Music Recommendation System

Team TBD
Final Report for CSC 448 Project

Group Members:

- **Anthony Zhu**
- **Alan Concepcion**
- **Konrad Zielinski**
- **Daphne Tang**

Table of Contents

Abstract

The goal of this project is to utilize our learnings throughout the semester and incorporate external research to create a model to solve a problem. Our problem that we strive to solve is to identify similar songs to our personal favorite music tracks, intended for all music lovers and Spotify users. We intend to recommend songs similar to a certain track (accepted through user input in the form of a Spotify link). To achieve this, we incorporated the Spotify API to process the input song and allows us to extract its attributes and features of the song. We utilized two unsupervised machine learning models: k-means and k-nearest neighbors. Along with the two different models, we also incorporated 4 distinct dimension reduction techniques: Factor Analysis, FeatureAgglomeration Reduction, Feature Selection, and Low Variance Filter. With this, we can achieve our endgame by obtaining recommendations based on the song's characteristics and features. The greatest takeaway from this project is that despite the volume of the extensive dataset we used, we were unable to procure accurate enough results. More details can be found in the analysis section of the report.

Introduction

We are group 4, Team TBD. Our members include:

Alan Concepcion: I created a K-NN model with factor analysis for the back-end; assisted in front-end development; set up the keys for the Spotify API; structured the report

Konrad Zielinski: I created a K-NN model which uses feature agglomeration to perform dimensionality reduction; created the basis for the front-end functionality; created the basis for our flask backend and connected it to the front-end; wrote for the Modeling and K-Nearest Neighbors sections, as well as the FeatureAgglomeration section in the report.

Anthony Zhu: I created a K-Means Clustering model with feature selection to group the data songs in the data set into 10 clusters and integrated the model with the flask app.

Daphne Tang: I created a K-means model with low variance filtering for the back-end; integrated the model into the flask application; created visualizations for the Analysis part; proofread and edited the report; wrote the K-means Clustering and Low Variance Filter sections in the report

The models used in our project are K-means clustering and K-nearest neighbors. We decided on 2 unsupervised learning models while using 4 distinct dimension reduction techniques. The reduction techniques used are Factor Analysis, FeatureAgglomeration Reduction, Feature Selection reduction, and Low Variance Filter. More information will be provided below in the analysis section.

Libraries used in this Project:

- **Pandas**; used for reading the datasets, creating one of our own, and exploratory data analysis
- **dotenv /os**; used to create environmental variables
- **Spotipy**; Spotify API that allows us to show audio previews and album art covers in our application and helps us process the user input and retrieve information to use in our models
- **Sklearn**; using k-means and k-nearest neighbors unsupervised learning models rather than implementing the two from scratch, using the scalers from the preprocessing library, using normalize to normalize the variable values for low variance filter
- **Matplotlib**; this is used to create graphs for analysis purposes
- **Factor_analyzer**; this is used for the factor analysis portion of this report
- **Seaborn**; used for some visualizations

- **Plotly.express**; used to create most of the data visualizations

Pre-processing

We start by loading the necessary libraries

```
import numpy as np
import pandas as pd
from dotenv import load_dotenv
import os
from sklearn.cluster import FeatureAgglomeration
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler, MinMaxScaler, MaxAbsScaler
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
import matplotlib.pyplot as plt
import plotly.express as px

import string
import re
```

We then have to load the dataset

```
df_spotify = pd.read_csv(SPOTIFY_DATA)
```

We want to look for duplicate rows and any null rows

```
print(f"Shape of Dataset: {df_spotify.shape}")

num_duplicate_rows = df_spotify.duplicated().sum()
print(f"\nNumber of Duplicate Rows: {num_duplicate_rows}")

any_null_values = df_spotify.isnull().any()
print(f"\nAny Null Values in DataFrame: \n{any_null_values}")
```

Which results in

```
Shape of Dataset: (170653, 1

Number of Duplicate Rows: 0

Any Null Values in DataFrame
valence             False
year                False
acousticness        False
artists             False
danceability        False
duration_ms         False
energy              False
explicit            False
id                  False
instrumentalness    False
key                 False
liveness            False
loudness            False
mode                False
name                False
popularity          False
release_date        False
speechiness         False
tempo               False
dtype: bool
```

