
8082522	Zombified	The Crown	155109	metal	0.000

	song	artist	duration_ms	genre	acousticness
8727602	Violin Partita No. 1 in B Minor, BWV 1002: V. ...	Trio SR9	120200	classical	0.996

```
number of outliers in the B100:  3
```

```
first 5 on B100:
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	k
297497	2O2ii9OPZYh1NBXo9FtE0Y	Check My Brain	Alice In Chains	rock	2009-01-01	0.000	0.441	237907	0.845	0.053	
3161426	7bbdNZXwpwMSNAvoxMjJPc	Speak	Godsmack	alternative metal	2006-01-01	0.000	0.417	237280	0.918	0.001	
8030669	7DoMGZLVzga3vhyZlb0hBX	An Honest Mistake	The Bravery	rock	2005-01-01	0.000	0.465	219707	0.883	0.506	

```
----------------------------------------------------------------------------------------------------
danceability
```

```
min: 0.064, max: 0.918
```

```
random sample of outliers:
```

	song	artist	duration_ms	genre	danceability
4748927	Si voi riempiri nu bellu buttaru (Proverbio)	Leo Siviglia	13320	NaN	0.000
256590	Top	Alex Arrowsmith	442131	<NA>	0.058
6684274	Etho's Anthem	Raider	200202	<NA>	0.968
5888345	Mists Outside	Nature Sounds	143733	sleep	0.061
1748408	Perreo Santafesino, Pt. 3	Cue DJ	155202	NaN	0.953

```
number of outliers in the B100:  166
```

```
first 5 on B100:
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liven
28478	7eH6rFngQor2A08suABvGI	Be True To Yourself	2nd II None	gangster rap	1991-01-01	0.003	0.970	164960	0.578	0.000	6	0.
28479	5MGSSII7LQxTJ2IJsyRDlq	If You Want It	2nd II None	gangster rap	1991-01-01	0.009	0.930	226667	0.547	0.000	0	0.
42144	69bHJ9qs5FrUJbKP8xU8uZ	Disco Inferno	50 Cent	hip hop	2005-03-03	0.206	0.925	214227	0.659	0.000	3	0.
59538	0VZe8C7xgAIQC0E0qEVIEh	Mood Swings	A Boogie Wit da Hoodie	rap	2020-02-14	0.211	0.941	156960	0.676	0.000	4	0.
99508	3krjKKOH1CjeGXn6hIqyLA	Aaron's Party (Come Get It)	Aaron Carter	europop	2003-10-21	0.040	0.926	205827	0.904	0.000	1	0.

```
----------------------------------------------------------------------------------------------------
duration_ms
```

```
min: 21960.000, max: 871749.990
```

```
random sample of outliers:
```

	song	artist	duration_ms	genre	duration_ms
14174	Butt Cheek Squeak	140 Farts	5669	sound effects	5669

	song	artist	duration_ms	genre	duration_ms
6282925	Burn Something	Parts & Labor	13013	<NA>	13013
4072280	Where the Wild Thyme Blows	John Blackwood McEwen	907560	<NA>	907560
8932191	Male All the Music You Ask for More With FX So...	Vicious Vocal Sound Effects	3111	sound effects	3111
5707139	Pumpkin Patch	Monster Mash Halloween	13250	halloween	13250

```
number of outliers in the B100:  7

first 5 on B100:
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
2243576	1bHq4U7NKCgeIuX0o9geEC	Try Me, I Know We Can Make It	Donna Summer	dance pop	1976-03-05	0.034	0.869	1077133	0.556	0.090	0
2243609	6Szw3sQC5Zssr15AJsUY9J	Love To Love You Baby	Donna Summer	dance pop	1975-08-27	0.292	0.622	1008533	0.425	0.000	10
3462047	4Ce66JznW8QbeyTdSzdGwR	Chameleon	Herbie Hancock	jazz	1973-10-26	0.104	0.581	941360	0.672	0.856	8
3704871	1S73njkLJ5orszfbNeqDQs	By The Time I Get To Phoenix	Isaac Hayes	soul	2016-01-01	0.215	0.398	1124153	0.261	0.000	8
6645092	45LrQ3tg1z8plpuQRCuSwE	Trapped In The Closet	R. Kelly	NaN	2005-11-08	0.539	0.551	992160	0.583	0.000	4

```
--------------------------------------------------------------------------------------------------------
energy

min: 0.005, max: 0.995

random sample of outliers:
```

	song	artist	duration_ms	genre	energy
784977	97Hz Sine Wave	Bass Mekanik	15000	miami bass	0.000
8804924	4'33": III. —	Ulrich Krieger	87000	<NA>	0.000
5545748	Thousand Year Dreaming: The Chi Stirs	Michael Pugliese	640053	avant-garde	0.002
2079439	A Photograph of You - Live in Hammersmith; 201...	Depeche Mode	201710	dance rock	0.998
6728109	Light Surrounds Me	Raspberry Bulbs	202088	post-doom metal	0.998

```
number of outliers in the B100:  5

first 5 on B100:
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness
1751547	2rTYgHxgcndkUrRoU7x0Sv	Mr. Vain	Culture Beat	europop	1993-01-01	0.048	0.676	336840	0.996	0.262
2773135	7DyCmfT0WNViykLTHGT3yO	Under And Over It	Five Finger Death Punch	alternative metal	2011-01-01	0.000	0.473	218133	0.996	0.000
3109582	7pHZZhLoWuyUMSquwPZMo0	Ooh Aah... Just A Little Bit	Gina G	europop	2007-12-11	0.029	0.564	213947	0.996	0.010
3835955	3k6L0H08zSY5yRDgqI8fqx	Just Because	Jane's Addiction	rock	2003-01-01	0.000	0.369	231853	0.996	0.140

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness
6559645	7ashP0HnlAkHKWEK91xx1v	Bongo Rock	Preston Epps	NaN	1960-05-26	0.807	0.376	126613	0.996	0.907

---

instrumentalness

min: 0.000, max: 0.975

random sample of outliers:

	song	artist	duration_ms	genre	instrumentalness
6575289	Applause In A Small Hall	Pro Studio Library - Sound Effects Download Se...	19253	sound effects	0.994
6805005	Arcoiris	Relaxphonic	311333	NaN	0.988
2026087	Binary	Death Toll 80K	43293	grindcore	0.977
3073547	Geistliche Chore, Vol. 1: Maria und die arme S...	Gerd Guglhör	142480	german choir	0.978
2667528	Sleeping in	FX Sound Forest	131819	sleep	0.999

number of outliers in the B100: 3

first 5 on B100:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
1498108	5uydHeAg7MsIKSvS7GLw3a	Land Of 1000 Dances	Chris Kenner	rhythm and blues	2007-06-01	0.646	0.537	81467	0.660	0.981	3
1937032	6Sy9BUbgFse0n0LPA5lwy5	Sandstorm	Darude	trance	2001-01-01	0.141	0.528	225493	0.965	0.985	11
8464041	2n4jL9nv4CS7uwqmu4MHu8	The Green Mosquito	The Tune Rockers	NaN	2017-02-24	0.877	0.646	139040	0.881	0.980	4

---

liveness

min: 0.030, max: 0.961

random sample of outliers:

	song	artist	duration_ms	genre	liveness
6192671	Blues Etude - Live	Oscar Peterson	325867	adult standards	0.967
7940275	Wonderboy - Live	Tenacious D	240080	rock	0.984
9175817	Data Reveal Musical Tense Sound Design Multime...	World Class Sound Effects	6026	sound effects	0.000
8395997	Strip the Willow	The Scottish Country Dance Band	173560	traditional scottish folk	0.029
5444271	La Calle (La Calle Está Caliente) - En Vivo	Maykel Blanco Y Su Salsa Mayor	674441	timba	0.971

number of outliers in the B100: 194

first 5 on B100:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
7154	5KqlNh56tApMe4LDMfLwOp	Few And Far Between	10,000 Maniacs	mellow gold	1992-01-01	0.008	0.614	194373	0.928	0.001	6
42139	7JeKXMQKm6GoLGTkNy2jZ0	If I Can't	50 Cent	hip hop	2003-02-06	0.244	0.892	196640	0.631	0.000	11
350505	5jwQH54Qg6gcm4hqXm0uLt	Magical Mystery Tour	Ambrosia	folk rock	2002-01-01	0.081	0.257	232800	0.834	0.000	9
477959	2qAqxEV45AAAN60GtFhvW7	Iesha	Another Bad	new jack	1991-02-11	0.040	0.714	261907	0.786	0.000	1

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	l
			Creation	swing								
542550	0lpIVCLNQRBKffcRRNrHyh	Freeway Of Love	Aretha Franklin	soul	1985-01-01	0.242	0.688	352093	0.899	0.000	2	

---

loudness

min: -34.669, max: -1.825

random sample of outliers:

	song	artist	duration_ms	genre	loudness
2974052	La Pareja Ideal	Gabino Pampini	247200	tropical	-1.667
9006896	Symphony No. 7 in C Major, Op. 60: II. Memorie...	WDR Sinfonieorchester	134959	classical	-34.985
1965942	Drum, 6 Feet Live Studio: Announcement	David Chesky	5333	<NA>	-55.758
8509794	Imperayritz De La Ciutat Joyosa (Anonymous)	Theatrum Instrumentorum	572907	medieval ensemble	-34.788
5460153	Meditación Asmr 7	Meditación Tántrica ASMR	76046	asmr	-34.952

number of outliers in the B100: 50

first 5 on B100:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	l
588550	1EiwZsCTu8cGSRdog30oYY	Just Stand Up!	Artists Stand Up To Cancer	dance pop	2008-01-01	0.174	0.560	217067	0.958	0.000	0	
674895	0BAeqyZ6pc4Yb9EoiMgMfO	Bumble Boogie	B. Bumble & The Stingers	rock-and-roll	2015-01-01	0.452	0.487	131342	0.949	0.944	9	
766922	7JUalHySYqNFmz1P3nfDyz	Too Little Too Late	Barenaked Ladies	pop rock	2000-09-01	0.031	0.497	203773	0.987	0.005	9	
899801	0Z9YP4ntjcqlMCfgi5Eete	Wild West Show	Big & Rich	country	2004-04-20	0.026	0.551	260813	0.879	0.000	0	
899807	56EYmR9IrFKKYdygcKXrvH	Comin' To Your City	Big & Rich	country	2005-11-15	0.258	0.559	207133	0.944	0.000	9	

---

speechiness

min: 0.024, max: 0.947

random sample of outliers:

	song	artist	duration_ms	genre	speechiness
2481975	Humpty Dumpty	Ella Jenkins	46693	children's music	0.959
3887366	Schwarzes Requiem, Kapitel 72	Jean-Christophe Grangé	364805	lesen	0.968
274781	Kapitel 163 - Opferzahl	Alexander Simon	125727	reading	0.959
1988561	Mieses Karma hoch 2, Kapitel 15	David Safier	337509	lesen	0.948
8113302	Atmospheres	The Dunes	235787	NaN	0.023

number of outliers in the B100: 58

first 5 on B100:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liv
216679	3ZQ7bFBquNdzelPHbzwJd6	Feels So Right	Alabama	country rock	1980-01-01	0.522	0.546	215787	0.338	0.000	10	
312525	4WUnM4KNZ0kjp0CUeoyOnS	I Swear	All-4-One	dance pop	1994-01-01	0.235	0.532	259853	0.407	0.000	6	
449440	6Ycf7Ch2VlEKlORbz7yfpJ	Sweet Love	Anita Baker	soul	1986-03-20	0.360	0.504	266201	0.728	0.000	10	
470287	2fYYmPNwDaNpw1KRSxpQ5H	Why	Annie Lennox	mellow gold	1992-04-02	0.281	0.532	293838	0.471	0.000	0	
992722	7haN4XreV4WkoBQmkYM709	Drink On It	Blake Shelton	country	2011-07-11	0.036	0.643	211067	0.725	0.000	7	

---

tempo

min: 48.370, max: 201.457

random sample of outliers:

	song	artist	duration_ms	genre	tempo
3709808	I'll see you in my dreams	Isham Jones	173893	tin pan alley	207.589
3110117	Come to Bed	Gina79	10853	erotica	0.000
2232803	Amaneciendo	Don Medardo y Sus Players	230806	ecuadorian pop	206.169
7441962	Fuck The Hook	Skinnyman	294107	uk hip hop	204.462
2873725	String Quartet No. 7 in A Major, Op. 2 No. 1, ...	Franz Joseph Haydn	260800	<NA>	42.169

number of outliers in the B100: 145

first 5 on B100:

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	ke
11596	2SLZ2rX5773RS16mcqBxG7	Peaches & Cream	112	gangster rap	2001-01-01	0.003	0.677	193093	0.520	0.000	
129087	3c5Rv4XYKsAN4TV4f7hUUe	Tuff	Ace Cannon	NaN	2005-01-01	0.397	0.590	136667	0.250	0.827	1
168168	4AFHB8laVETw8ruPZscDbe	Remember (Walking In The Sand)	Aerosmith	rock	1979-11-16	0.237	0.269	244400	0.509	0.001	
211679	55shJy826YxyMjwqbOTDtJ	Teach Me Tonight	Al Jarreau	adult standards	1981-01-01	0.087	0.285	253133	0.294	0.000	
223480	2NKXcEKxgJFLW79EpjofqI	The Talkin' Song Repair Blues	Alan Jackson	country	2004-08-24	0.325	0.469	180373	0.607	0.000	

---

valence

min: 0.000, max: 0.973

random sample of outliers:

	song	artist	duration_ms	genre	valence
6888455	Flintstone Run	Ring Tonez	1856	NaN	0.000
1079848	Soul Limbo	Booker T. & the M.G.'s	142933	soul	0.977
9422501	Q太郎	張圓圓	81409	NaN	0.977

	song	artist	duration_ms	genre	valence
2901177	Diomay - Interlude	Fredo	12307	old school rap francais	0.000
3340087	Goblin	Halloween Sound Effects	9565	halloween	0.000

```
number of outliers in the B100:  107
```

```
first 5 on B100:
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
79516	2HPB3px8MJZRMfu1L65Z41	Does Your Mother Know	ABBA	europop	1979-01-01	0.151	0.728	193440	0.865	0.000	7
782180	7BxmI8dRehWmhyFBjPxUg9	Cruising For Bruising	Basia	smooth jazz	1990-02-12	0.091	0.782	249827	0.522	0.002	5
814987	7EcsJXtKtUkzTa8mgjJKiE	The Woman In You	Bee Gees	mellow gold	1977-12-13	0.114	0.809	243933	0.491	0.000	9
825385	4yHq1fsp9vhX1T34zoHnAu	Satin Sheets	Bellamy Brothers	country rock	1976-07-01	0.044	0.694	247053	0.831	0.000	7
871838	1BPXaPBTN9oF2HJ4PDx2sp	Afrikaan Beat	Bert Kaempfert And His Orchestra	adult standards	1962-01-01	0.585	0.654	146133	0.492	0.009	10

In [ ]:

```
# most look like they should be left, sometimes outliers are optimal
# check a few,
# valence=0, speechiness>AF_max, tempo<AF_min (esp 0), loudness<AF_min, energy<AF_min, danceability<AF_min
```

In [189…

```
# many valence <= 0.00001 aren't songs
# check == 0 to confirm

AF_current = 'valence'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} min: {AF_min:.6f}')
print('number to exclude:', sum(df_10M[AF_current] < AF_min))
display(df_10M[(df_10M[AF_current] < AF_min)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])

# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] < AF_min)].in_B100))
display(df_10M[((df_10M[AF_current] < AF_min)) & (df_10M.in_B100)].head())
```

```
valence min: 0.000010
number to exclude: 46338
```

	song	artist	duration_ms	genre	valence
3429493	Surprise	Heinz Roemheld	13304	classic soundtrack	0.000
1564050	Knocking Window Inside Sound Effects Sound Eff...	City Lights Sound Effects	4040	sound effects	0.000
7612560	Comedy Slide Whistle Twirly Burly Ride with Co...	Sports Charge	6013	NaN	0.000
2562742	Mongolian Throat Singing, No. 9	Erdem Baatar	20264	mongolian folk	0.000
3336048	What's That?	Halloween FX Productions	6409	halloween	0.000
7288438	Time To	Settle The Score	29853	beatdown	0.000
6404678	50 Russian Folk Songs, TH 176: No. 36, Oh, Mea...	Peter Hill	16420	classical	0.000
2761447	Carpet Beater (Version 1) [Rug Carpetbeater Ru...	Finnolia Sound Effects	16131	sound effects	0.000
8796023	Drums - Four Flams	US Navy Band	44133	marching band	0.000
5747586	Running	Movie Sound Effects	11715	sound effects	0.000

```
number of outliers in the B100:  1
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
5015786	63kd4m3VFxcJjPVVtbVNAu	Hello, Dolly!	Louis Armstrong And The All Stars	adult standards	1964-10-25	0.842	0.000	147000	0.405	0.001	0

In [185…

```
# CONSIDER EXCLUDING valence == 0 (exactly the same as bottom 0.5%ile)

AF_current = 'valence'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} min: {AF_min:.6f}')
print('number to exclude:', sum(df_10M[AF_current] == 0))
display(df_10M[(df_10M[AF_current] == 0)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])

# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] == 0)].in_B100))
display(df_10M[((df_10M[AF_current] == 0)) & (df_10M.in_B100)].head())
```

```
valence min: 0.000010
number to exclude: 46338
```

	song	artist	duration_ms	genre	valence
2622749	Mentally Ill Farmer Is Finally Getting Help	Eugene Mirman	4127	comedy	0.000
3368243	Hannah	Happy Birthday Singers	16147	NaN	0.000
1544639	Hidden 1	Chuck Loeb	10000	<NA>	0.000
3272449	Meteo	Guerilla Poubelle	7027	french rock	0.000
2025713	Blank Track	Death Ride 69	4000	NaN	0.000
762815	Harp Interlude	Barbara Russell	10373	NaN	0.000
7566916	Whip - Bull Whip: Single Crack, Whips	Sound Effects Library	4672	sleep	0.000
1339970	Intro - Live from Club Bla - Oslo, Norway	Carrie Rodriguez	8560	folk	0.000
6984804	Give Me A Break	Ron Jarzombek	4133	<NA>	0.000
3924456	Suite In D - Prince Of Denmark's March, "Trump...	Jeremiah Clarke	176573	classical	0.000

```
number of outliers in the B100:  1
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
5015786	63kd4m3VFxcJjPVVtbVNAu	Hello, Dolly!	Louis Armstrong And The All Stars	adult standards	1964-10-25	0.842	0.000	147000	0.405	0.001	0

In [182…

```
# EXCLUDE speechiness > 0.947000 (99.5%ile)
# most aren't songs

AF_current = 'speechiness'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} max: {AF_max:.6f}')
print('number to exclude:', sum(df_10M[AF_current] > AF_max))
display(df_10M[(df_10M[AF_current] > AF_max)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])

# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] > AF_max)].in_B100))
display(df_10M[((df_10M[AF_current] > AF_max)) & (df_10M.in_B100)].head())
```

```
speechiness max: 0.947000
number to exclude: 46959
```

	song	artist	duration_ms	genre	speechiness
1178422	Märchen von einem, der auszog das Fürchten zu ...	Brüder Grimm	86821	reading	0.949

	song	artist	duration_ms	genre	speechiness
2057291	Shaorma (Skit) [feat. Teo]	Deliric	53332	romanian rap	0.948
465443	Das Gurkodil	Anne Rottenberger	252133	kinderchor	0.957
7243203	Kids	Sebastian Maniscalco	493520	new comedy	0.948
5857232	Kapitel 281 - Think and Grow Rich - Deutsche A...	Napoleon Hill	126600	motivation	0.960
2137351	Schwarzes Requiem, Kapitel 28	Dietmar Wunder	344822	lesen	0.953
5530491	Timeline - rejsen til fortiden, del008	Michael Crichton	311950	reading	0.953
1499491	Episode One: Exchange Student	Chris Lilley	116750	australian comedy	0.950
6518958	Pol Pot	Pol Pot	119676	spanish post-punk	0.959
7889125	Kapitel 54 - Der dunkle Garten	Tana French	241758	lesen	0.967