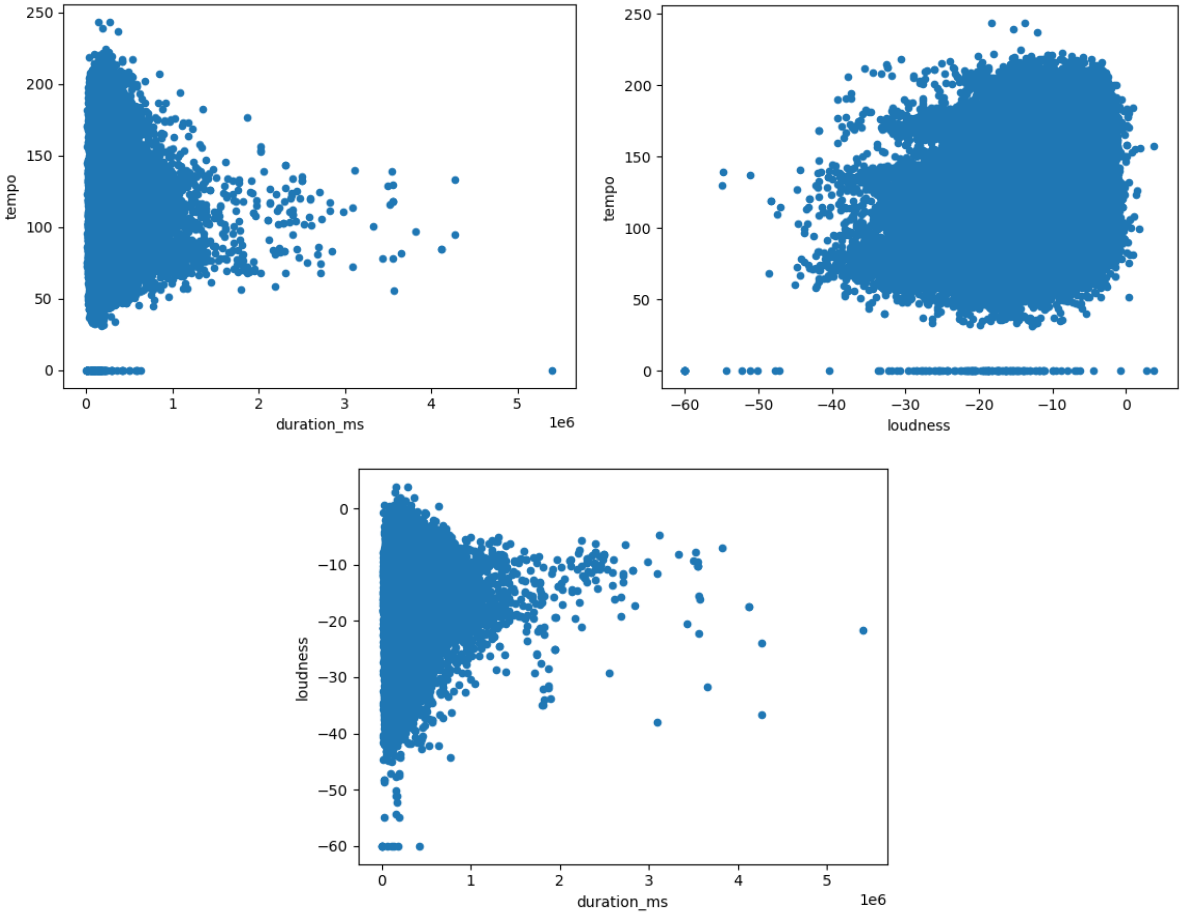
We want to drop any redundant columns such as columns that contain strings as we only want the numerical factors of the songs. We will drop those columns and see what it looks like after

```
df_relevant_columns = df_spotify.drop(columns=['id','name','release_date','year','artists', 'popularity','explicit'], axis=1
display(df_relevant_columns.head(10))
```

	valence	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	0.0594	0.982	0.279	831667	0.211	0.878000	10	0.665	-20.096	1	0.0366	80.954
1	0.9630	0.732	0.819	180533	0.341	0.000000	7	0.160	-12.441	1	0.4150	60.936
2	0.0394	0.961	0.328	500062	0.166	0.913000	3	0.101	-14.850	1	0.0339	110.339
3	0.1650	0.967	0.275	210000	0.309	0.000028	5	0.381	-9.316	1	0.0354	100.109
4	0.2530	0.957	0.418	166693	0.193	0.000002	3	0.229	-10.096	1	0.0380	101.665
5	0.1960	0.579	0.697	395076	0.346	0.168000	2	0.130	-12.506	1	0.0700	119.824
6	0.4060	0.996	0.518	159507	0.203	0.000000	0	0.115	-10.589	1	0.0615	66.221
7	0.0731	0.993	0.389	218773	0.088	0.527000	1	0.363	-21.091	0	0.0456	92.867
8	0.7210	0.996	0.485	161520	0.130	0.151000	5	0.104	-21.508	0	0.0483	64.678
9	0.7710	0.982	0.684	196560	0.257	0.000000	8	0.504	-16.415	1	0.3990	109.378

Modeling

In order to start off in developing and building our models, we first need to figure out how we want to approach the task in hand, keeping in mind the form our dataset takes. The data we are using, after some general preprocessing, consists of data samples full of numeric information for each of the columns we end up with. In our case, we do not have any sort of classes or labels to perform classification on, and the range of our values in each of our columns are all discrete, with the exception of our duration_ms, tempo, and loudness column. We can see by plotting said 3 columns against each other, that there does not seem to be a pattern between them specifically which we could exploit:



With a lack of patterns evident, and only 3 out of 12 columns being non-discrete, performing regression does not seem like it would benefit us much for our end goal. The best way to tackle data samples, such as those in our dataset, is to perform unsupervised learning methods, unlike the supervised learning of classification and regression. In unsupervised learning, patterns are explored within unlabeled data, with no input-output pairs for the data to recognize patterns from. The method of unsupervised learning we chose to go with is clustering,where the goal is to separate data examples into groups, which are called clusters. Clustering algorithms help to provide distinction between groups of similar values, which helps to determine patterns among

unlabeled sets of data. We chose to explore two forms of unsupervised learning for our project, an unsupervised variation of the KNN algorithm, named NearestNeighbours in sklearn, and the K-means algorithm, named KMeans in sklearn, a dedicated unsupervised version of the original KNN algorithm.