```
number of outliers in the B100:  0
```

id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness

In [183…

```
# TEMPO???
# some are still songs, check tempo==0

AF_current = 'tempo'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} min: {AF_min:.6f}')
print('number to exclude:', sum(df_10M[AF_current] < AF_min))
display(df_10M[(df_10M[AF_current] < AF_min)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])

# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] < AF_min)].in_B100))
display(df_10M[((df_10M[AF_current] < AF_min)) & (df_10M.in_B100)].head())
```

```
tempo min: 48.369999
number to exclude: 47156
```

	song	artist	duration_ms	genre	tempo
7037345	Press Studs Undo	Rude & Crude FX	11347	NaN	0.000
2603250	Iturralde	Espanyol Supporters	12827	fan chant	0.000
4183500	Ende Der Diskussion, Teil 1	Josef Hader	110092	NaN	35.636
2150604	Somebody Loves Me	Dinah Shore	167547	adult standards	43.394
2140116	Stretcher Moved in Ambulance with Alarm	Digiffects Sound Effects Library	12597	sound effects	0.000
785038	Full Bandwidth Pink Noise	Bass Mekanik	60760	miami bass	0.000
6574330	Boxing Ring Bell 1	Pro Sound Effects Library	10027	sleep	0.000
4745994	Nerve Event #6	Leo Ciesa	8200	jazz trumpet	0.000
3204870	White Noise - Bit Crush 6,3 khz	Granular	116129	sleep	0.000
7721178	Skit 1	Streetlordz	26427	detroit hip hop	34.459

```
number of outliers in the B100:  4
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	ke
952897	5ZrKA1WzfkiVsfTzR2CHV4	I'll Be Home For Christmas	Bing Crosby	adult standards	2006-01-01	0.961	0.329	73040	0.065	0.005	
5015786	63kd4m3VFxcJjPVVtbVNAu	Hello, Dolly!	Louis Armstrong And The All Stars	adult standards	1964-10-25	0.842	0.000	147000	0.405	0.001	
6333498	3gIBSlXYIN1mru35l4LWPB	Still Crazy After All These Years	Paul Simon	rock	1975-10-25	0.800	0.267	206533	0.252	0.000	

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	ke
7143618	0wTfrDYntTLsYJ37t3UxkH	As Long As She Needs Me	Sammy Davis Jr.	adult standards	1963-01-01	0.444	0.163	184227	0.235	0.000	

In [184…

```
# EXCLUDE tempo == 0
# most aren't songs

AF_current = 'tempo'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} min: {AF_min:.6f}')
print('number to exclude:', sum(df_10M[AF_current] == 0))
display(df_10M[(df_10M[AF_current] == 0)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])

# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] == 0)].in_B100))
display(df_10M[((df_10M[AF_current] == 0)) & (df_10M.in_B100)].head())
```

```
tempo min: 48.369999
number to exclude: 36609
```

	song	artist	duration_ms	genre	tempo
3335519	Demon Rat	Halloween FX Productions	4630	halloween	0.000
1980450	Piano Axioms Suite: I. Zero Is a Natural Number	David Mandelberg	4125	<NA>	0.000
1289916	Game Start (NES ver.)	Capcom Sound Team	8533	japanese vgm	0.000
7548250	Pink Noise	Soothing Sounds	569083	sleep	0.000
700926	Outside Refrigerator Unit	Baby Lullaby	69957	sleep	0.000
1902163	Bloco de Maluco	Daniel Sansil e os Maluco do Brasil	10400	musica cearense	0.000
6914809	Shaggy - Like It's a Gggghost, Yikes	Robert Burton - Deblume	8152	NaN	0.000
5022281	Interlude	Louise Vertigo	13200	NaN	0.000
3518317	Intro	HomeGrown Kush	12278	NaN	0.000
8492022	Relaxing Rain & White Noise (Thunderstorm)	The White Noise Zen & Meditation Sound Lab	160625	sleep	0.000

```
number of outliers in the B100:  1
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
5015786	63kd4m3VFxcJjPVVtbVNAu	Hello, Dolly!	Louis Armstrong And The All Stars	adult standards	1964-10-25	0.842	0.000	147000	0.405	0.001	0

In [186…

```
# EXCLUDE loudness < -34.668999
# many aren't songs

AF_current = 'loudness'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} min: {AF_min:.6f}')
print('number to exclude:', sum(df_10M[AF_current] < AF_min))
display(df_10M[(df_10M[AF_current] < AF_min)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])

# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] < AF_min)].in_B100))
display(df_10M[((df_10M[AF_current] < AF_min)) & (df_10M.in_B100)].head())
```

```
loudness min: -34.668999
number to exclude: 47158
```

	song	artist	duration_ms	genre	loudness
3876065	After Sunset	Jazz Lounge Music Club Chicago	69500	spa	-41.988

	song	artist	duration_ms	genre	loudness
7573518	Bird Songs	Sounds Of Bird Songs	68415	birdsong	-38.566
785132	52Hz Sine Wave	Bass Mekanik	15000	miami bass	-35.830
626904	ricercar spagnuola, "duna cossa spagnola": Ric...	Atrium Musicae de Madrid	48920	musica antigua	-40.033
3930291	Una Mattina: No. 4, Leo	Jeroen van Veen	405240	classical piano	-36.489
3179861	Eleven Note Pieces & Decimal Passacaglia: à Ro...	Gordon Mumma	7507	avant-garde	-35.285
3244524	Dallgë Deti	Grupi i Lapardhas Gjirokastër	206547	NaN	-40.578
9425284	Butcher	王宇波	103393	chinese soundtrack	-34.734
3572540	Christe, Redemptor omnium	Huw Williams & organist	169600	NaN	-36.270
38536	Alpha Wave 10Hz Full Tremolo	432Hz Yoga, Binaural Reality Therapy	278076	binaural	-41.488

```
number of outliers in the B100:  0
```

id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness

In [187…

```
# energy???
# most of these look like songs

AF_current = 'energy'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} min: {AF_min:.6f}')
print('number to exclude:', sum(df_10M[AF_current] < AF_min))
display(df_10M[(df_10M[AF_current] < AF_min)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])

# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] < AF_min)].in_B100))
display(df_10M[((df_10M[AF_current] < AF_min)) & (df_10M.in_B100)].head())
```

```
energy min: 0.004660
number to exclude: 47130
```

	song	artist	duration_ms	genre	energy
7569862	Calm Sleep	Sound Library XL	101358	NaN	0.002
3778036	Sonata: II. Adagio Molto	Jacob Bogaart	207387	NaN	0.005
3625692	Stravinsky: Suite After Themes, Fragments and ...	Igor Stravinsky	189107	classical	0.004
302739	Clairs de lune - Clair de Lune: No. 3, Le Cime...	Aline Piboule	319200	NaN	0.001
3167198	Finding It There	Goldmund	220187	compositional ambient	0.003
1141698	Pas de Deux: VI. Andante	Bridget Carey	175533	classical piano	0.004
2223754	The Andree Expedition: No. 13. Final Words	Dominick Argento	97973	contemporary jazz	0.002
4284515	Via Crucis, S53/R534: Station V. Cyrenei Simon...	János Dobra	152573	classical	0.003
5090637	White Noise for Baby Sleep	Lullaby Lullaby	1919975	lullaby	0.000
1580181	Préludes, Book 1, L. 117: No. 6, Des pas sur l...	Claude Debussy	264947	<NA>	0.001

```
number of outliers in the B100:  0
```

id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness

In [190…

```
# EXCLUDE danceability < 0.064400
# most aren't songs

AF_current = 'danceability'
AF_min, AF_max = df_99.loc[AF_current]

print(f'{AF_current} min: {AF_min:.6f}')
print('number to exclude:', sum(df_10M[AF_current] < AF_min))
display(df_10M[(df_10M[AF_current] < AF_min)].sample(10)[['song', 'artist', 'duration_ms', 'genre', AF_current]])
```

```
# how many are in the B100
print('number of outliers in the B100: ', sum(df_10M[(df_10M[AF_current] < AF_min)].in_B100))
display(df_10M[((df_10M[AF_current] < AF_min)) & (df_10M.in_B100)].head())
```

```
danceability min: 0.064400
number to exclude: 46952
```

	song	artist	duration_ms	genre	danceability
7455615	Deep White Noise Drone	Sleep Baby White Noise	196000	white noise	0.000
6210097	White Noise: Microwave	Outside Broadcast Recordings	226064	sleep	0.000
7567410	Aircraft Dehavilland Tigermoth 1943 Flying01 I...	Sound Effects Master Collection	99091	sound effects	0.000
7566686	Medium Fireworks Bangs	Sound Effects Library	16382	sleep	0.000
8824332	Instrumental	Unidad de música zen relajante	229994	zen	0.061
5495095	Royal Spheres Intro	MentPlus	14333	<NA>	0.000
8692870	Trae Speaks 5	Trae Tha Truth	12159	rap	0.000
5180047	Red and White Plum Blossom	Maher Shalal Hash Baz	22400	japanese experimental	0.000
7459274	Radio Static - Loopable with No Fade	Sleeping Deeply	72145	sleep	0.000
3532316	A High Ashen Breeze - Part 2	Horseback	400347	<NA>	0.056

```
number of outliers in the B100:  1
```

	id	song	artist	genre	release_date	acousticness	danceability	duration_ms	energy	instrumentalness	key
5015786	63kd4m3VFxcJjPVVtbVNAu	Hello, Dolly!	Louis Armstrong And The All Stars	adult standards	1964-10-25	0.842	0.000	147000	0.405	0.001	0

Attachment 4

Imports

In [68]:
```
# math and dataframes
import pandas as pd
import numpy as np

# machine learning
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.model_selection import GridSearchCV

# jupyter notebook full-width display
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

# pandas formatting
pd.set_option('display.float_format', '{:.3f}'.format)
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 200)

# plotting
import matplotlib.pyplot as plt
import matplotlib as mpl
import time
import seaborn as sns
sns.set_theme()
```

In [2]:
```
# import data
# all songs with audio features
df_10M = pd.read_pickle('df_10M.pickle')
```

PIPELINE OUTLINE

- remove outliers
  - sets that aren't music
  - by inspection using songs less than 1min or more than 10min (+ more extreme)
- OPTIONAL: encode
  - alt: drop when clustering
- transform data using min max scaler
- cluster based on AF (not genre)
  - optimise based on silhouette
- examine clusters by inspecting genres and popularity

ADD GENRE FEATURES

In [3]:
```
# approximating genre categories
genre_dict = {
    'is_Adult_Standard': r'(?:adult standards|mellow gold|soft rock|contemporary)',
    'is_Rock': r'(?:rock|^country|^soft)',
    'is_R&B':  r'(?:soul|rhythm and blues|motown|funk|disco|quiet storm)',  # this one is a stretch
    'is_Country': r'(?:country|folk|nashville sound)',
    'is_Pop': r'\b(?:pop|europop)\b',  # whole word only
    'is_Rap': r'\b(?:rap|trap|hip hop)\b',  # whole word only
    'is_Alternative': r'(?:alternative)',  # not sure if grunge or punk should be here
    'is_EDM': r'(?:house|trance|techno|drum and bass|dubstep|synth|edm|breakbeat)', # not very popular on Bilboard
    'is_Metal': r'(?:heavy|metal|thrash|djent|deathcore|mathcore|grindcore)' # not very popular on Bilboard
}

for genre in genre_dict:
    df_10M[genre] = df_10M.genre.str.contains(genre_dict[genre]).fillna(False)
```

OUTLIERS

In [4]:
```
%%time
# REMOVE OUTLIERS
# based on details in outlier analysis

# add cluster column to df_10M and add 'outliers' as a cluster
df_10M['cluster'] = pd.NA

# extrema to exclude (domain knowledge + inspection of percentiles)
dur_min = df_10M.duration_ms < 60_000
dur_max = df_10M.duration_ms > 600_000
val_0 = df_10M.valence == 0
sp_min = df_10M.speechiness > 0.947000
tempo_0 = df_10M.tempo == 0
loud_min = df_10M.loudness < -34.668999
dance_min = df_10M.danceability < 0.064400

df_10M.loc[(dur_min | dur_max | val_0 | sp_min | tempo_0 | loud_min | dance_min), 'cluster'] = 'outlier'

# genres that aren't music
genres_to_exclude = set([
    'sleep', 'football', 'halloween', 'birthday', 'lullaby', 'ringtone', 'fan chant',
    'sound effects', 'spoken word', 'bible', 'prank', 'wrestling', 'language', 'oratory',
    'erotica', 'tone', 'vintage radio show', 'sound', 'quran', 'islamic recitation',
    'reading', 'asmr', 'mindfulness', 'meditation', 'guided meditation', 'workout product',
    'theme', 'environmental', 'motivation'
])

# add 2 clustering columns (optimal and generalised genres)
df_10M.loc[df_10M.genre.isin(genres_to_exclude), 'cluster'] = 'outlier'
df_10M['cluster2'] = df_10M['cluster']  # same as cluster
```

```
Wall time: 907 ms
```

CREATE FEATURE DATAFRAME FOR CLUSTERING

In [5]:
```
# create a feature set to cluster on
X = df_10M[df_10M.cluster != 'outlier'].reset_index(drop=True)
```

In [6]:
```
# 6% of data dropped as outliers
X.shape[0], (1 - X.shape[0] / df_10M.shape[0])*100
```

Out[6]:
```
(8827719, 6.404280009393792)
```

DROP AND ENCODE COLUMNS

In [7]:
```
# encode and/or drop columns (JUST DROP FOR NOW)
drop_columns = [
    'song', 'artist', 'genre', 'release_date', 'cluster', 'cluster2', 'in_B100',
    'is_Adult_Standard', 'is_Rock', 'is_R&B', 'is_Country', 'is_Pop', 'is_Rap', 'is_Alternative', 'is_EDM', 'is_Metal'
]
encode_columns = ['key', 'mode', 'time_signature']
X_not = X.set_index('id')[drop_columns + encode_columns]
X = X.drop(drop_columns, axis=1).drop(encode_columns, axis=1).set_index('id')
```

TRANSFORM DATA

In [8]:
```
## transform data to range from 0 to 1
attributes_to_transform = ['duration_ms', 'loudness', 'tempo']

for attribute in attributes_to_transform:
    X[attribute] = (X[attribute] - X[attribute].min()) / (X[attribute].max() - X[attribute].min())
```

In [9]:
```
X_not.head()
```

Out[9]:

id	song	artist	genre	release_date	cluster	cluster2	in_B100	is_Adult_Standard	is_Rock	is_R&B	is_Country	is_Pop
6SFc2WVQmARn6NDS3LrTR8	I Feel So Free - Lost Souls of Saturn Remix	!!!	dance rock	2016-02-23	<NA>	<NA>	False	False	True	False	False	False
6NMSTM4UQMC5emaYKIueyc	The Most Certain Sure (Liv Spencer Remix)	!!!	dance rock	2010-08-16	<NA>	<NA>	False	False	True	False	False	False
1FqHPzJuRdupHbcw09tYUa	Except Death	!!!	dance rock	2013-01-01	<NA>	<NA>	False	False	True	False	False	False
7y8aVfDkqt6qirGNivvs0M	One Girl / One Boy	!!!	dance rock	2013-01-01	<NA>	<NA>	False	False	True	False	False	False
70w8loBbdl4qZOH2brrqKF	When the Going Gets Tough, the Tough Get Karazzee	!!!	dance rock	2004-06-08	<NA>	<NA>	False	False	True	False	False	False

In [10]:

```
X.head()
```

Out[10]:

id	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	speechiness	tempo	valence
6SFc2WVQmARn6NDS3LrTR8	0.007	0.728	0.817	0.747	0.434	0.113	0.573	0.084	0.436	0.284
6NMSTM4UQMC5emaYKIueyc	0.027	0.703	0.736	0.763	0.847	0.103	0.619	0.051	0.418	0.336
1FqHPzJuRdupHbcw09tYUa	0.010	0.812	0.402	0.776	0.071	0.074	0.670	0.039	0.400	0.912
7y8aVfDkqt6qirGNivvs0M	0.003	0.702	0.340	0.851	0.000	0.322	0.706	0.041	0.395	0.870
70w8loBbdl4qZOH2brrqKF	0.000	0.786	0.587	0.751	0.446	0.053	0.664	0.078	0.418	0.651

CLUSTER DATA

Choose k: Silhouette Method

In [26]:

```
%%time
# try with 50k samples

# setup
n_clusters = [3, 4, 5, 6, 8, 10, 15, 20]
sample_size=50_000
results = {}

# loop
for n in n_clusters:
    kmeans = KMeans(n).fit(X)
    score = silhouette_score(X, kmeans.labels_, sample_size=sample_size)
    results[n] = score

# display results
results
```

```
Wall time: 13min 27s
```

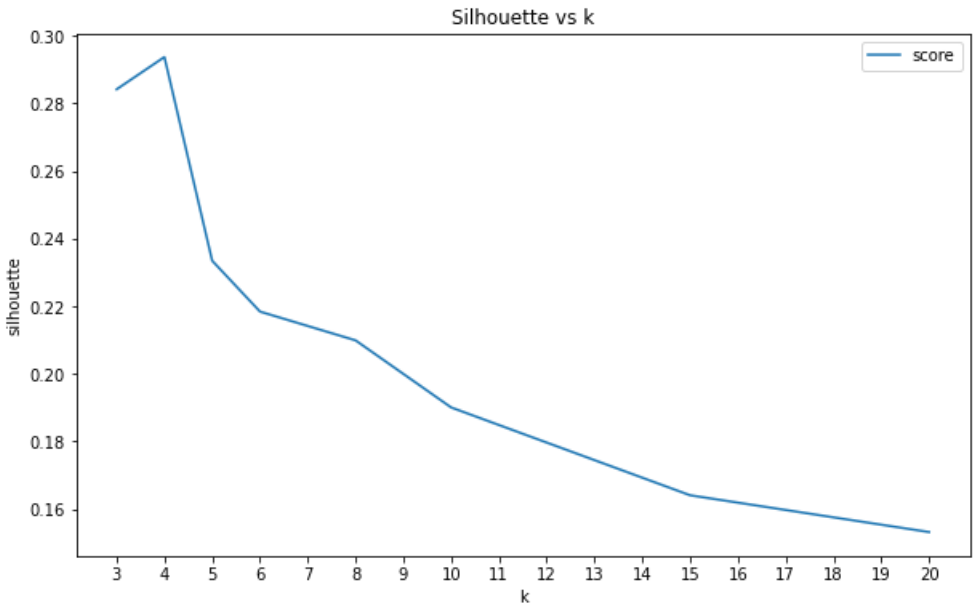
Out[26]:
```
{3: 0.2821724751018356,
 4: 0.2939383278460177,
 5: 0.23304552754496116,
 6: 0.2176436030028089,
 8: 0.2096411079563615,
 10: 0.20141793272826253,
 15: 0.16660631033595052,
 20: 0.14102224497793467}
```

In [27]:
```
%%time
# try with 200k samples
# same general results, no need for larger samples

# setup
n_clusters = [3, 4, 5, 6, 8, 10, 15, 20]
sample_size = 200_000
results = {}

# loop
for n in n_clusters:
    kmeans = KMeans(n).fit(X)
    score = silhouette_score(X, kmeans.labels_, sample_size=sample_size)
    results[n] = score

# display results
results
```

```
Wall time: 1h 1min 21s
```

Out[27]:
```
{3: 0.28418687162139655,
 4: 0.2937472857044909,
 5: 0.23345142491189227,
 6: 0.21839763822682923,
 8: 0.20987825401326532,
 10: 0.19002569873477773,
 15: 0.1640299274836951,
 20: 0.15314680377584255}
```

In [80]:
```
df_silhouette_results = pd.DataFrame(results.items(), columns=['k', 'score'])
df_silhouette_results.plot(x='k', y='score',figsize=(10, 6), title='Silhouette vs k', ylabel='silhouette');
plt.xticks(range(3, 21))
plt.show()
```



In [ ]:
```
# looks like very similar results
# k = 4 is optimal using the silhouette method
```

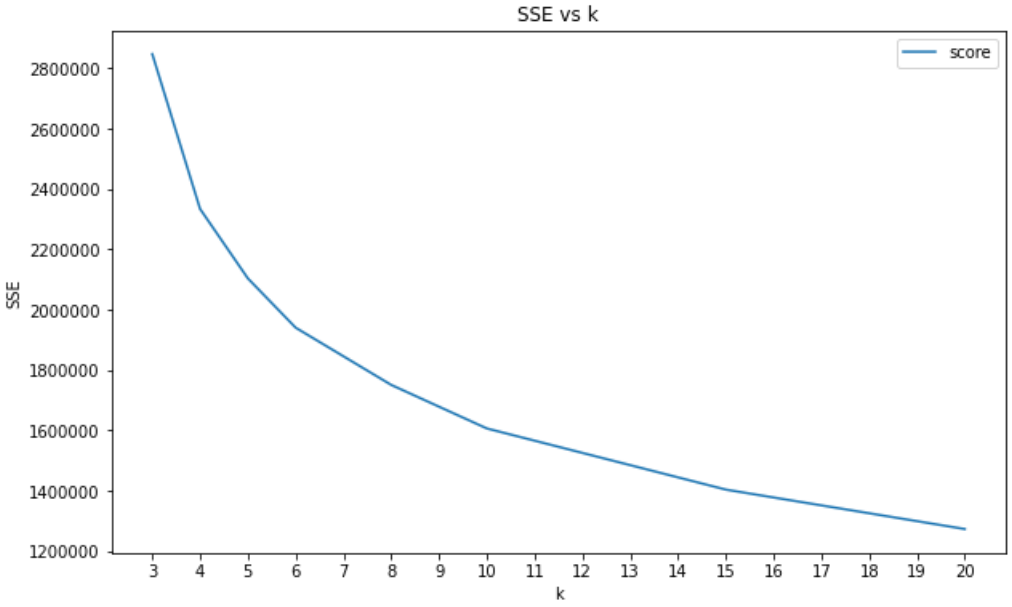
Choose k: Elbow Method

In [30]:
```
%%time
# time per n:
```

```
# setup
n_clusters = [3, 4, 5, 6, 8, 10, 15, 20]
elbow_results = {}

# loop
for n in n_clusters:
    kmeans = KMeans(n).fit(X)
    score = kmeans.score(X)
    elbow_results[n] = score

# display results
elbow_results
```

Out[30]:
```
Wall time: 10min 12s
{3: -2846513.221138529,
 4: -2333667.7577719083,
 5: -2103607.005861436,
 6: -1940477.7070383092,
 8: -1750942.2205866904,
 10: -1606817.5040655455,
 15: -1404414.0579733923,
 20: -1273836.771174847}
```

In [79]:
```
# score is negative kmeans.inertia_ ~ within cluster sum of squared error
# no obvious elbow

df_elbow_results = pd.DataFrame(elbow_results.items(), columns=['k', 'score'])
df_elbow_results['score'] = df_elbow_results['score'].abs()

plot = df_elbow_results.plot(x='k', y='score',figsize=(10, 6), title='SSE vs k', ylabel='SSE')
plot.get_yaxis().get_major_formatter().set_scientific(False)
plt.xticks(range(3, 21))
plt.show()
```



DO THE CLUSTERING

- optimal clusters looks like k=4
- also try the genres from the EDA (9 genres plus 1 for misc)

In [19]:
```
%%time
# optimal clustering
cluster = KMeans(n_clusters=4)
X['cluster'] = cluster.fit_predict(X)

# genre clustering
cluster2 = KMeans(n_clusters=10)
X['cluster2'] = cluster2.fit_predict(X)
```

```
Wall time: 1min 36s
```

In [20]:
```
X.sample(10)
```

Out[20]:

id	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	speechiness	tempo	valence	clus
5bxs4v4cLhiyTJhPb6jVhs	0.424	0.513	0.312	0.314	0.132	0.084	0.404	0.034	0.295	0.206	
080Bka8mBk5aY7vunW8cpd	0.964	0.503	0.181	0.157	0.588	0.115	0.422	0.035	0.323	0.064	
0OvscMNHLH3fzmgYC5X3W3	0.956	0.509	0.122	0.234	0.000	0.207	0.508	0.031	0.372	0.314	
0zxFTLHYX4UtdCGgaW8jDO	0.892	0.373	0.303	0.329	0.000	0.100	0.553	0.046	0.517	0.180	
1fDrUXI3bOI3qTerSJUMHg	0.002	0.484	0.280	0.938	0.000	0.137	0.718	0.040	0.300	0.484	
64ltMGayNCTrlZLHGwzwgs	0.019	0.745	0.311	0.667	0.000	0.130	0.693	0.034	0.364	0.576	
5MqUHcFbWIXm9tFDd5s3Ej	0.008	0.463	0.267	0.901	0.006	0.724	0.760	0.062	0.281	0.800	
2R2u4ERFXWoUzNVRqesEYJ	0.267	0.734	0.360	0.536	0.001	0.126	0.561	0.103	0.264	0.732	
1q1yOFIeQ8wSnxGu6sa5AY	0.017	0.862	0.354	0.602	0.560	0.288	0.586	0.062	0.460	0.965	
57X3x75MsrFo2vldEp1Ftn	0.871	0.373	0.286	0.390	0.000	0.318	0.604	0.043	0.146	0.496	

Examine Clustering Performance by inspecting genres and popularity

In [32]:
```
X_clustered = pd.concat([X_not.drop(['cluster', 'cluster2'], axis=1), X], axis=1)
```

In [45]:
```
# backup files if you want to
if False:
    X.to_pickle('X.pickle')
    X_not.to_pickle('X_not.pickle')
    X_clustered.to_pickle('X_clustered.pickle')
```

In [ ]:
```
# load files if you want to
if False:
    X = pd.read_pickle('X.pickle')
    X_not = pd.read_pickle('X_not.pickle')
    X_clustered = pd.read_pickle('X_clustered.pickle')
```

In [102…
```
X_clustered.groupby('cluster').sum()
```

Out[102…

cluster	in_B100	is_Adult_Standard	is_Rock	is_R&B	is_Country	is_Pop	is_Rap	is_Alternative	is_EDM	is_Metal	key	mode	time_signature
0	184	35338	24487	4505	19984	14314	1991	2506	7232	7765	6171635	821850	4605047
1	14659	43569	407871	76953	99112	348364	361542	59196	82412	140407	21227957	2568712	15672827
2	5844	124268	148837	31884	138919	148211	36907	11665	4991	5432	11826385	1666739	8683050
3	542	5685	75982	23236	12201	29237	15558	11583	154962	91795	7088783	778920	5149056

In [113…
```
cluster1_results = pd.concat([
    X_clustered.groupby('cluster').sum().iloc[:, 0:10],
    X_clustered.groupby('cluster').count()['song']
], axis=1).rename({'song': 'TOTAL'}, axis=1)
```

In [114...]

```
cluster2_results = pd.concat([
    X_clustered.groupby('cluster2').sum().iloc[:, 0:10],
    X_clustered.groupby('cluster2').count()['song']
], axis=1).rename({'song': 'TOTAL'}, axis=1)
```

In [115...]

```
cluster1_results
```

Out[115...]

cluster	in_B100	is_Adult_Standard	is_Rock	is_R&B	is_Country	is_Pop	is_Rap	is_Alternative	is_EDM	is_Metal	TOTAL
0	184	35338	24487	4505	19984	14314	1991	2506	7232	7765	1233380
1	14659	43569	407871	76953	99112	348364	361542	59196	82412	140407	3976534
2	5844	124268	148837	31884	138919	148211	36907	11665	4991	5432	2303244
3	542	5685	75982	23236	12201	29237	15558	11583	154962	91795	1310784

In [116...]

```
cluster2_results
```

Out[116...]

cluster2	in_B100	is_Adult_Standard	is_Rock	is_R&B	is_Country	is_Pop	is_Rap	is_Alternative	is_EDM	is_Metal	TOTAL
0	2156	3502	138291	5268	14065	71354	45838	27346	33953	119273	995322
1	3443	55490	75274	19743	68313	65695	26940	4820	1920	1093	1086178
2	56	19062	14701	1875	9423	8520	885	1624	5017	6288	819320
3	101	1774	47714	7398	5901	14832	6917	7185	101961	85116	755177
4	128	16276	9786	2630	10561	5794	1106	882	2215	1477	414060
5	3772	11300	78162	16950	27730	86395	136348	11417	19342	7748	933671
6	883	4862	45819	5339	8574	24897	32163	5199	8062	8091	317740
7	7848	23905	145599	49396	48743	165718	147193	15234	21055	5295	1729801
8	2401	68778	73563	12141	70606	82516	9967	6845	3071	4339	1217066
9	441	3911	28268	15838	6300	14405	8641	4398	53001	6679	555607

In [122...]

```
# examine top 3 genres
def top_genres(dataframe, top_n=3):
    dataframe = dataframe.drop('TOTAL', axis=1)
    for i in range(dataframe.shape[0]):
        print('cluster', i)
        print(dataframe.iloc[i].sort_values(ascending=False).head(top_n))
        print()
```

In [123...]

```
top_genres(cluster1_results)
```

```
cluster 0
is_Adult_Standard    35338
is_Rock              24487
is_Country           19984
Name: 0, dtype: int64

cluster 1
is_Rock    407871
is_Rap     361542
is_Pop     348364
Name: 1, dtype: int64

cluster 2
is_Rock       148837
is_Pop        148211
is_Country    138919
Name: 2, dtype: int64
```

```
cluster 3
is_EDM      154962
is_Metal     91795
is_Rock      75982
Name: 3, dtype: int64
```

In [124…

```
top_genres(cluster2_results)
```

```
cluster 0
is_Rock     138291
is_Metal    119273
is_Pop       71354
Name: 0, dtype: int64

cluster 1
is_Rock       75274
is_Country    68313
is_Pop        65695
Name: 1, dtype: int64

cluster 2
is_Adult_Standard    19062
is_Rock              14701
is_Country            9423
Name: 2, dtype: int64

cluster 3
is_EDM      101961
is_Metal     85116
is_Rock      47714
Name: 3, dtype: int64

cluster 4
is_Adult_Standard    16276
is_Country           10561
is_Rock               9786
Name: 4, dtype: int64

cluster 5
is_Rap     136348
is_Pop      86395
is_Rock     78162
Name: 5, dtype: int64

cluster 6
is_Rock    45819
is_Rap     32163
is_Pop     24897
Name: 6, dtype: int64

cluster 7
is_Pop     165718
is_Rap     147193
is_Rock    145599
Name: 7, dtype: int64

cluster 8
is_Pop        82516
is_Rock       73563
is_Country    70606
Name: 8, dtype: int64

cluster 9
is_EDM     53001
is_Rock    28268
is_R&B     15838
Name: 9, dtype: int64
```

Check results using proportions

- could be percentage of cluster
- or percentage of genre

In [125…

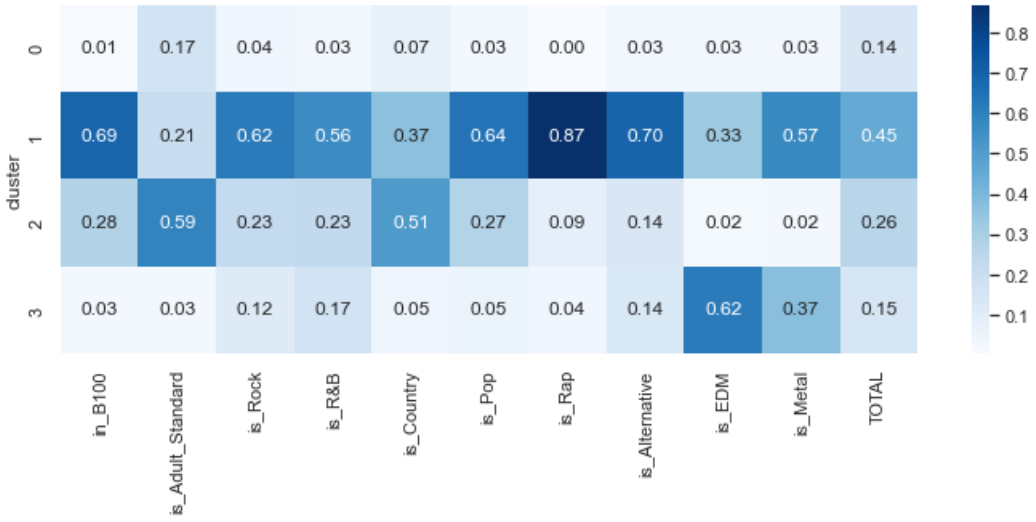
```
# proportions of columns/features
cluster1_prop = cluster1_results / cluster1_results.sum()
cluster2_prop = cluster2_results / cluster2_results.sum()
```

Viz 1: proportion of feature counts

In [139…

```
print()
print('Proportion of Total Feature Count - 4 Clusters')
plt.subplots(figsize=(12, 4))
sns.heatmap(cluster1_prop, cmap='Blues', annot=True, fmt='.2f')
plt.show()
```

```
Proportion of Total Feature Count - 4 Clusters
```



In [140…

```
print()
print('Proportion of Total Feature Count - 10 Clusters')
plt.subplots(figsize=(12, 9))
sns.heatmap(cluster2_prop, cmap='Blues', annot=True, fmt='.2f')
plt.show()
```

```
Proportion of Total Feature Count - 10 Clusters
```

Viz 2: Proportion of Cluster Counts

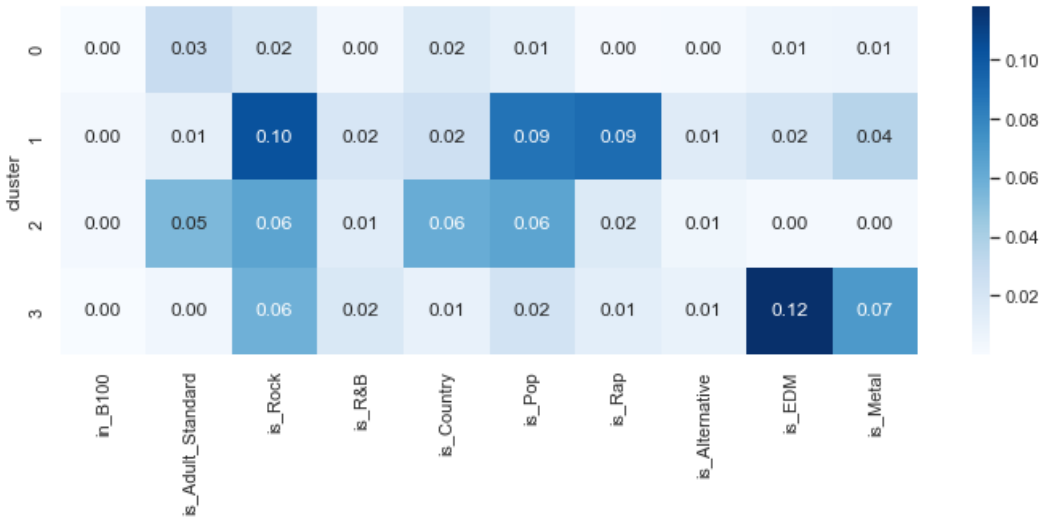
In [134…

```
# proportions of clusters
cluster1_prop2 = cluster1_results.divide(cluster1_results['TOTAL'], axis=0).drop('TOTAL', axis=1)
cluster2_prop2 = cluster2_results.divide(cluster2_results['TOTAL'], axis=0).drop('TOTAL', axis=1)
```

In [141…

```
print()
print('Proportion of Cluster - 4 Clusters')
plt.subplots(figsize=(12, 4))
sns.heatmap(cluster1_prop2, cmap='Blues', annot=True, fmt='.2f')
plt.show()
```

```
Proportion of Cluster - 4 Clusters
```



In [142…

```
print()
print('Proportion of Cluster - 10 Clusters')
plt.subplots(figsize=(12, 9))
sns.heatmap(cluster2_prop2, cmap='Blues', annot=True, fmt='.2f')
plt.show()
```

```
Proportion of Cluster - 10 Clusters
```

recombine with df_10M

In [199…

```
# convert clusters to categorical datatype
X_clustered['cluster'] = X_clustered['cluster'].astype('category')
X_clustered['cluster2'] = X_clustered['cluster2'].astype('category')
```

In [217…

```
df_10M = df_10M.drop(['cluster', 'cluster2'], axis=1).merge(X_clustered[['cluster', 'cluster2']], left_on='id', right_index=Tr

# add an outlier category for formerly excluded clusters
df_10M['cluster'] = df_10M['cluster'].cat.add_categories('outlier')
df_10M['cluster2'] = df_10M['cluster2'].cat.add_categories('outlier')

df_10M.loc[(df_10M.cluster.isna()), 'cluster'] = 'outlier'
df_10M.loc[(df_10M.cluster2.isna()), 'cluster2'] = 'outlier'
```

In [220…

```
# save
df_10M.to_pickle('df_10M_clustered.pickle')
```

In [ ]:

Attachment 5

Import

In [1]:
```
import pickle

# math and dataframes
import pandas as pd
import numpy as np

# machine learning
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier

# Pipeline and Evaluation
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import GridSearchCV, cross_val_score, StratifiedKFold
from imblearn.pipeline import make_pipeline
from imblearn.under_sampling import RandomUnderSampler

# jupyter notebook full-width display
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

# pandas formatting
pd.set_option('display.float_format', '{:.3f}'.format)
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 200)

# plotting
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
sns.set_theme()
```

In [2]:
```
df_10M = pd.read_pickle('df_10M_clustered.pickle')
X_all = pd.read_pickle('X_clustered.pickle')
X_all = X_all.reset_index()
```

Setup inputs for statistical scenarios

In [3]:
```
# columns for datasets

y_column = 'in_B100'
X_columns = [
    'mode', 'acousticness', 'danceability', 'duration_ms', 'energy',
    'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo', 'valence'
]
genre_columns = [
    'is_Adult_Standard', 'is_Rock', 'is_R&B', 'is_Country', 'is_Pop',
    'is_Rap', 'is_Alternative', 'is_EDM', 'is_Metal'
]
cluster_columns = ['cluster', 'cluster2']
other_columns = ['key', 'time_signature', 'genre', 'release_date']
```

In [4]:
```
# hyperparameters

param_by_model = {}

params_lr = {}
orders_of_magnitude = []
for lst in [[int(x)/10000 for x in range(1, 11)],
            [int(x)/1000 for x in range(1, 11)],
            [int(x)/100 for x in range(1, 11)],
            [int(x)/10 for x in range(1, 11)],
            [1 * x for x in range(1, 11)],
            [10 * x for x in range(1, 11)],
```