K-Nearest Neighbors

The K-Nearest Neighbours (KNN) algorithm is a simple and widely used algorithm in the topic of machine learning, particularly for classification and regression tasks. The problem with using the traditional KNN algorithm for us, however, is that it is considered a supervised learning algorithm, while we need to perform unsupervised learning. To do so, we will be using the NearestNeighbours class in the sklearn library. As provided by the scikit learn documentation on NearestNeighbours, "[t]he principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these". The NearestNeighbours class differs with its unsupervised nearest neighbor learning. The class acts as an interface to three different algorithms, Balltree, KDTree, and a Brute-force algorithm. For our NearestNeighbour models, we will be using the following parameters specifically:

n_neighbors : *int, default=5*
Number of neighbors to use by default for `kneighbors` queries.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*
Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

metric : *str or callable, default='minkowski'*
Metric to use for distance computation. Default is "minkowski", which results in the standard Euclidean distance when p = 2. See the documentation of scipy.spatial.distance and the metrics listed in `distance_metrics` for valid metric values.

If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors.

If metric is a callable function, it takes two arrays representing 1D vectors as inputs and must return one value indicating the distance between those vectors. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

In our case, the Balltree algorithm, and the euclidean metric function will be used for both of the unsupervised KNN models. We do this partially because we want to keep some semblance of consistency in our models, to more easily see the differences that our different dimensionality reduction techniques cause, but in our case since we have 12 features, the Balltree algorithm will

suffice nicely since it "address[es] the inefficiencies of KD Trees in higher dimensions", as supplied by the scikit learn documentation on the subject, and the dimensionality of our data is not so high as to detracts from a tree based methods functionalities. We chose the euclidean metric function as it is a very popular metric to compute distance between neighbors. Finally, we are going to be using the n_neighbours parameter to denote the number of nearest neighbors the kneighbors method will be computing of our input song features.

Implementation Example from our code:

```
knn_model = NearestNeighbors(n_neighbors=5, metric='euclidean', algorithm='ball_tree')
knn_model.fit(X_agglo)
```

```
distances, indices = knn_model.kneighbors(input_song_agglo)
recommended_songs = df_spotify.iloc[indices.flatten()][['artists', 'name']]
```

K-Means Clustering

K-means clustering is a popular unsupervised machine learning algorithm most commonly used for partitioning a dataset into k distinct, non-overlapping subsets, known as clusters. According to the scikit learn documentation, "The algorithm separates samples into n groups of equal variance, minimizing the inertia."

The purpose of using K-means clustering is to group similar data points into clusters. Each cluster has a centroid or the center of the cluster. This algorithm works iteratively as it assigns data points to each cluster, then updates the cluster centroid when necessary.

We used the Kmeans() function from sklearn.cluster library to implement this. Some of the parameters that the function accepts are (shown in the K-Means documentation:

n_clusters : *int, default=8*

The number of clusters to form as well as the number of centroids to generate.

init : *{'k-means++', 'random'}, callable or array-like of shape (n_clusters, n_features), default='k-means++'*

Method for initialization:

- 'k-means++' : selects initial cluster centroids using sampling based on an empirical probability distribution of the points' contribution to the overall inertia. This technique speeds up convergence. The algorithm implemented is "greedy k-means++". It differs from the vanilla k-means++ by making several trials at each sampling step and choosing the best centroid among them.
- 'random': choose `n_clusters` observations (rows) at random from data for the initial centroids.
- If an array is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.
- If a callable is passed, it should take arguments X, n_clusters and a random state and return an initialization.

max_iter : *int, default=300*
Maximum number of iterations of the k-means algorithm for a single run.

tol : *float, default=1e-4*
Relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence.

verbose : *int, default=0*
Verbosity mode.

random_state : *int, RandomState instance or None, default=None*
Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See Glossary.

Implementation Example from our code:

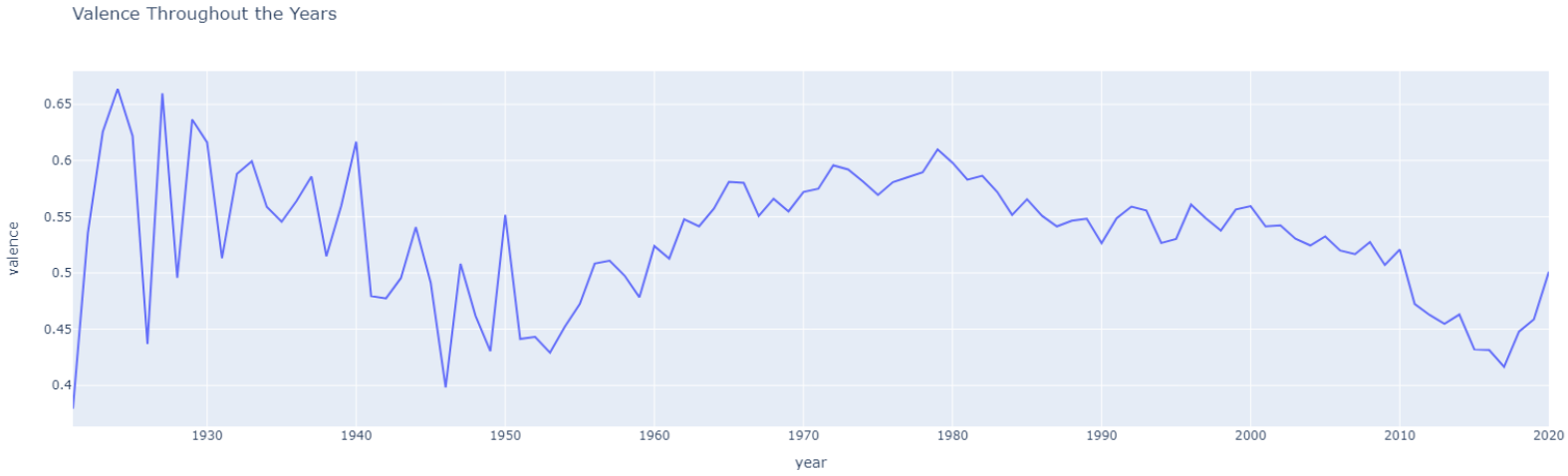
```
# utilizing K-means clustering technique
kmeans_model = KMeans(n_clusters=10)

# fitting the K-means model using our dataset
# (with filtered features)
kmeans_model.fit(df_new)
```

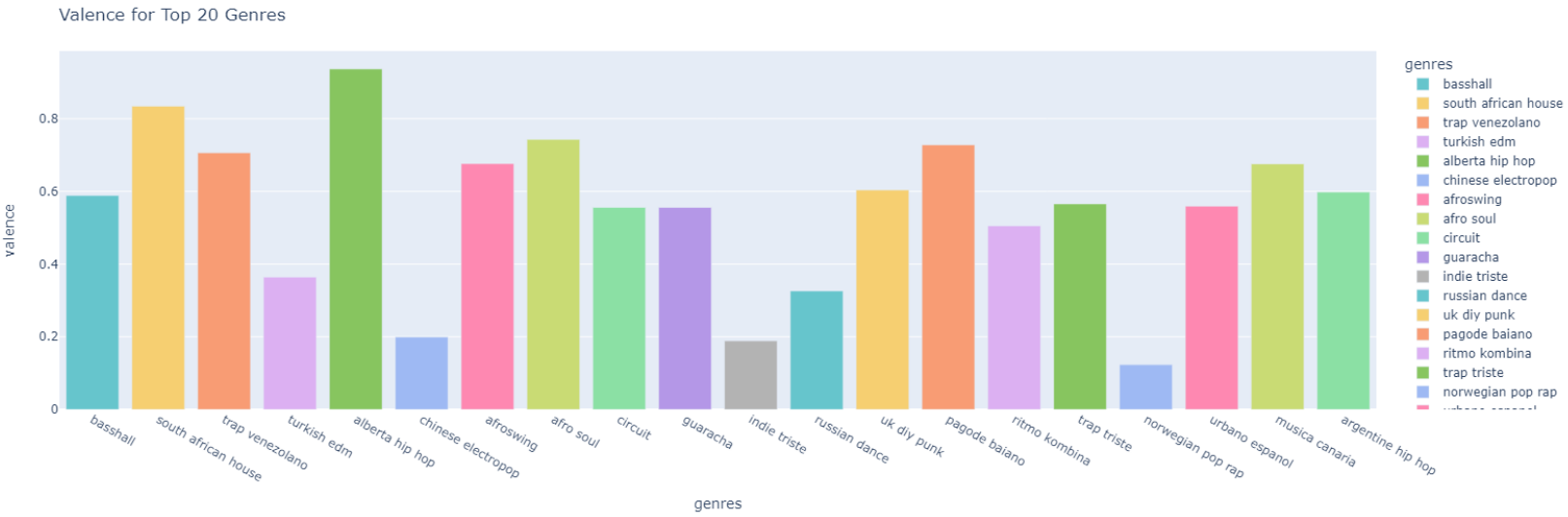
```
clustered = df.copy()
clustered['type'] = kmeans_model.labels_

# creating predictions using Kmeans model
predictions = kmeans_model.predict(df_song)
```