```
            [100 * x for x in range(1, 11)],
            [1000 * x for x in range(1, 11)]]:
    orders_of_magnitude += lst
params_lr['logisticregression__penalty'] = ['l1', 'l2']
params_lr['logisticregression__C'] = orders_of_magnitude
params_lr['logisticregression__solver'] = ['liblinear']
param_by_model[0] = params_lr

params_dt = {}
params_dt['decisiontreeclassifier__max_depth'] = [3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50, 100, None]
params_dt['decisiontreeclassifier__min_samples_leaf'] = [5, 10, 50, 100, 1000]
params_dt['decisiontreeclassifier__criterion'] = ['gini', 'entropy']
param_by_model[1] = params_dt

params_knn = {}
params_knn['kneighborsclassifier__n_neighbors'] = [x for x in range(2,20)]+[x for x in range(20,101,5)]
params_knn['kneighborsclassifier__weights'] = ['uniform', 'distance']
params_knn['kneighborsclassifier__metric'] = ['minkowski', 'euclidean', 'manhattan']
param_by_model[2] = params_knn

params_rf = {}
params_rf['randomforestclassifier__n_estimators'] = [5, 10, 20, 50, 100, 200, 500, 1000, 2000]
params_rf['randomforestclassifier__max_features'] = ['sqrt', 'log2']
params_rf['randomforestclassifier__max_depth'] = [3, 5, 7, 10, 15, 20, 30, 50, 100, None]
params_rf['randomforestclassifier__min_samples_leaf'] = [5, 10, 50, 100, 1000]
params_rf['randomforestclassifier__bootstrap'] = [True, False]
param_by_model[3] = params_rf

params_ab = {}
params_ab['adaboostclassifier__n_estimators'] = [10, 50, 100, 200, 500, 1000, 2000, 5000, 10000]
params_ab['adaboostclassifier__learning_rate'] = [0.0001, 0.001, 0.01, 0.05, 0.1, 0.5, 1.0, 1.5, 2.0]
params_ab['adaboostclassifier__algorithm'] = ['SAMME', 'SAMME.R']
param_by_model[4] = params_ab

# scoring metrics

metrics = [
    'balanced_accuracy', 'average_precision', 'neg_brier_score', 'f1', 'f1_micro',
    'f1_macro', 'f1_weighted', 'neg_log_loss', 'precision', 'recall', 'roc_auc', 'jaccard'
]

# how many hyperparameter scenarios in the grid search

def how_many_scenarios(n_ML):
    n_scenarios = 1
    for key in param_by_model[n_ML].keys():
        n_scenarios *=  len(param_by_model[n_ML][key])
    return n_scenarios

for i in range(5):
    print(how_many_scenarios(i))
```

```
160
160
210
1800
162
```

Make Predictions Dataframe for Statistics

- split into 5 stratified folds
  - using a consistent random_state to use the same folds between tests
- for each fold:
  - train on undersampled training fold
  - predict on full test fold
  - add out of fold predictions to the predictions dataframe

NOTES:

- Tuning individual models on limited datasets has been investigated in NOTEBOOK 5B (and 5D).
- Random undersampling and oversampling were investigated in NOTEBOOK 5A

- More involved oversampling methods like SMOTE were not considered because the nature of music. For example, interpolating between modes leads to an atonal, non-musical result. Discrete combinations of features are likely to be important in terms of audio features as well. More importantly, with over 20k positive cases in our dataset, we should have enough data for a well trained model.

In [5]:
```
# initial setup

# use the same stratified split for all test cases
stratified_5fold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# scenarios to test
stats_scenarios = [
    'y_lr', 'y_dt', 'y_knn', 'y_rf', 'y_ab', 'y_lr_tuned', 'y_dt_tuned', 'y_knn_tuned', 'y_rf_tuned', 'y_ab_tuned', 'y_cl_1',
]
```

In [6]:
```
# initialise the dataframe
df_predictions = pd.DataFrame(columns=stats_scenarios)
df_predictions['y_actual'] = pd.NA  # for debugging
df_predictions = pd.concat([X_all, df_predictions], axis=1)
```

Predict Using Default Algorithms

In [7]:
```
### OUTDATED FUNCTION, UPDATED BELOW

def evaluate_default_performance(dataframe, kfold, feature_columns, class_column, n_scenario):

    # use a copy of the dataframe to leave the original alone
    dataframe = dataframe.copy()

    # entire dataset for predictions
    X_, y_ = dataframe[feature_columns], dataframe[class_column]

    # initialise actual y and predicted y as blank dataframes
    y_actual = pd.DataFrame()
    y_pred = pd.DataFrame()

    # loop through folds
    for train_i, test_i in kfold.split(X_, y_):

        # train test split for current fold
        train_X, test_X = X_.iloc[train_i], X_.iloc[test_i]
        train_y, test_y = y_.iloc[train_i], y_.iloc[test_i]

        # create and fit pipeline
        undersampler = RandomUnderSampler(sampling_strategy='majority', random_state=42)

        if n_scenario in [1, 6]:
            model = DecisionTreeClassifier()
        elif n_scenario in [2, 7]:
            model = KNeighborsClassifier()
        elif n_scenario in [3, 8]:
            model = RandomForestClassifier()
        elif n_scenario in [4, 9]:
            model = AdaBoostClassifier()
        else:
            model = LogisticRegression()

        pipe = make_pipeline(undersampler, model)

        pipe.fit(train_X, train_y)

        # append results
        y_pred_temp = pipe.predict(test_X)
        y_pred_temp = pd.concat([
            pd.DataFrame(y_pred_temp),
            pd.DataFrame(test_i, columns=[''])
        ], axis=1).set_index('')
        y_pred = pd.concat([y_pred, y_pred_temp], axis=0)
        y_actual = pd.concat([y_actual, test_y], axis=0)  # for debugging

    # return the full sorted results, appended into the input dataframe
    dataframe[stats_scenarios[n_scenario]] = y_pred.sort_index()
    dataframe['y_actual'] = y_actual.sort_index()  # for debugging, this should always be equal to in_B100 or folds are misali
```

```
    # DEBUGGING: should be zero
    is_ERRORS = sum(dataframe['in_B100'] != dataframe['y_actual'])
    if is_ERRORS != 0:
        print('THERE WERE ERRORS!!!! (compare the in_B100 and y_actual columns)')

    return dataframe
```

In [8]:
```
%%time
# Logistic Regression
n_scenario = 0
df_predictions = evaluate_default_performance(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)
```

```
Wall time: 27.9 s
```

In [9]:
```
%%time
# Decision Tree
n_scenario = 1
df_predictions = evaluate_default_performance(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)
```

```
Wall time: 26.7 s
```

In [10]:
```
%%time
# K Nearest Neighbours
n_scenario = 2
df_predictions = evaluate_default_performance(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)
```

```
Wall time: 17min 55s
```

In [11]:
```
%%time
# Random Forest
n_scenario = 3
df_predictions = evaluate_default_performance(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)
```

```
Wall time: 3min 8s
```

In [12]:
```
%%time
# AdaBoost
n_scenario = 4
df_predictions = evaluate_default_performance(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)
```

```
Wall time: 1min 18s
```

In [13]:
```
# save dataframe
df_predictions.to_pickle('df_predictions_DEFAULT.pickle')
```

Make Predictions With Tuned Models

In [209…
```
### OUTDATED FUNCTION, UPDATED BELOW

def append_predictions(dataframe, kfold, feature_columns, class_column, n_scenario):

    # use a copy of the dataframe to leave the original alone
    dataframe = dataframe.copy()

    # entire dataset for predictions
    X_, y_ = dataframe[feature_columns], dataframe[class_column]

    # initialise actual y and predicted y as blank dataframes
    y_actual = pd.DataFrame()
    y_pred = pd.DataFrame()

    # loop through folds
    for train_i, test_i in kfold.split(X_, y_):

        # train test split for current fold
        train_X, test_X = X_.iloc[train_i], X_.iloc[test_i]
        train_y, test_y = y_.iloc[train_i], y_.iloc[test_i]

        # create pipeline
        undersampler = RandomUnderSampler(sampling_strategy='majority', random_state=42)
```

```
        if n_scenario in [1, 6]:
            n_ML = 1
            model = DecisionTreeClassifier()
        elif n_scenario in [2, 7]:
            n_ML = 2
            model = KNeighborsClassifier()
        elif n_scenario in [3, 8]:
            n_ML = 3
            model = RandomForestClassifier()
        elif n_scenario in [4, 9]:
            n_ML = 4
            model = AdaBoostClassifier()
        else:
            n_ML = 0
            model = LogisticRegression()

        pipe = make_pipeline(undersampler, model)

        # tune hyperparameters if required
        if n_scenario in [5, 6, 7, 8, 9]:
            # create and fit gridsearch
            grid = GridSearchCV(
                pipe,
                param_grid = param_by_model[n_ML],
                scoring = 'roc_auc'
            )
            grid.fit(train_X, train_y)
            y_pred_temp = grid.predict(test_X)
        else:
            pipe.fit(train_X, train_y)
            y_pred_temp = pipe.predict(test_X)

        # append results
        y_pred_temp = pd.concat([
            pd.DataFrame(y_pred_temp),
            pd.DataFrame(test_i, columns=[''])
        ], axis=1).set_index('')
        y_pred = pd.concat([y_pred, y_pred_temp], axis=0)
        y_actual = pd.concat([y_actual, test_y], axis=0)  # for debugging

    # return the full sorted results, appended into the input dataframe
    dataframe[stats_scenarios[n_scenario]] = y_pred.sort_index()
    dataframe['y_actual'] = y_actual.sort_index()  # for debugging, this should always be equal to in_B100 or folds are misalig

    # DEBUGGING: should be zero
    is_ERRORS = sum(dataframe['in_B100'] != dataframe['y_actual'])
    if is_ERRORS != 0:
        print('THERE WERE ERRORS!!!! (compare the in_B100 and y_actual columns)')

    return dataframe
```

In [206…

```
# this is wrong, they take a lot longer (esp adaboost)

n_LogisticRegression = 160
n_DecisionTreeClassifier = 160
n_KNeighborsClassifier = 210
n_RandomForestClassifier = 1800
n_AdaBoostClassifier = 162

# KNN and Random Forest Will Take Too Long, only try
n_LogisticRegression * 23/60, n_DecisionTreeClassifier * 25/60, n_KNeighborsClassifier * 18, n_RandomForestClassifier  * 186/6
```

Out[206…

```
(61.333333333333336, 66.66666666666667, 3780, 5580.0, 213.3)
```

In [232…

```
%%time
n_scenario = 5
df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)

# save dataframe
df_predictions.to_pickle('df_predictions_TUNED.pickle')
```

```
Wall time: 2h 54min 36s
```

In [233…

```
%%time
n_scenario = 6
df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)

# save dataframe
df_predictions.to_pickle('df_predictions_TUNED.pickle')
```

```
Wall time: 2h 56min 53s
```

These are too time consuming to complete

these were investigated using smaller fitting dataset in earlier notebook

In [ ]:

```
# %%time
# # calculated this from early afternoon until the next morning, and it didn't complete
# # drop adaboost from partially tuned models
# n_scenario = 9
# df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)

# save dataframe
# df_predictions.to_pickle('df_predictions_FULLYTUNED.pickle')
```

In [ ]:

```
# %%time
# # this should take too long
# n_scenario = 7
# df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)

# save dataframe
# df_predictions.to_pickle('df_predictions_FULLYTUNED.pickle')
```

In [ ]:

```
# %%time
# # this should take too long
# n_scenario = 8
# df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario)

# save dataframe
# df_predictions.to_pickle('df_predictions_FULLYTUNED.pickle')
```

Make predictions separating into clusters and genres

In [32]:

```
### THIS IS THE FINAL APPEND PREDICITONS FUNCTION
# should be applicable to all situations, but haven't been tested for all scenarios (some take hours/days to run)
# works on scenario 0, 5, 6, 10+

def append_predictions(dataframe, kfold, feature_columns, class_column, n_scenario):
    """
    loop through folds, append out of fold predictions to database of predictions
    cluster could be 'genre', 'cluster1', 'cluster2', or False (default)
    """
    # use a copy of the dataframe to leave the original alone
    dataframe = dataframe.copy()

    # based on the scenario number, do we need to cluster?
    if n_scenario == 10:
        cluster = 'cluster1'
    elif n_scenario == 11:
        cluster = 'cluster2'
    elif n_scenario == 12:
        cluster = 'genre'
    else:
        cluster = False

    # entire dataset for predictions
    X_, y_ = dataframe[feature_columns], dataframe[class_column]

    # initialise actual y and predicted y as blank dataframes
    y_actual = pd.DataFrame()
    y_pred = pd.DataFrame()

    # loop through folds
```

```
for train_i, test_i in kfold.split(X_, y_):

    # train test split for current fold
    train_X, test_X = X_.iloc[train_i], X_.iloc[test_i]
    train_y, test_y = y_.iloc[train_i], y_.iloc[test_i]

    # create pipeline
    undersampler = RandomUnderSampler(sampling_strategy='majority', random_state=42)

    # initialise a new classifier (inside fold loop to prevent spillover from refitting)
    if n_scenario in [1, 6]:
        n_ML = 1
        model = DecisionTreeClassifier()
    elif n_scenario in [2, 7]:
        n_ML = 2
        model = KNeighborsClassifier()
    elif n_scenario in [3, 8]:
        n_ML = 3
        model = RandomForestClassifier()
    elif n_scenario in [4, 9]:
        n_ML = 4
        model = AdaBoostClassifier()
    else:
        n_ML = 0
        model = LogisticRegression()

    pipe = make_pipeline(undersampler, model)

    # THREE OPTIONS: tune hyperparameters, loop through clusters, just fit the pipe

    # OPTION 1: tune hyperparameters, tune/fit the grid
    if n_scenario in [5, 6, 7, 8, 9]:
        # create and fit gridsearch
        # NOTE: n_jobs=1 leads to a PicklingError
        """
        from: https://stackoverflow.com/questions/56884020/spacy-with-joblib-library-generates-pickle-picklingerror-could-n
            'Same issue. I solved by changing the backend from loky to threading in Parallel.'
        """
        grid = GridSearchCV(
            pipe,
            param_grid = param_by_model[n_ML],
            scoring = 'roc_auc'
        )
        grid.fit(train_X, train_y)
        y_pred_temp = grid.predict(test_X)

    # OPTION 2: loop through clusters, individually fit the pipe
    elif cluster:

        # initialise dataframe to append results
        y_pred_temp = pd.DataFrame()

        # cluster 1
        if cluster == 'cluster1':
            for i in range(4):
                # this seems convoluted, but it speeds the code 10x vs loc bool combo
                cluster_index = dataframe[['cluster']][dataframe['cluster'] == i]
                i_train_cluster = cluster_index[cluster_index.index.isin(train_i)].index
                i_test_cluster = cluster_index[cluster_index.index.isin(test_i)].index

                # iloc doesn't work on index, it works on position, loc works
                train_X_c = train_X.loc[i_train_cluster]
                test_X_c = test_X.loc[i_test_cluster]
                train_y_c = train_y.loc[i_train_cluster]
                test_y_c = test_y.loc[i_test_cluster]

                # make prediction
                pipe.fit(train_X_c, train_y_c)
                y_pred_cluster = pipe.predict(test_X_c)

                # append prediction to y_pred_temp
                y_pred_temp = pd.concat([
                    y_pred_temp,
                    pd.DataFrame(y_pred_cluster, index=i_test_cluster)
                ], axis=0)

        # cluster 2
```

```
        elif cluster == 'cluster2':

            for i in range(10):
                # this seems convoluted, but it speeds the code up by a factor of 10
                cluster_index = dataframe[['cluster2']][dataframe['cluster2'] == i]
                i_train_cluster = cluster_index[cluster_index.index.isin(train_i)].index
                i_test_cluster = cluster_index[cluster_index.index.isin(test_i)].index

                # iloc doesn't work on index, it works on position, loc works
                train_X_c = train_X.loc[i_train_cluster]
                test_X_c = test_X.loc[i_test_cluster]
                train_y_c = train_y.loc[i_train_cluster]
                test_y_c = test_y.loc[i_test_cluster]

                # make prediction
                pipe.fit(train_X_c, train_y_c)
                y_pred_cluster = pipe.predict(test_X_c)

                # append prediction to y_pred_temp
                y_pred_temp = pd.concat([
                    y_pred_temp,
                    pd.DataFrame(y_pred_cluster, index=i_test_cluster)
                ], axis=0)

        # genre
        elif cluster == 'genre':
            genre_columns = [
                'is_Adult_Standard', 'is_Rock', 'is_R&B', 'is_Country', 'is_Pop',
                'is_Rap', 'is_Alternative', 'is_EDM', 'is_Metal'
            ]
            for genre in genre_columns:

                # NOTE: could consider adding a 'misc' genre, which is not in these genres

                # this seems convoluted, but it speeds the code 10x vs loc bool combo
                cluster_index = dataframe[[genre]][dataframe[genre]]  # confirm that this works
                i_train_cluster = cluster_index[cluster_index.index.isin(train_i)].index
                i_test_cluster = cluster_index[cluster_index.index.isin(test_i)].index

                # iloc doesn't work on index, it works on position, loc works
                train_X_c = train_X.loc[i_train_cluster]
                test_X_c = test_X.loc[i_test_cluster]
                train_y_c = train_y.loc[i_train_cluster]
                test_y_c = test_y.loc[i_test_cluster]

                # make prediction
                pipe.fit(train_X_c, train_y_c)
                y_pred_cluster = pipe.predict(test_X_c)

                # append prediction to y_pred_temp
                y_pred_temp = pd.concat([
                    y_pred_temp,
                    pd.DataFrame(y_pred_cluster, index=i_test_cluster)
                ], axis=0)

        # no matching cluster exists
        else:
            print('NO SUCH CLUSTER')  # could raise an error instead
            return dataframe  # do nothing, just return the input dataframe

        # sort y_pred_temp by index so it aligns properly
        y_pred_temp = np.array(y_pred_temp.sort_index())

    # OPTION 3: just fit the pipe
    else:
        pipe.fit(train_X, train_y)
        y_pred_temp = pipe.predict(test_X)

    # debugging statement: after fitting the data for the fold
    print('fold complete')

    # fitting complete for fold
    # append results
    y_pred_temp = pd.concat([
        pd.DataFrame(y_pred_temp),
        pd.DataFrame(test_i, columns=[''])
    ], axis=1).set_index('')
```

```
        y_pred = pd.concat([y_pred, y_pred_temp], axis=0)
        y_actual = pd.concat([y_actual, test_y], axis=0)  # for debugging

    # return the full sorted results, appended into the input dataframe
    dataframe[stats_scenarios[n_scenario]] = y_pred.sort_index()
    dataframe['y_actual'] = y_actual.sort_index()  # for debugging, this should always be equal to in_B100 or folds are misalig

    # DEBUGGING: should be zero
    is_ERRORS = sum(dataframe['in_B100'] != dataframe['y_actual'])
    if is_ERRORS != 0:
        print('THERE WERE ERRORS!!!! (compare the in_B100 and y_actual columns)')

    return dataframe
```

In [37]:

```
%%time
# clustering results: cluster 1
df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario=10)

# save dataframe
df_predictions.to_pickle('df_predictions_CLUSTERS.pickle')
```

```
fold complete
fold complete
fold complete
fold complete
fold complete
Wall time: 1min
```

In [38]:

```
%%time
# clustering results: cluster 2
df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario=11)

# save dataframe
df_predictions.to_pickle('df_predictions_CLUSTERS.pickle')
```

```
fold complete
fold complete
fold complete
fold complete
fold complete
Wall time: 1min 15s
```

In [39]:

```
%%time
# clustering results: genres
df_predictions = append_predictions(df_predictions, stratified_5fold, X_columns, y_column, n_scenario=12)

# save dataframe
df_predictions.to_pickle('df_predictions_CLUSTERS.pickle')
```

```
fold complete
fold complete
fold complete
fold complete
fold complete
Wall time: 52.8 s
```

Save Final Predictions Dataframe

In [40]:

```
# save final predictions dataframe
df_predictions.drop(['y_knn_tuned', 'y_rf_tuned', 'y_ab_tuned', 'y_actual'], axis=1).to_pickle('df_predictions.pickle')
```

In [ ]:

Attachment 6

Import

In [1]:
```
import pickle