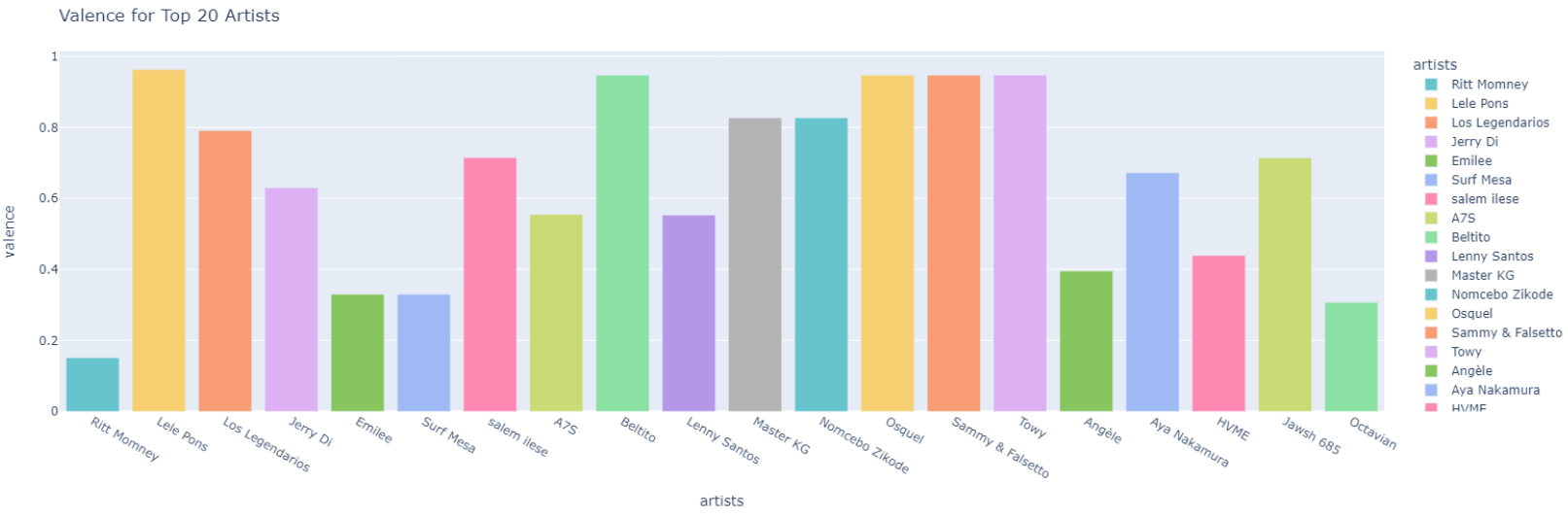
Analysis

After preprocessing the data, we are left with most of the columns. Information on each column:

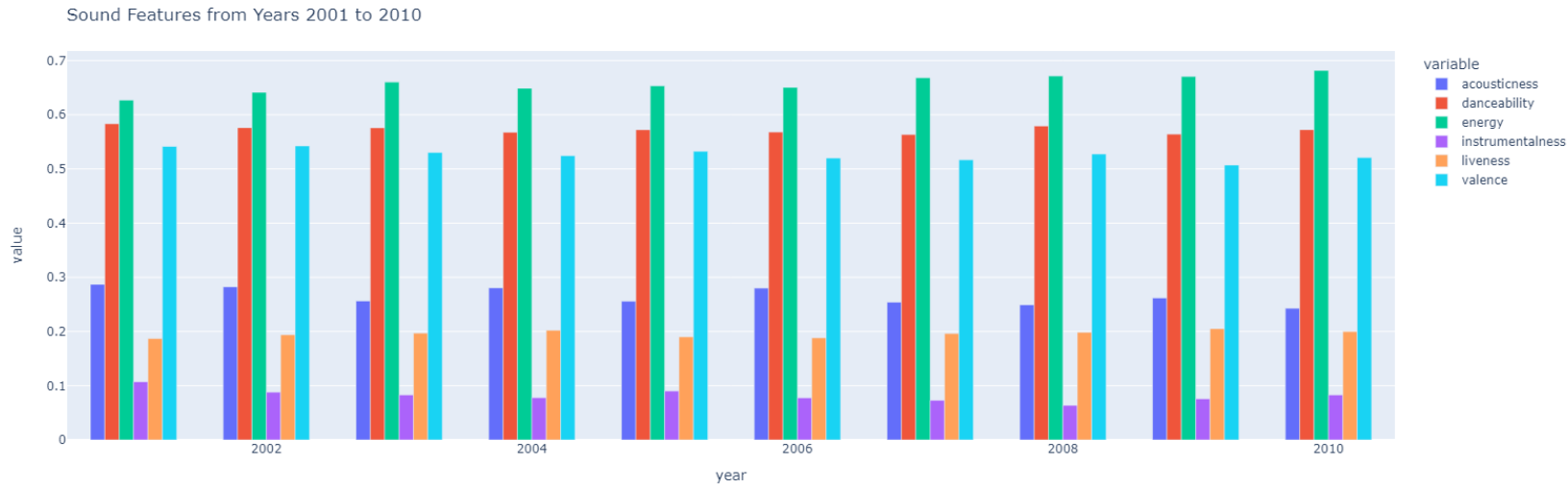
Valence: describes the positivity of the track; measurement ranges from 0.0 to 1.0

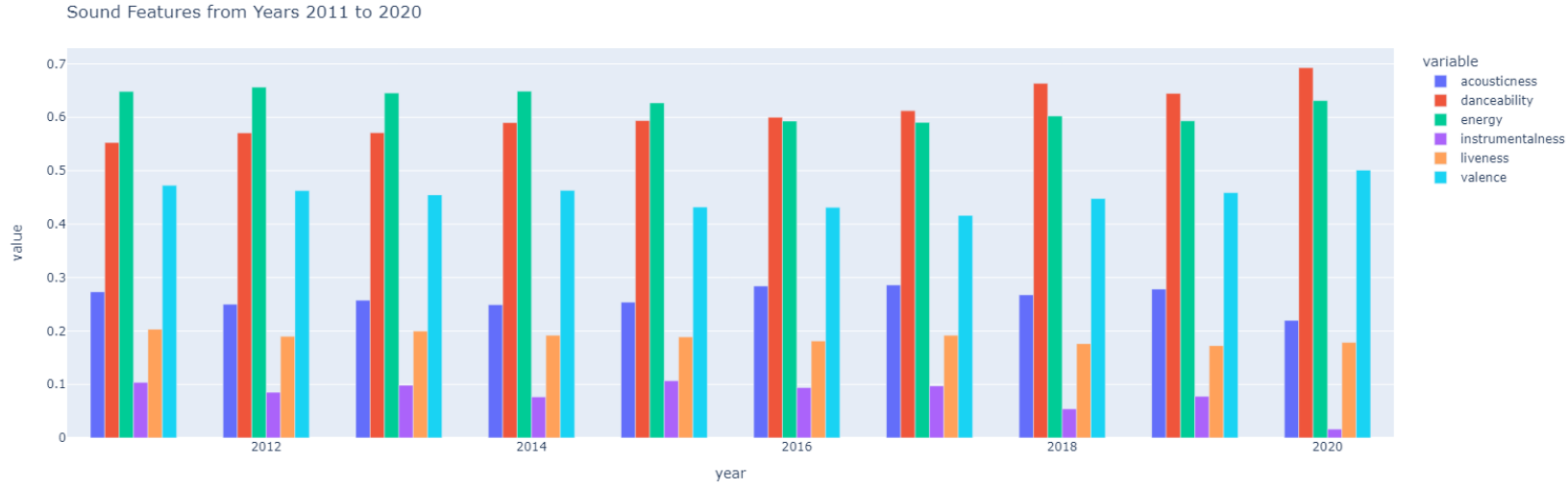


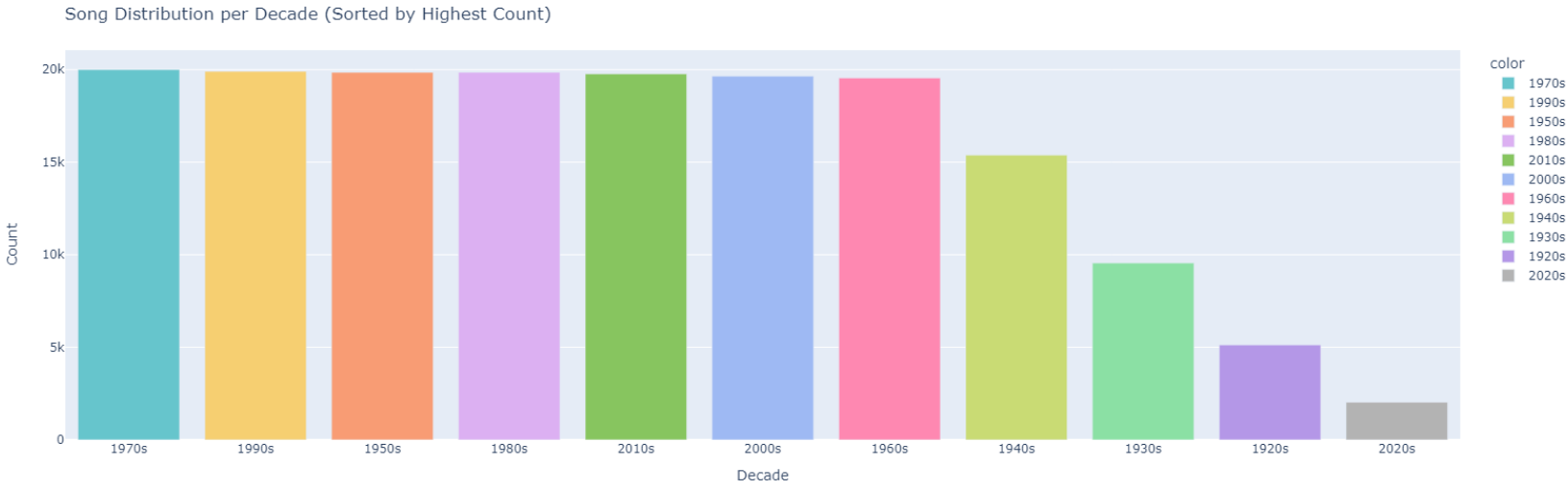




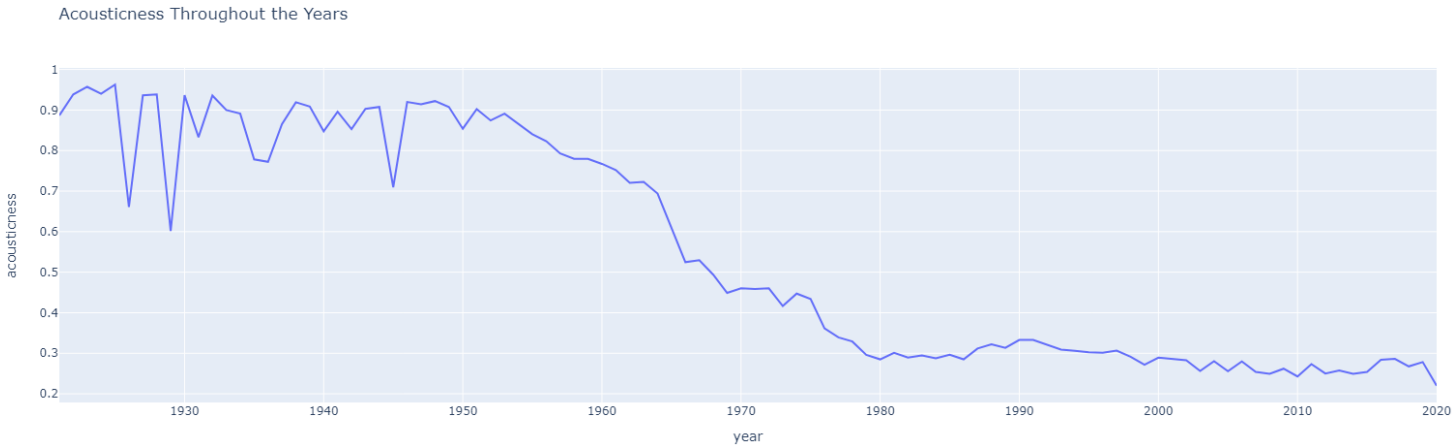
Year: describes the year in which the track was released