# math and dataframes
import pandas as pd
import numpy as np

# statistics
from sklearn.metrics import r2_score
from scipy.stats import friedmanchisquare, wilcoxon
import scipy.stats as stats
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Pipeline and Evaluation
from sklearn.metrics import classification_report, confusion_matrix

# jupyter notebook full-width display
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))

# pandas formatting
pd.set_option('display.float_format', '{:.3f}'.format)
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 200)

# plotting
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
sns.set_theme()
```

In [2]:
```
df_predictions = pd.read_pickle('df_predictions.pickle')
```

In [3]:
```
# columns for datasets

y_column = 'in_B100'
X_columns = [
    'mode', 'acousticness', 'danceability', 'duration_ms', 'energy',
    'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo', 'valence'
]
genre_columns = [
    'is_Adult_Standard', 'is_Rock', 'is_R&B', 'is_Country', 'is_Pop',
    'is_Rap', 'is_Alternative', 'is_EDM', 'is_Metal'
]
cluster_columns = ['cluster', 'cluster2']
other_columns = ['key', 'time_signature', 'genre', 'release_date']
prediction_columns = [
    'y_lr', 'y_dt', 'y_knn', 'y_rf', 'y_ab', 'y_lr_tuned', 'y_dt_tuned',
    'y_cl_1', 'y_cl_2', 'y_genres'
]

# descriptions for scenarios
predictions_dict = {
    'y_lr': 'Logistic Regression - Default Hyperparameters',
    'y_dt': 'Decision Tree - Default Hyperparameters',
    'y_knn': 'K-Nearest Neighbours - Default Hyperparameters',
    'y_rf': 'Random Forest - Default Hyperparameters',
    'y_ab': 'AdaBoost - Default Hyperparameters',
    'y_lr_tuned': 'Logistic Regression - Tuned Hyperparameters',
    'y_dt_tuned': 'Decision Tree - Tuned Hyperparameters',
    'y_cl_1': 'Logistic Regression - Clustered By KMeans Version 1',
    'y_cl_2': 'Logistic Regression - Clustered By KMeans Version 2',
    'y_genres': 'Logistic Regression - Clustered By Genre'
}
```

Explore Predictions

In [4]:
```
df_predictions[prediction_columns].describe().T.iloc[:-1]
```

Out[4]:

	count	unique	top	freq
y_lr	8827719	2	False	5012328
y_dt	8827719	2	False	5759549
y_knn	8827719	2	False	5389246
y_rf	8827719	2	False	5972306
y_ab	8827719	2	False	5368640
y_lr_tuned	8827719	2	False	5015482
y_dt_tuned	8827719	2	False	5730894
y_cl_1	8827719	2	False	5524314
y_cl_2	8827719	2	False	5445396

In [5]:
```
# where genre data is available
df_predictions[~df_predictions.y_genres.isna()][prediction_columns].describe().T
```

Out[5]:

	count	unique	top	freq
y_lr	2808901	2	False	1632876
y_dt	2808901	2	False	1856456
y_knn	2808901	2	False	1747625
y_rf	2808901	2	False	1934264
y_ab	2808901	2	False	1746821
y_lr_tuned	2808901	2	False	1633972
y_dt_tuned	2808901	2	False	1855688
y_cl_1	2808901	2	False	1792586
y_cl_2	2808901	2	False	1769889
y_genres	2808901	2	False	1693361

In [6]:
```
# fill na values for y_genres (if not predicted yes, set to no)
df_predictions = df_predictions.fillna(False)  # for plotting genres
```

PCA Scatterplots

In [7]:
```
pca = PCA()
transformer = StandardScaler()

# transform the data
transformed_X = transformer.fit_transform(df_predictions[X_columns])

# get the PCA
pca_data = pca.fit_transform(transformed_X)
```

In [8]:
```
df_predictions = pd.concat([
    df_predictions,
    pd.DataFrame(pca_data).rename({0: 'PCA1', 1: 'PCA2'}, axis=1)[['PCA1', 'PCA2']]
], axis=1)
```

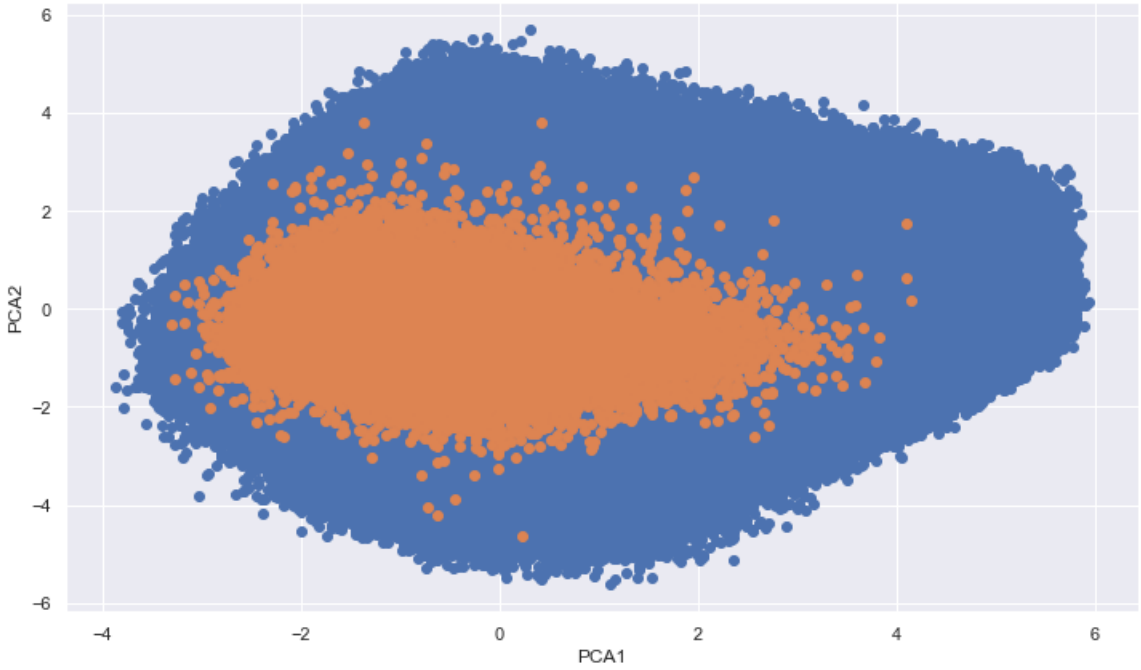
In [9]:
```
pd.DataFrame(pca_data).rename({0: 'PCA1', 1: 'PCA2'}, axis=1)[['PCA1', 'PCA2']].describe()
```

Out[9]:

	PCA1	PCA2
count	8827719.000	8827719.000
mean	-0.000	-0.000
std	1.761	1.215
min	-3.878	-5.615
25%	-1.352	-0.834
50%	-0.462	-0.072
75%	1.055	0.781
max	5.945	5.697

In [10]:

```
figsize = (12, 7)
plot_x, plot_y = 'PCA1', 'PCA2'
plt.figure(figsize=figsize)
plt.scatter(df_predictions[~df_predictions.in_B100][plot_x], df_predictions[~df_predictions.in_B100][plot_y])
plt.scatter(df_predictions[df_predictions.in_B100][plot_x], df_predictions[df_predictions.in_B100][plot_y])
plt.xlabel(plot_x)
plt.ylabel(plot_y)
plt.show()
```



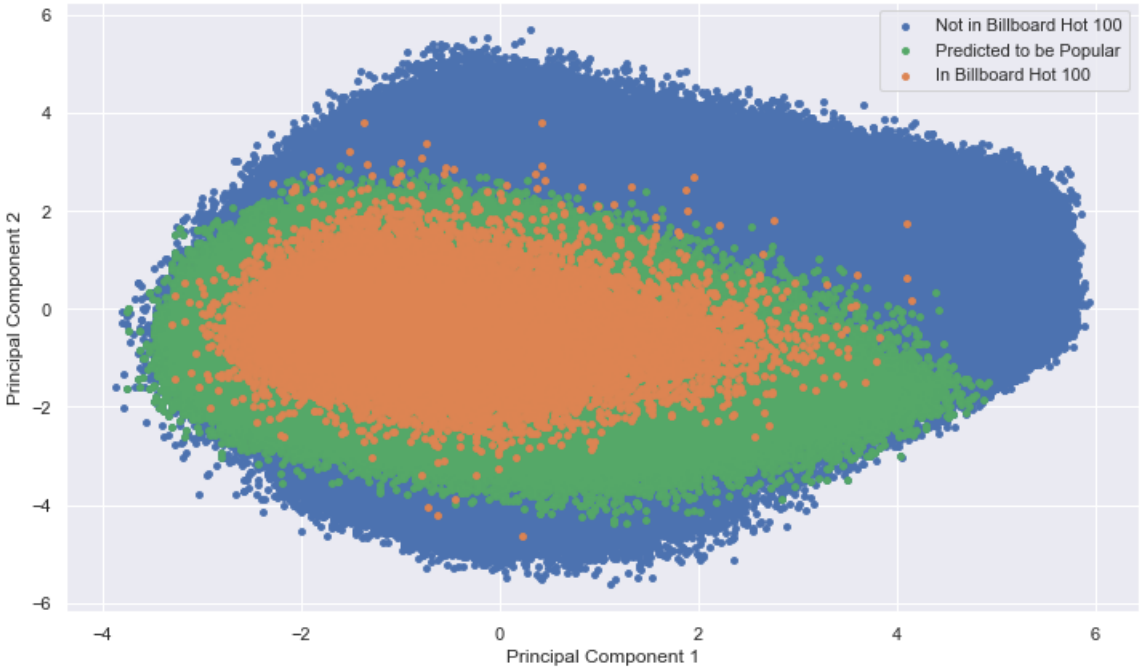
In [11]:

```
def pca_plot(n_scenario, plot_x='PCA1', plot_y='PCA2'):

    # evaluated scenarios (with numeric keys)
    scenarios_dict = {
        0: 'Logistic Regression - Default Hyperparameters',
        1: 'Decision Tree - Default Hyperparameters',
        2: 'K-Nearest Neighbours - Default Hyperparameters',
        3: 'Random Forest - Default Hyperparameters',
        4: 'AdaBoost - Default Hyperparameters',
        5: 'Logistic Regression - Tuned Hyperparameters',
        6: 'Decision Tree - Tuned Hyperparameters',
        7: 'Logistic Regression - Clustered By KMeans Version 1',
        8: 'Logistic Regression - Clustered By KMeans Version 2',
        9: 'Logistic Regression - Clustered By Genre'
    }
    print('Comparison of Predicted and Actual Billboard Hits:', scenarios_dict[n_scenario])

    figsize = (12, 7)
    markersize = 15
    alpha = 1

    # setup the plot
```

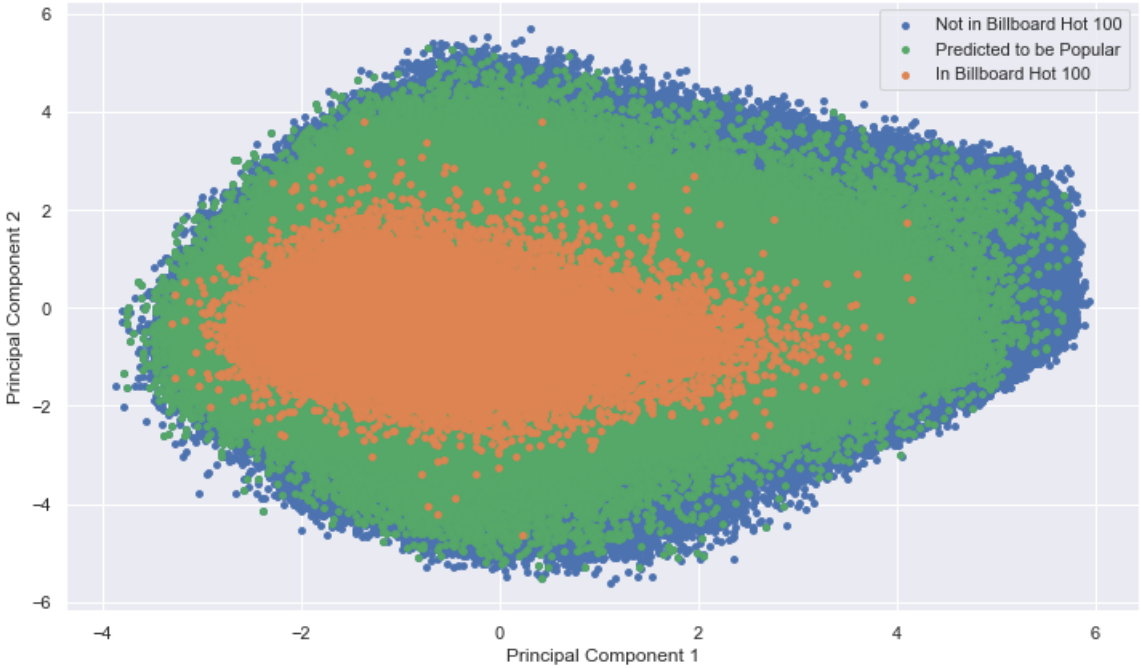
```
    plt.figure(figsize=figsize)

    # labels
    if plot_x[:3] == 'PCA':
        xlabel = 'Principal Component '+ plot_x[-1]
    else:
        xlabel = plot_x.title().replace('_', ' ')

    if plot_y[:3] == 'PCA':
        ylabel = 'Principal Component '+ plot_y[-1]
    else:
        ylabel = plot_y.title().replace('_', ' ')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    # actually no
    plt.scatter(
        df_predictions[~df_predictions.in_B100][plot_x],
        df_predictions[~df_predictions.in_B100][plot_y],
        label='Not in Billboard Hot 100',
        s=markersize,
        alpha=alpha
    )

    # predicted yes
    plt.scatter(
        df_predictions[df_predictions[prediction_columns[n_scenario]]][plot_x],
        df_predictions[df_predictions[prediction_columns[n_scenario]]][plot_y],
        color=sns.color_palette()[2],
        label='Predicted to be Popular',
        s=markersize,
        alpha=alpha
    )

    # actually yes
    plt.scatter(
        df_predictions[df_predictions.in_B100][plot_x],
        df_predictions[df_predictions.in_B100][plot_y],
        color=sns.color_palette()[1],
        label='In Billboard Hot 100',
        s=markersize,
        alpha=alpha
    )

    # legend
    plt.legend(loc='upper right')

    plt.show()
```

In [ ]:

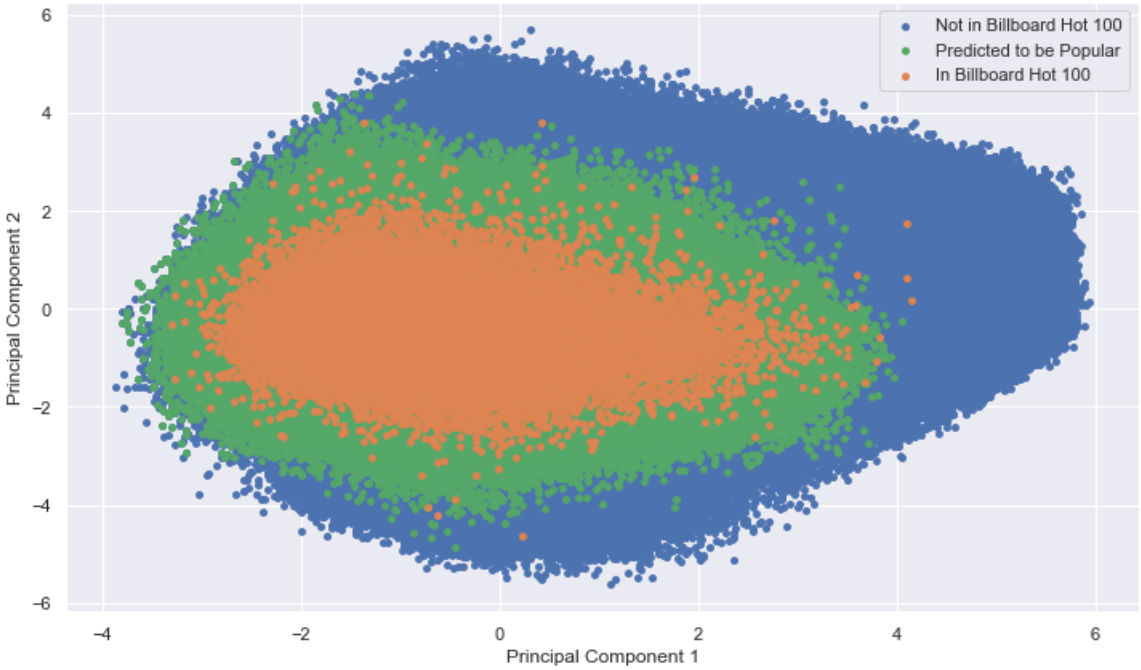
```
%%time
for i in range(10):
    pca_plot(i)
```



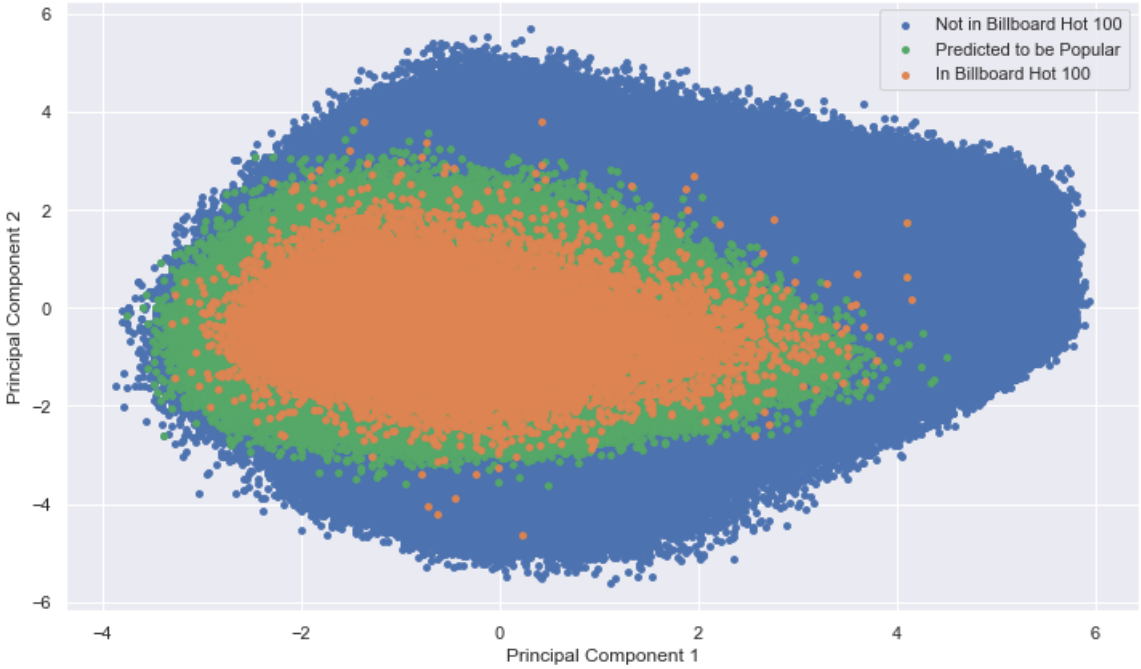
Comparison of Predicted and Actual Billboard Hits: Decision Tree - Default Hyperparameters



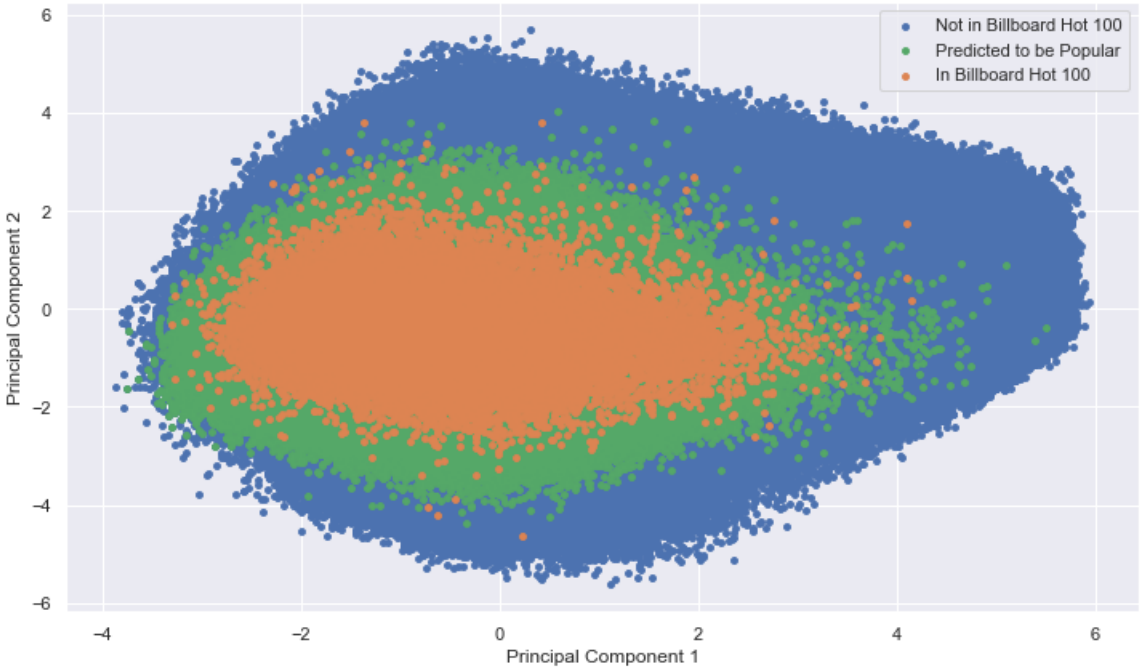
Comparison of Predicted and Actual Billboard Hits: K-Nearest Neighbours - Default Hyperparameters



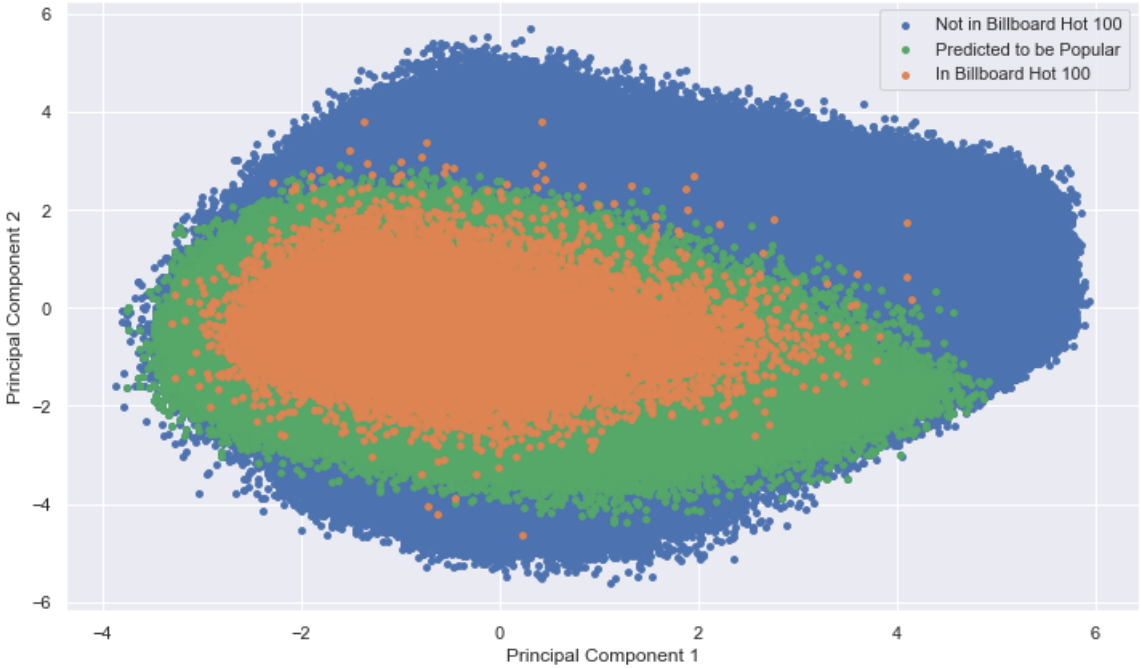
Comparison of Predicted and Actual Billboard Hits: Random Forest - Default Hyperparameters



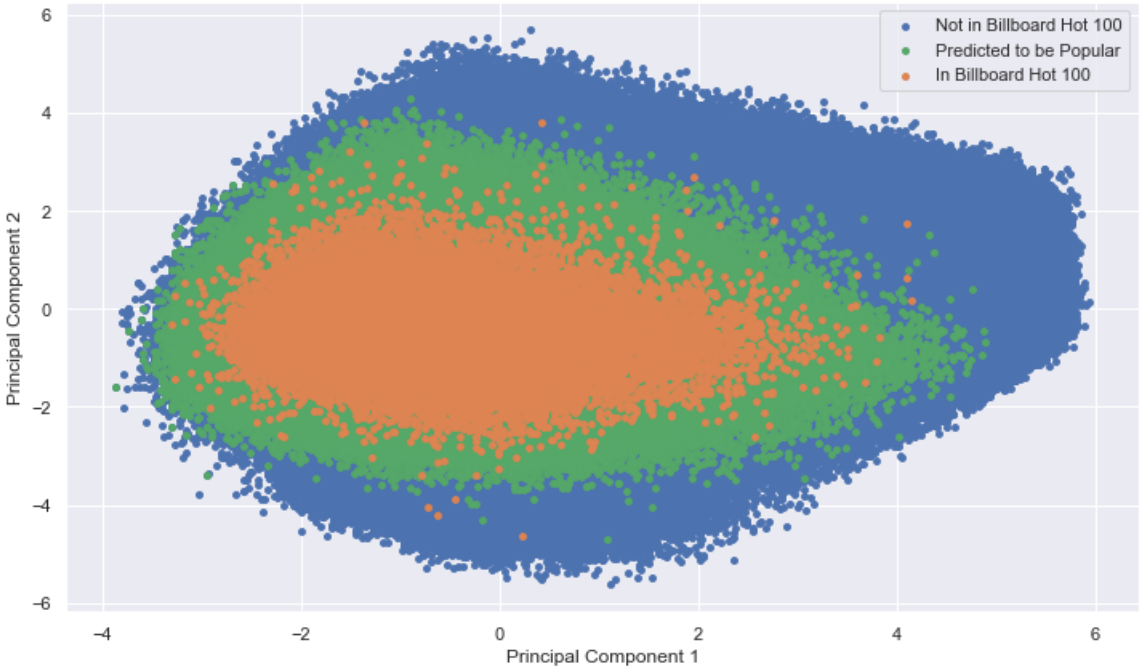
Comparison of Predicted and Actual Billboard Hits: AdaBoost - Default Hyperparameters



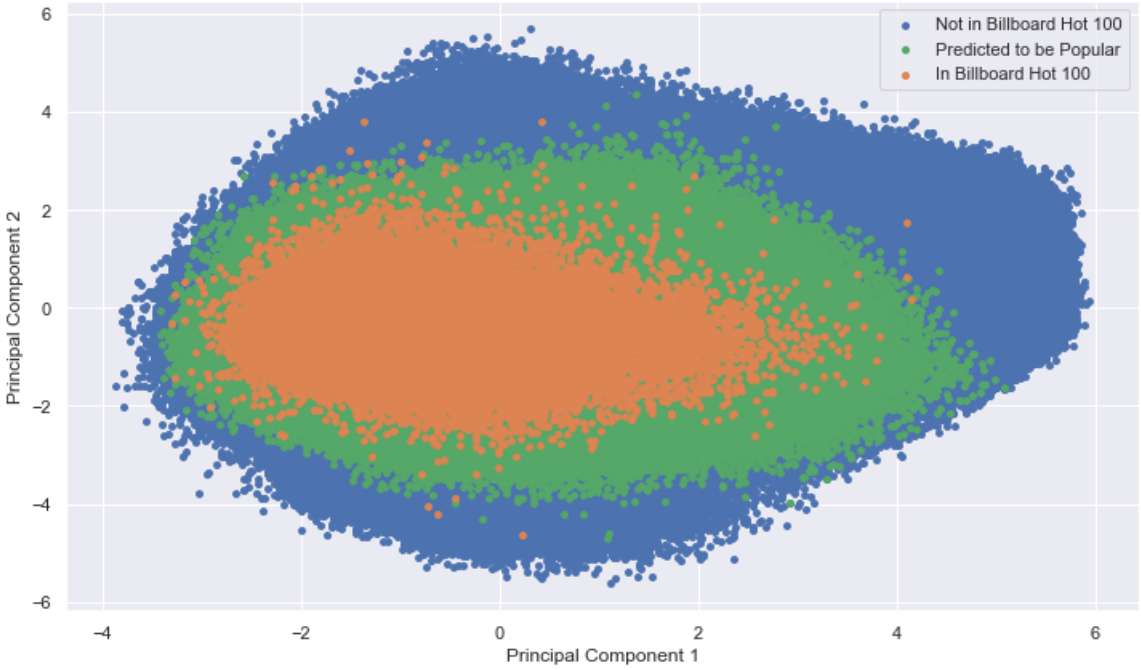
Comparison of Predicted and Actual Billboard Hits: Logistic Regression - Tuned Hyperparameters



Comparison of Predicted and Actual Billboard Hits: Decision Tree - Tuned Hyperparameters



Comparison of Predicted and Actual Billboard Hits: Logistic Regression - Clustered By KMeans Version 1



Comparison of Predicted and Actual Billboard Hits: Logistic Regression - Clustered By KMeans Version 2

Let's just check to see how this plot would look if we chose 2 audio features

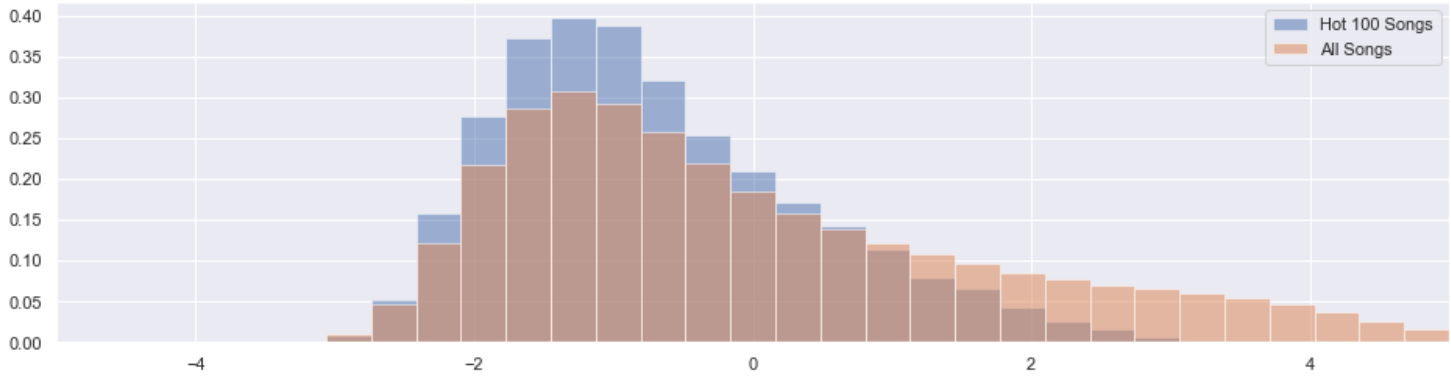
In [ ]:
```
pca_plot(0, 'energy', 'danceability')
```

Histograms

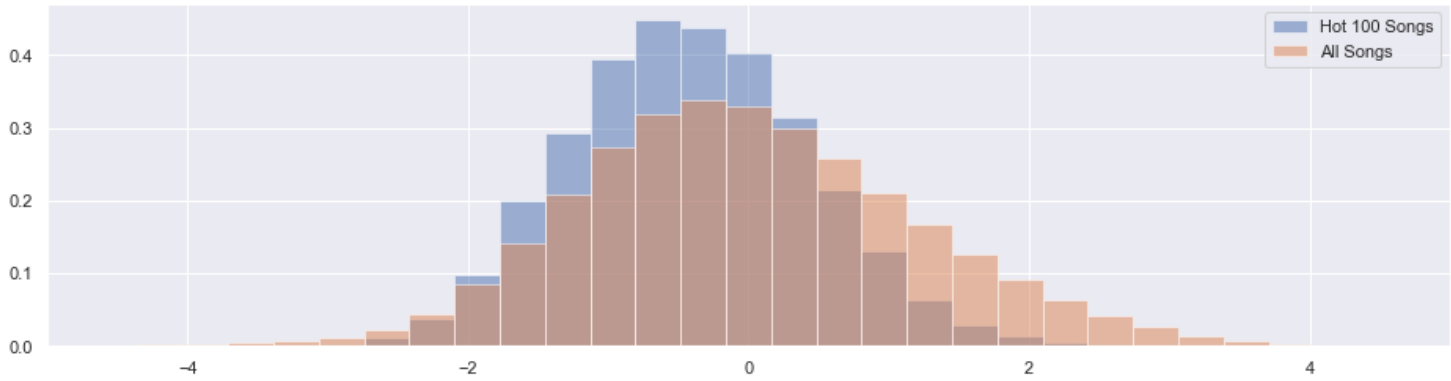
In [14]:
```
# check histograms for PCA

bins = 30
xmin, xmax = -5, 5
bins_plot = [(x/(bins+1))*(xmax-xmin)+xmin for x in range(bins+2)]
figsize = (16, 4)

print('Histogram: Principal Component 1')

plt.figure(figsize=figsize)
plt.xlim(xmin, xmax)
```

```
plt.hist(df_predictions[df_predictions.in_B100]['PCA1'], bins_plot, alpha=0.5, density=True, label='Hot 100 Songs')
plt.hist(df_predictions['PCA1'], bins_plot, alpha=0.5, density=True, label='All Songs')

plt.legend(loc='upper right')
plt.show()

print('Histogram: Principal Component 2')

plt.figure(figsize=figsize)
plt.xlim(xmin, xmax)

plt.hist(df_predictions[df_predictions.in_B100]['PCA2'], bins_plot, alpha=0.5, density=True, label='Hot 100 Songs')
plt.hist(df_predictions['PCA2'], bins_plot, alpha=0.5, density=True, label='All Songs')

plt.legend(loc='upper right')
plt.show()
```

```
Histogram: Principal Component 1
```




```
Histogram: Principal Component 2
```



In [8]:

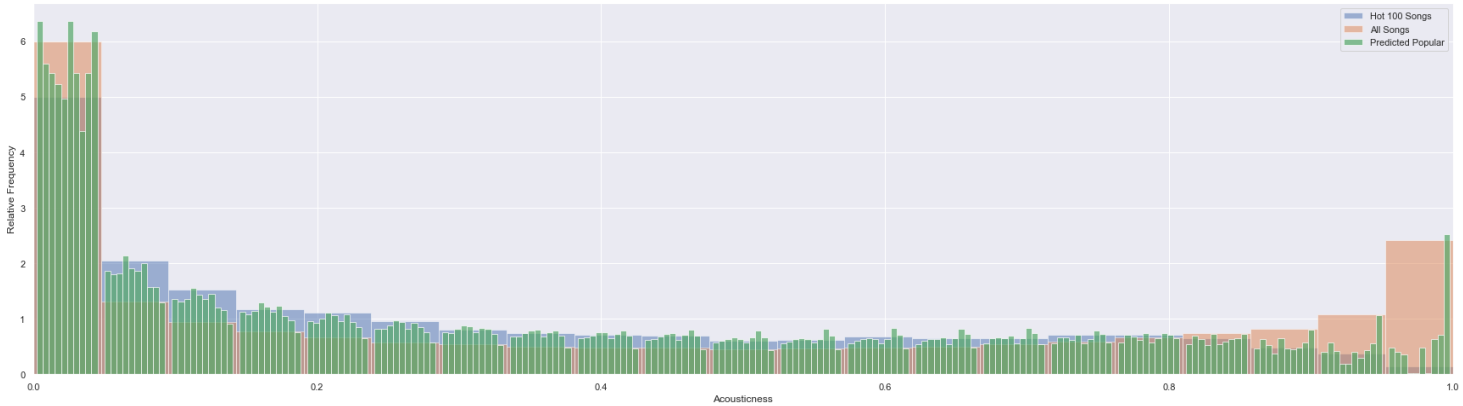
```
def compare_histograms(feature, bins=20, logy=False, figsize=(30, 8), xmin=0, xmax=1):

    plt.figure(figsize=figsize)

    # bins don't line up unless you do this
    plt.xlim(xmin, xmax)
    bins_plot = [(x/(bins+1))*(xmax-xmin)+xmin for x in range(bins+2)]

    # plots - actual
    plt.hist(df_predictions[df_predictions.in_B100][feature], bins_plot, alpha=0.5, density=True, label='Hot 100 Songs')
    plt.hist(df_predictions[feature], bins_plot, alpha=0.5, density=True, label='All Songs')

    # plots - predicitons
    prediction_columns = ['y_lr', 'y_dt', 'y_knn', 'y_rf', 'y_ab', 'y_lr_tuned', 'y_dt_tuned', 'y_cl_1', 'y_cl_2', 'y_genres']
    x_multi = []
    color = []
    for prediction in prediction_columns:
        x_multi.append(df_predictions[df_predictions[prediction]][feature])
        color.append(sns.color_palette()[2])

    plt.hist(
        x_multi,
        bins_plot,
        alpha=0.7,
        density=True,
        label='Predicted Popular',
```

```
        color=color,
        edgecolor='white',  #sns.color_palette()[2]
#         ls='dotted',
        rwidth=0.9
    )

    title = f'{feature.title()} Histogram: Comparing Hot 100 Songs with Predicted Popularity'
    plt.xlabel(feature.title().replace('_', ' '))
    plt.legend(loc='upper right')
    if logy:
        plt.yscale('log')
        plt.ylabel('Log Relative Frequency')
    else:
        plt.ylabel('Relative Frequency')