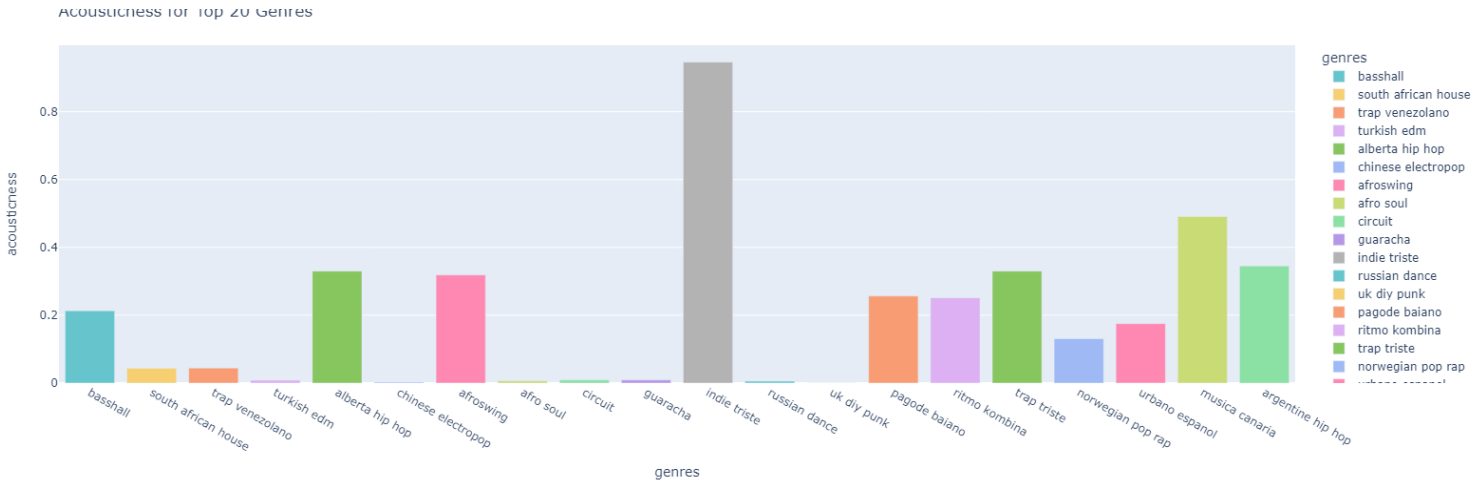


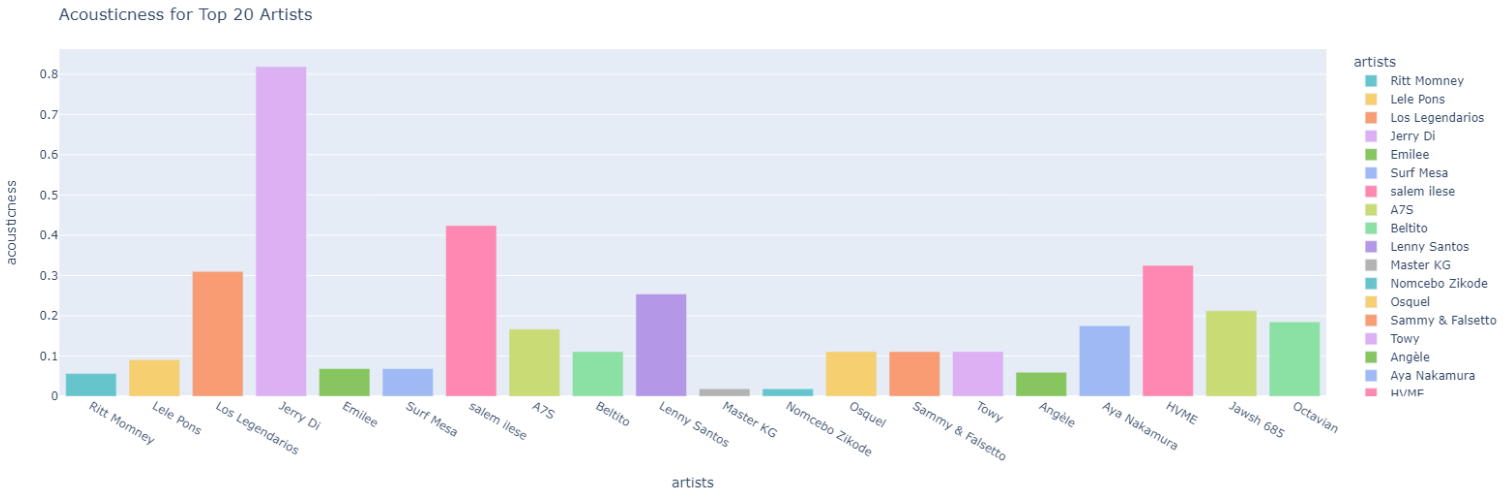




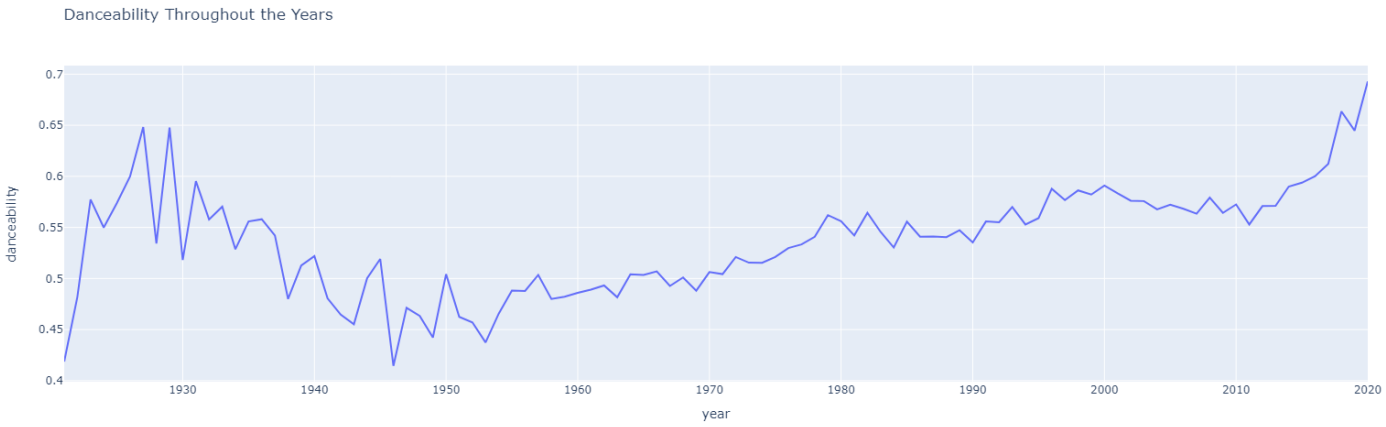
<u>Acousticness</u>: describes if track is acoustic; measurement ranges from 0.0 to 1.0 (higher value is better confidence)

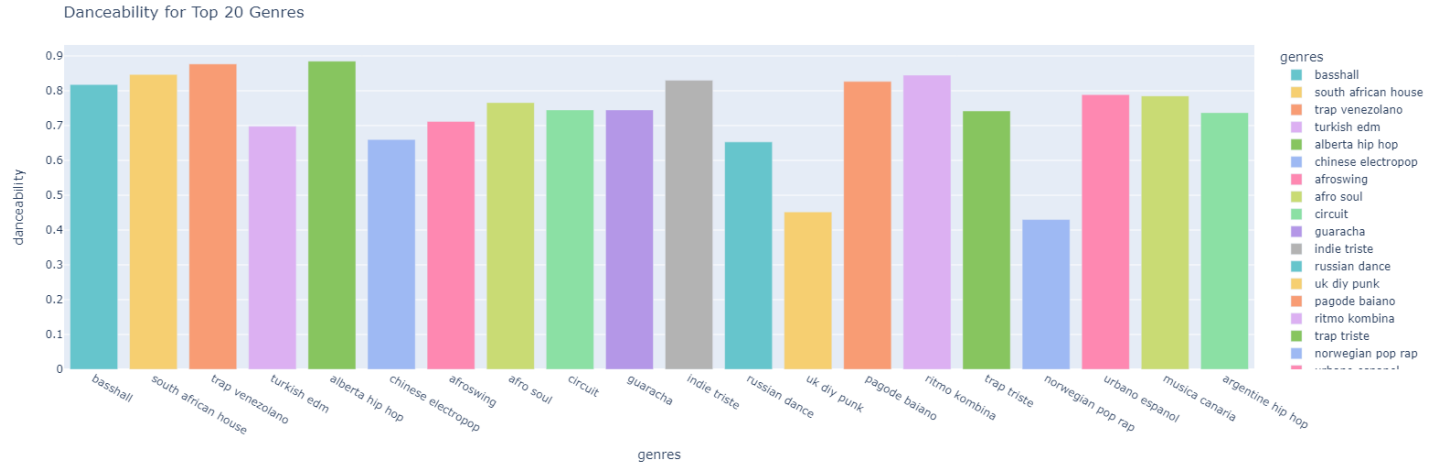