    # print the title
    print(title)

    plt.show()
```

In [22]:

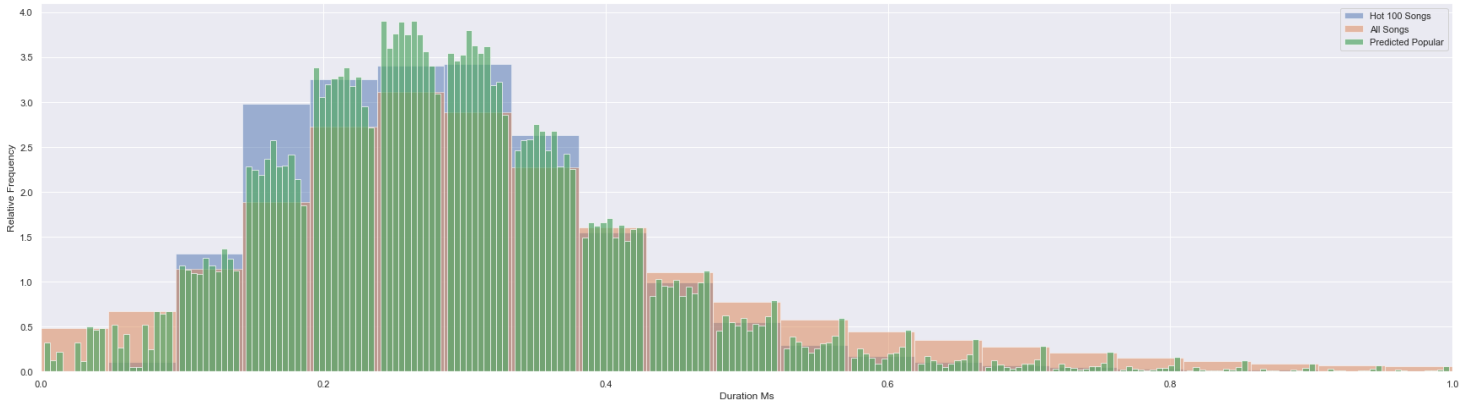
```
%%time
audio_features = [
    'acousticness', 'danceability', 'duration_ms', 'energy', 'instrumentalness',
    'liveness', 'loudness', 'mode', 'speechiness', 'tempo', 'valence'
]

for feature in audio_features:
    compare_histograms(feature)
```

Acousticness Histogram: Comparing Hot 100 Songs with Predicted Popularity



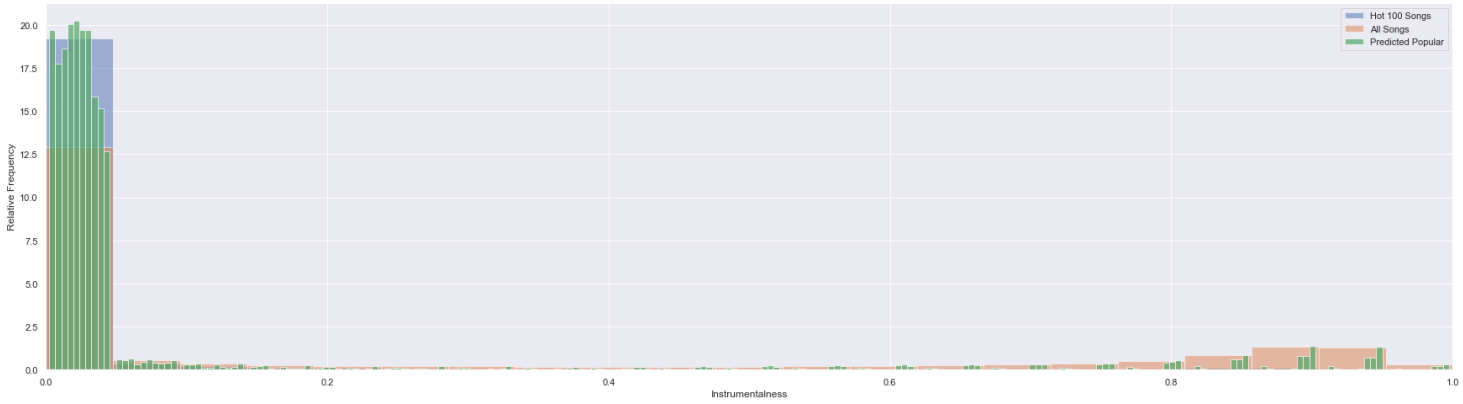
Danceability Histogram: Comparing Hot 100 Songs with Predicted Popularity

Duration_Ms Histogram: Comparing Hot 100 Songs with Predicted Popularity

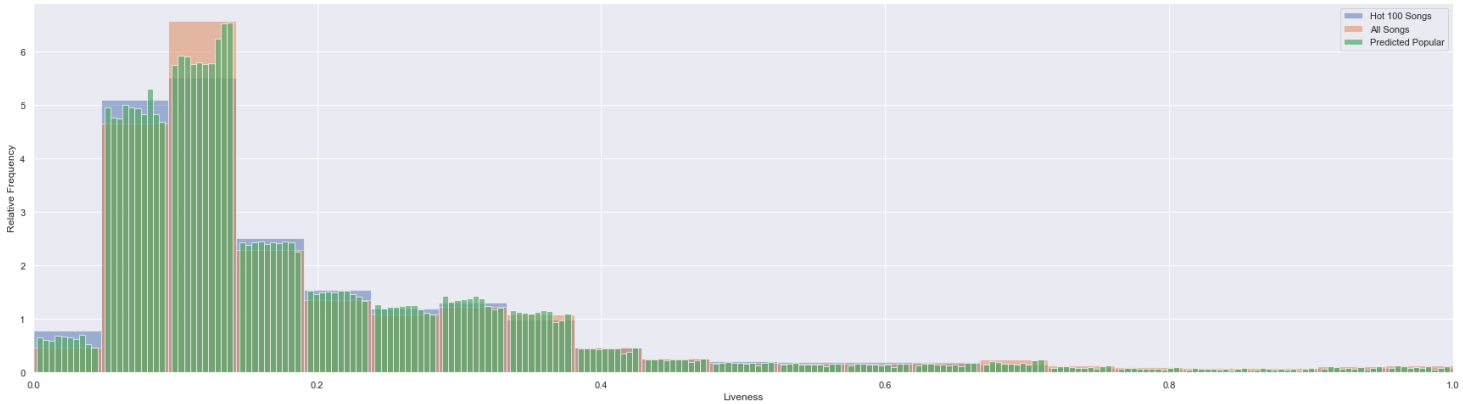


Energy Histogram: Comparing Hot 100 Songs with Predicted Popularity

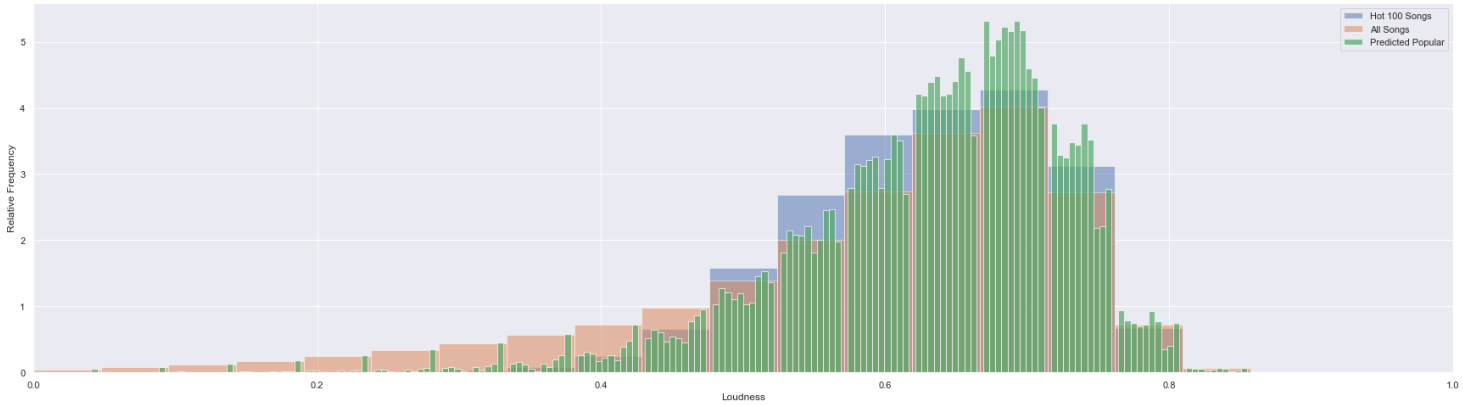
Instrumentalness Histogram: Comparing Hot 100 Songs with Predicted Popularity



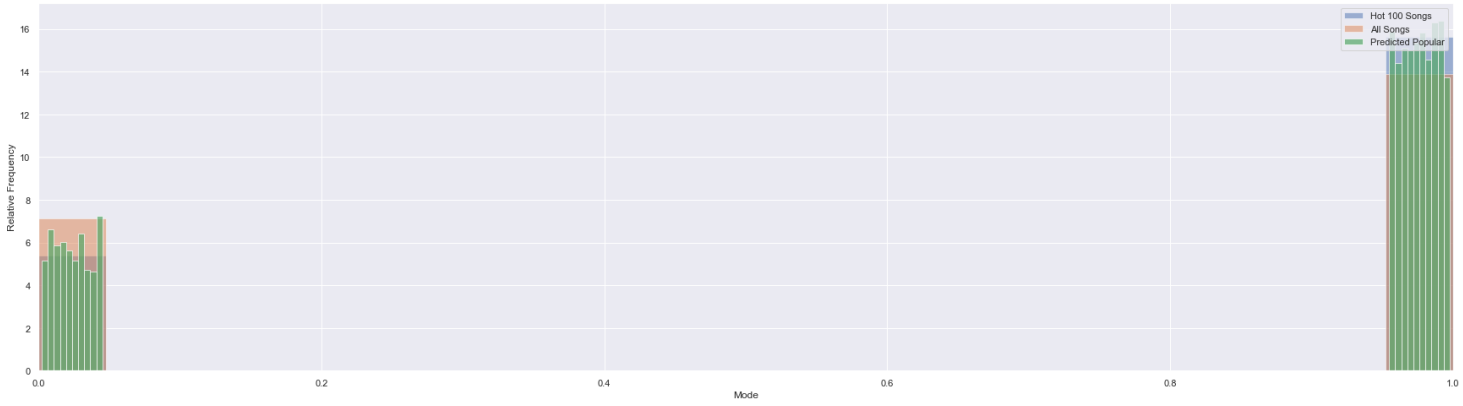
Liveness Histogram: Comparing Hot 100 Songs with Predicted Popularity



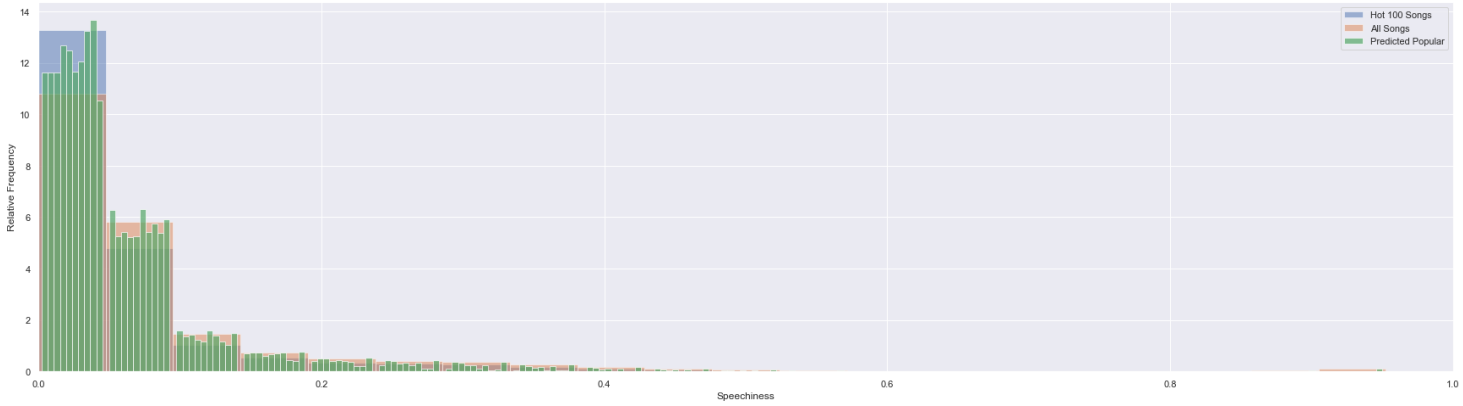
Loudness Histogram: Comparing Hot 100 Songs with Predicted Popularity



Mode Histogram: Comparing Hot 100 Songs with Predicted Popularity

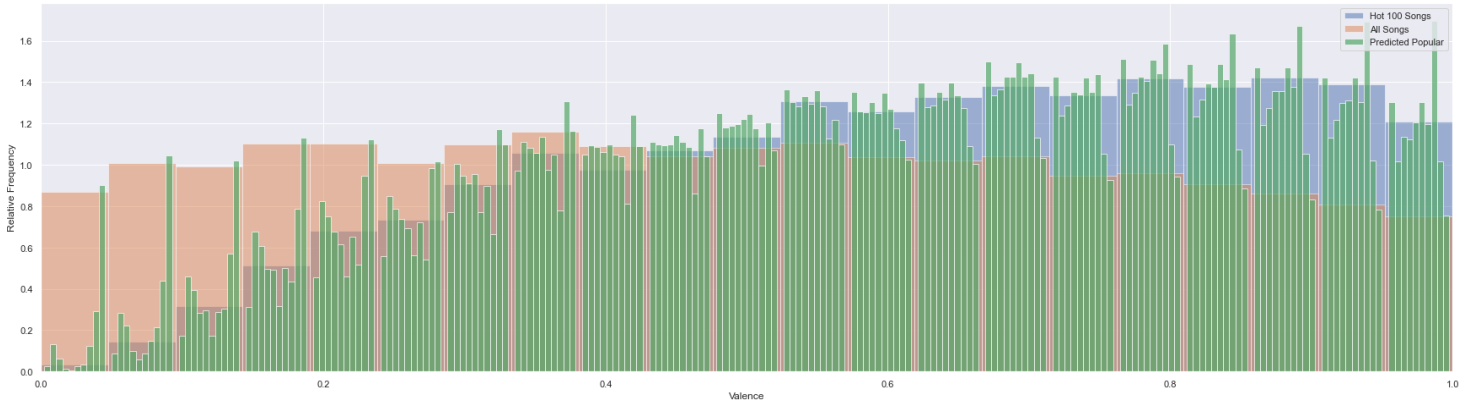


Speechiness Histogram: Comparing Hot 100 Songs with Predicted Popularity



Tempo Histogram: Comparing Hot 100 Songs with Predicted Popularity

Valence Histogram: Comparing Hot 100 Songs with Predicted Popularity

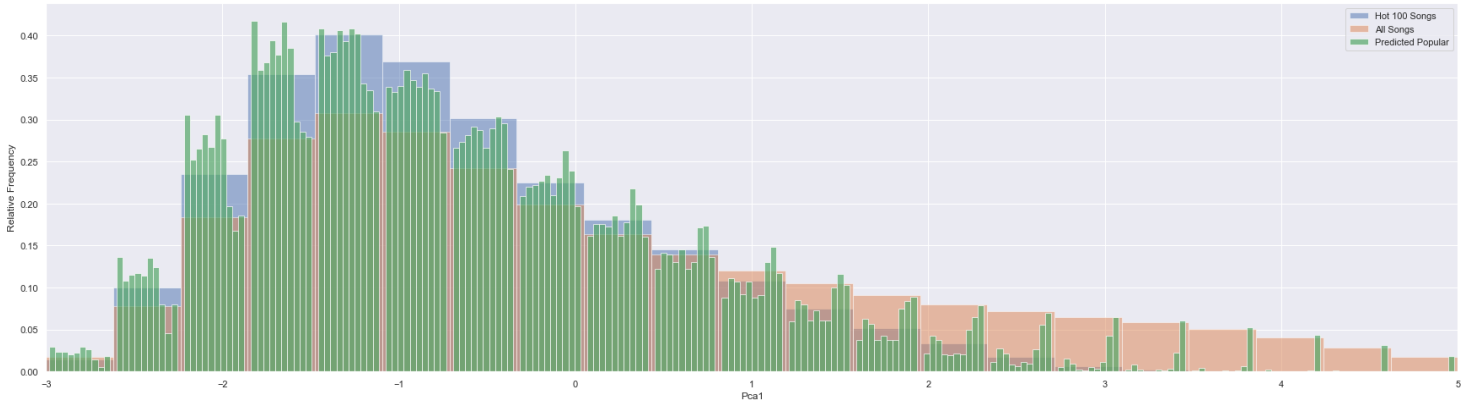



```
Wall time: 2min 9s
```

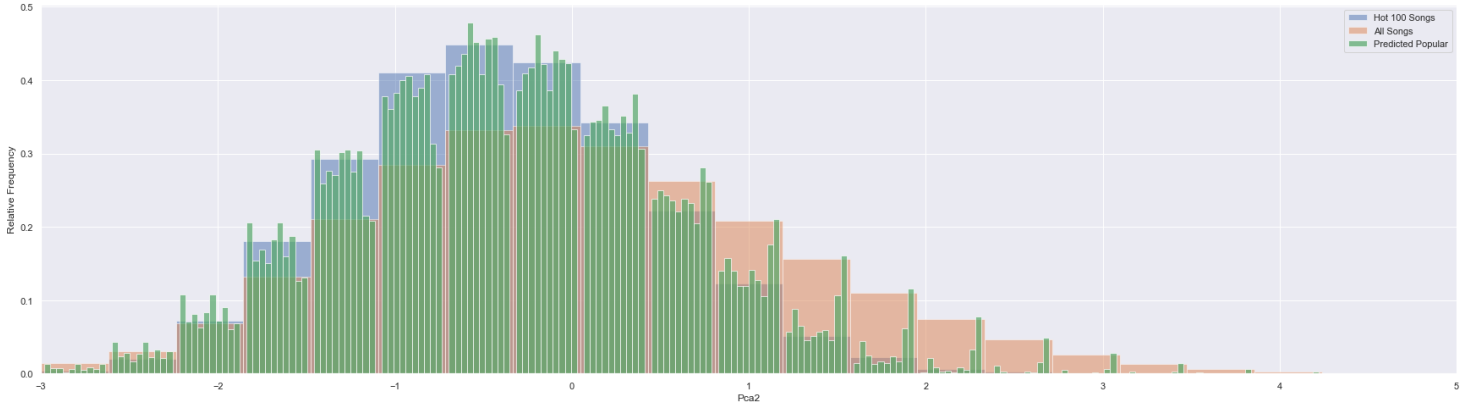
In [23]:

```
# check principal components
compare_histograms('PCA1', xmin=-3, xmax=5)
compare_histograms('PCA2', xmin=-3, xmax=5)
```

```
Pca1 Histogram: Comparing Hot 100 Songs with Predicted Popularity
```




```
Pca2 Histogram: Comparing Hot 100 Songs with Predicted Popularity
```



Statistics

- Friedman Test (Wilcoxon if required)
- R Squared

Friedman Test

- REJECT Null Hypothesis
  - to any arbitrary p value
  - prediction methods are not the same

In [29]:

```
# Friedman Test - Are the models different? to what statistical significance?
friedmanchisquare(*[df_predictions[column] for column in prediction_columns])
```

Out[29]:
```
FriedmanchisquareResult(statistic=4316978.654631455, pvalue=0.0)
```

Wilcoxon Signed Rank Tests

- REJECT Null Hypothesis (except prediction vs itself)
  - to any arbitrary p value
  - no prediction methods are equivalent

In [68]:
```
predictions_to_columns = [df_predictions[column] for column in prediction_columns]
n_col = len(prediction_columns)

wilcoxon_results = pd.DataFrame(
    np.nan,
    index=prediction_columns,
    columns=[x[:6] for x in prediction_columns]  # column names truncated for better spacing
)

for i in range(n_col):
    for j in range(n_col):
        if i == j:
            wilcoxon_results.iloc[i, j] = 1  # p=1 that groups match (i == j)
        else:
            # doesn't work with boolean / categorical values, so multiply by 1 to get 0 or 1s
            wilcoxon_results.iloc[i, j] = wilcoxon(predictions_to_columns[i]*1, predictions_to_columns[j]*1).pvalue

wilcoxon_results
```

Out[68]:

	y_lr	y_dt	y_knn	y_rf	y_ab	y_lr_t	y_dt_t	y_cl_1	y_cl_2	y_genr
y_lr	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
y_dt	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
y_knn	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
y_rf	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000
y_ab	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000
y_lr_tuned	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000
y_dt_tuned	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000
y_cl_1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000	0.000
y_cl_2	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000
y_genres	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000

R Squared

- These results imply that the models are bad, but I'm having trouble interpretting this
- This implies that the results are different than chance
  - but also that the predictions are bad
  - predicting 100% 'No' is the most *accurate* model, but not the *best*

In [79]:
```
y_true = df_predictions['in_B100']

for prediction in prediction_columns:
    r_squared_score = r2_score(y_true*1, df_predictions[prediction]*1)
    print(
        predictions_dict[prediction],
        'R-Squared Value: ',
        round(r_squared_score, 1)
    )
```

```
Logistic Regression - Default Hyperparameters R-Squared Value:  -178.6
Decision Tree - Default Hyperparameters R-Squared Value:  -143.6
K-Nearest Neighbours - Default Hyperparameters R-Squared Value:  -160.8
Random Forest - Default Hyperparameters R-Squared Value:  -133.2
AdaBoost - Default Hyperparameters R-Squared Value:  -161.7
Logistic Regression - Tuned Hyperparameters R-Squared Value:  -178.4
Decision Tree - Tuned Hyperparameters R-Squared Value:  -144.7
```

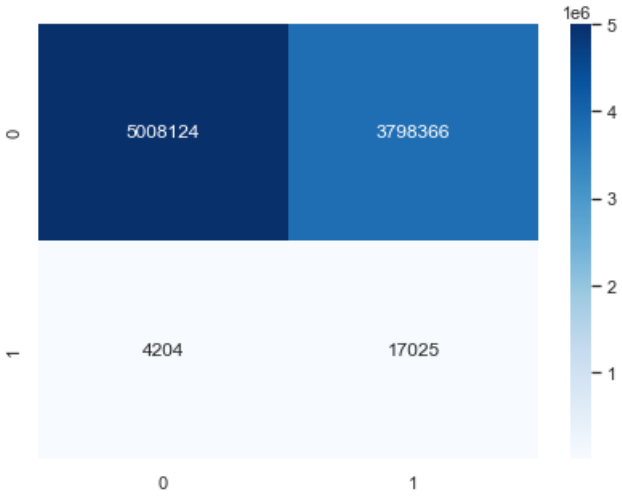
```
Logistic Regression - Clustered By KMeans Version 1 R-Squared Value:  -154.6
Logistic Regression - Clustered By KMeans Version 2 R-Squared Value:  -158.3
Logistic Regression - Clustered By Genre R-Squared Value:  -52.4
```

Which Models Worked The Best?

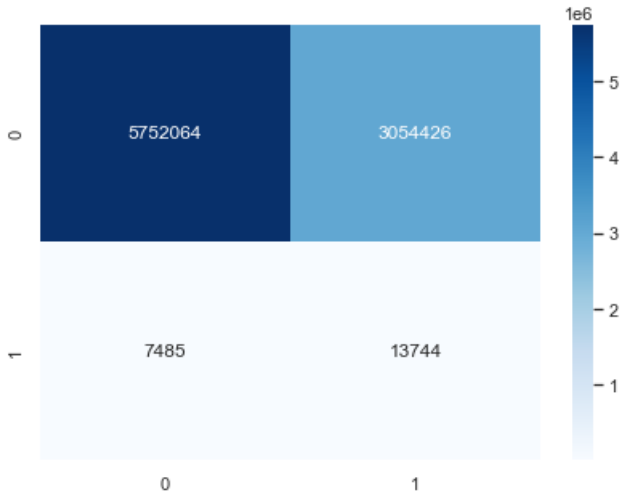
In [102…

```
%%time

modelling_results = {}

for prediction in prediction_columns:

    y, y_pred = df_predictions['in_B100'], df_predictions[prediction]

    print('-------------------------------------------------\n', predictions_dict[prediction])

    # classification_report
    modelling_results[prediction] = classification_report(y, y_pred, output_dict=True)
    print(classification_report(y, y_pred))

    # confusion matrix
    plt.figure(figsize=(7,5))
    sns.heatmap(
        confusion_matrix(y, y_pred),
        cmap='Blues',
        annot=True,
        fmt='.0f'
    )
    plt.show()
```

```
-------------------------------------------------
 Logistic Regression - Default Hyperparameters
              precision    recall  f1-score   support

       False       1.00      0.57      0.72   8806490
        True       0.00      0.80      0.01     21229

    accuracy                           0.57   8827719
   macro avg       0.50      0.69      0.37   8827719
weighted avg       1.00      0.57      0.72   8827719
```

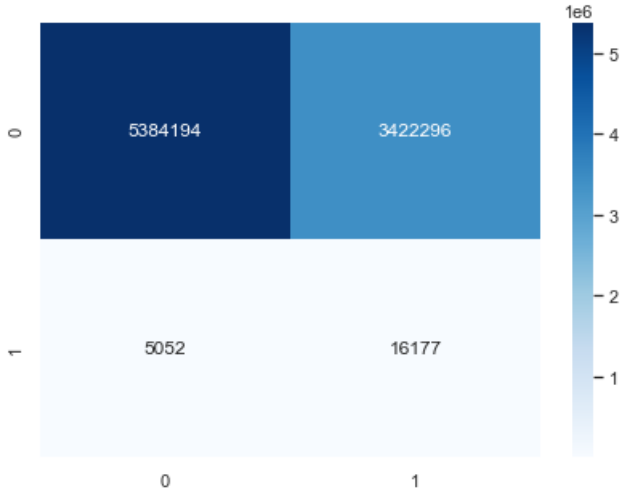



```
-------------------------------------------------
 Decision Tree - Default Hyperparameters
              precision    recall  f1-score   support

       False       1.00      0.65      0.79   8806490
        True       0.00      0.65      0.01     21229

    accuracy                           0.65   8827719
   macro avg       0.50      0.65      0.40   8827719
weighted avg       1.00      0.65      0.79   8827719
```



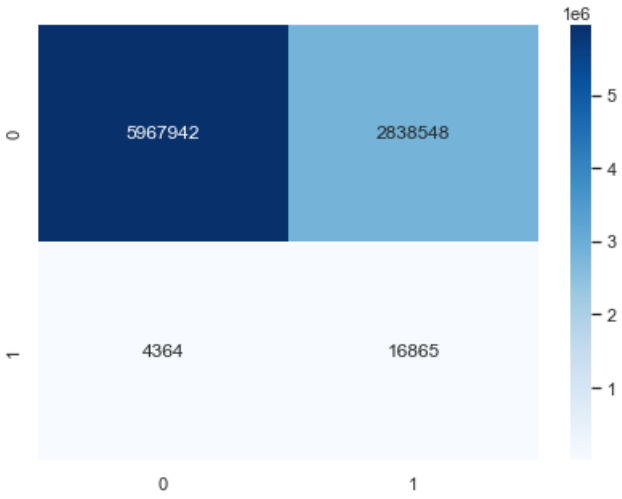
K-Nearest Neighbours - Default Hyperparameters

	precision	recall	f1-score	support
False	1.00	0.61	0.76	8806490
True	0.00	0.76	0.01	21229
accuracy			0.61	8827719
macro avg	0.50	0.69	0.38	8827719
weighted avg	1.00	0.61	0.76	8827719



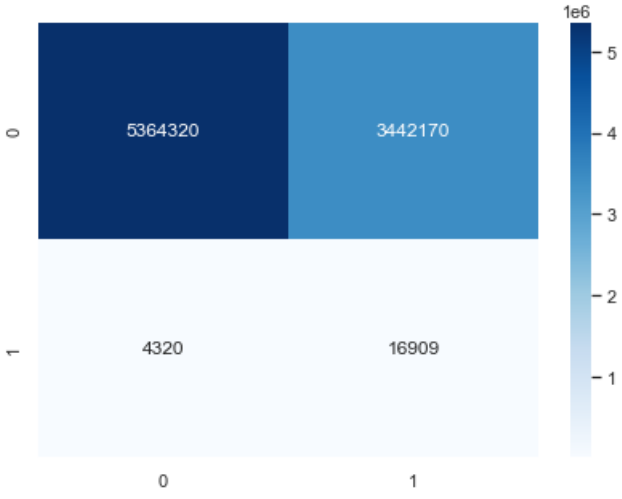
Random Forest - Default Hyperparameters

	precision	recall	f1-score	support
False	1.00	0.68	0.81	8806490
True	0.01	0.79	0.01	21229
accuracy			0.68	8827719
macro avg	0.50	0.74	0.41	8827719
weighted avg	1.00	0.68	0.81	8827719



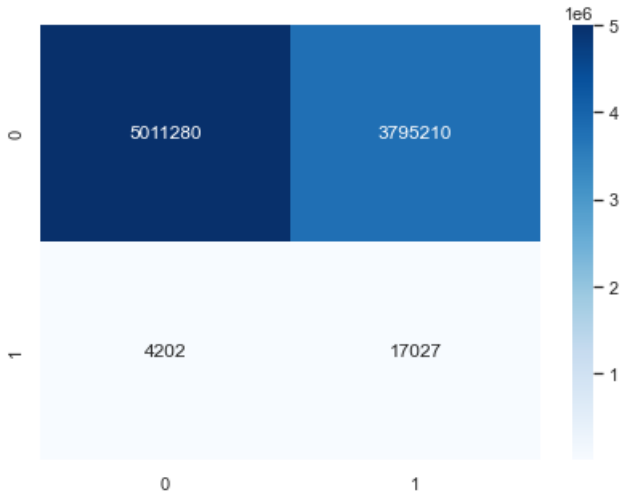
AdaBoost - Default Hyperparameters

	precision	recall	f1-score	support
False	1.00	0.61	0.76	8806490
True	0.00	0.80	0.01	21229
accuracy			0.61	8827719
macro avg	0.50	0.70	0.38	8827719
weighted avg	1.00	0.61	0.76	8827719



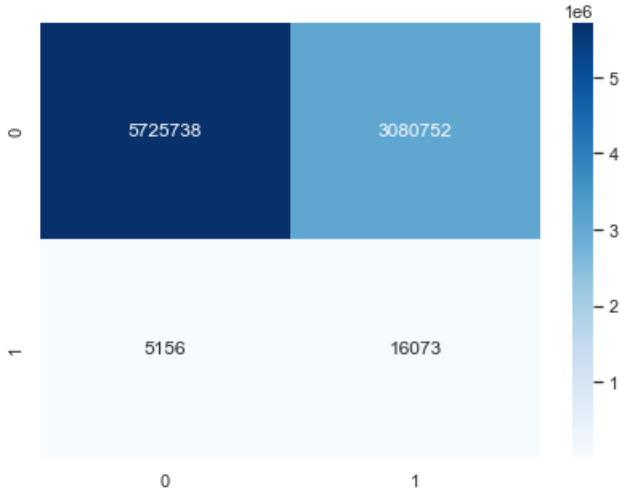
Logistic Regression - Tuned Hyperparameters

	precision	recall	f1-score	support
False	1.00	0.57	0.73	8806490
True	0.00	0.80	0.01	21229
accuracy			0.57	8827719
macro avg	0.50	0.69	0.37	8827719
weighted avg	1.00	0.57	0.72	8827719



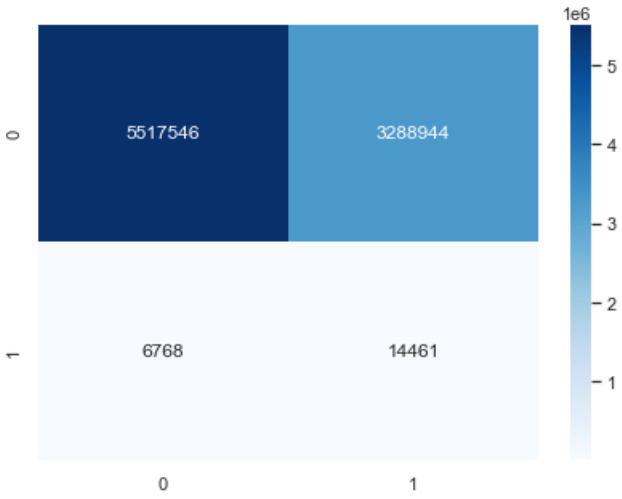
Decision Tree - Tuned Hyperparameters

	precision	recall	f1-score	support
False	1.00	0.65	0.79	8806490
True	0.01	0.76	0.01	21229
accuracy			0.65	8827719
macro avg	0.50	0.70	0.40	8827719
weighted avg	1.00	0.65	0.79	8827719



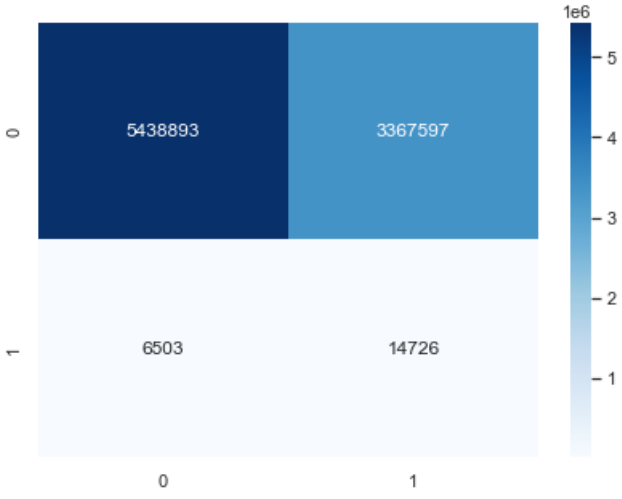
Logistic Regression - Clustered By KMeans Version 1

	precision	recall	f1-score	support
False	1.00	0.63	0.77	8806490
True	0.00	0.68	0.01	21229
accuracy			0.63	8827719
macro avg	0.50	0.65	0.39	8827719
weighted avg	1.00	0.63	0.77	8827719



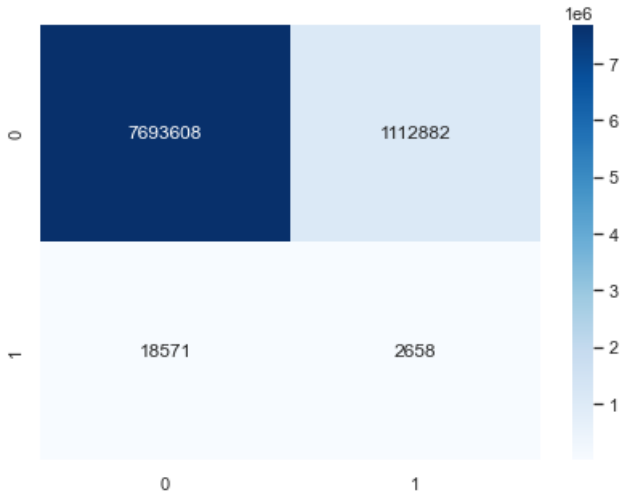
Logistic Regression - Clustered By KMeans Version 2

	precision	recall	f1-score	support
False	1.00	0.62	0.76	8806490
True	0.00	0.69	0.01	21229
accuracy			0.62	8827719
macro avg	0.50	0.66	0.39	8827719
weighted avg	1.00	0.62	0.76	8827719



Logistic Regression - Clustered By Genre

	precision	recall	f1-score	support
False	1.00	0.87	0.93	8806490
True	0.00	0.13	0.00	21229
accuracy			0.87	8827719
macro avg	0.50	0.50	0.47	8827719
weighted avg	1.00	0.87	0.93	8827719



Wall time: 2min 54s

In [117…

```
%%time
# calculate precision, recall, f1, and accuracy for each model, save to a dataframe

results_metrics = {}

for prediction in prediction_columns:
    precision = modelling_results[prediction]['True']['precision']
    recall = modelling_results[prediction]['True']['recall']
    f1 = modelling_results[prediction]['True']['f1-score']
    accuracy = modelling_results[prediction]['accuracy']

    results_metrics[predictions_dict[prediction]] = [precision, recall, f1, accuracy]

results_metrics = pd.DataFrame(results_metrics, index=['Precision', 'Recall', 'F1 Score', 'Accuracy']).T
```

Wall time: 1.99 ms

In [119…

```
# sorted by precision
results_metrics.sort_values('Precision', ascending=False)
```

Out[119…

	Precision	Recall	F1 Score	Accuracy
Random Forest - Default Hyperparameters	0.006	0.794	0.012	0.678
Decision Tree - Tuned Hyperparameters	0.005	0.757	0.010	0.650
AdaBoost - Default Hyperparameters	0.005	0.797	0.010	0.610
K-Nearest Neighbours - Default Hyperparameters	0.005	0.762	0.009	0.612
Decision Tree - Default Hyperparameters	0.004	0.647	0.009	0.653
Logistic Regression - Tuned Hyperparameters	0.004	0.802	0.009	0.570
Logistic Regression - Default Hyperparameters	0.004	0.802	0.009	0.569
Logistic Regression - Clustered By KMeans Version 1	0.004	0.681	0.009	0.627
Logistic Regression - Clustered By KMeans Version 2	0.004	0.694	0.009	0.618
Logistic Regression - Clustered By Genre	0.002	0.125	0.005	0.872

In [118…

```
# sorted by recall
results_metrics.sort_values('Recall', ascending=False)
```

Out[118…

	Precision	Recall	F1 Score	Accuracy
Logistic Regression - Tuned Hyperparameters	0.004	0.802	0.009	0.570
Logistic Regression - Default Hyperparameters	0.004	0.802	0.009	0.569
AdaBoost - Default Hyperparameters	0.005	0.797	0.010	0.610
Random Forest - Default Hyperparameters	0.006	0.794	0.012	0.678

	Precision	Recall	F1 Score	Accuracy
K-Nearest Neighbours - Default Hyperparameters	0.005	0.762	0.009	0.612
Decision Tree - Tuned Hyperparameters	0.005	0.757	0.010	0.650
Logistic Regression - Clustered By KMeans Version 2	0.004	0.694	0.009	0.618
Logistic Regression - Clustered By KMeans Version 1	0.004	0.681	0.009	0.627
Decision Tree - Default Hyperparameters	0.004	0.647	0.009	0.653
Logistic Regression - Clustered By Genre	0.002	0.125	0.005	0.872

In [120…

```
# sorted by f1 score
results_metrics.sort_values('F1 Score', ascending=False)
```

Out[120…

	Precision	Recall	F1 Score	Accuracy
Random Forest - Default Hyperparameters	0.006	0.794	0.012	0.678
Decision Tree - Tuned Hyperparameters	0.005	0.757	0.010	0.650
AdaBoost - Default Hyperparameters	0.005	0.797	0.010	0.610
K-Nearest Neighbours - Default Hyperparameters	0.005	0.762	0.009	0.612
Decision Tree - Default Hyperparameters	0.004	0.647	0.009	0.653
Logistic Regression - Tuned Hyperparameters	0.004	0.802	0.009	0.570
Logistic Regression - Default Hyperparameters	0.004	0.802	0.009	0.569
Logistic Regression - Clustered By KMeans Version 1	0.004	0.681	0.009	0.627
Logistic Regression - Clustered By KMeans Version 2	0.004	0.694	0.009	0.618
Logistic Regression - Clustered By Genre	0.002	0.125	0.005	0.872

In [121…

```
# sorted by accuracy
results_metrics.sort_values('Accuracy', ascending=False)
```

Out[121…

	Precision	Recall	F1 Score	Accuracy
Logistic Regression - Clustered By Genre	0.002	0.125	0.005	0.872
Random Forest - Default Hyperparameters	0.006	0.794	0.012	0.678
Decision Tree - Default Hyperparameters	0.004	0.647	0.009	0.653
Decision Tree - Tuned Hyperparameters	0.005	0.757	0.010	0.650
Logistic Regression - Clustered By KMeans Version 1	0.004	0.681	0.009	0.627
Logistic Regression - Clustered By KMeans Version 2	0.004	0.694	0.009	0.618
K-Nearest Neighbours - Default Hyperparameters	0.005	0.762	0.009	0.612
AdaBoost - Default Hyperparameters	0.005	0.797	0.010	0.610
Logistic Regression - Tuned Hyperparameters	0.004	0.802	0.009	0.570
Logistic Regression - Default Hyperparameters	0.004	0.802	0.009	0.569

Future Work

- use PCA to define ranges for audio features which are more likley to be popular
  - https://stats.stackexchange.com/questions/229092/how-to-reverse-pca-and-reconstruct-original-variables-from-several-principal-com

In [ ]: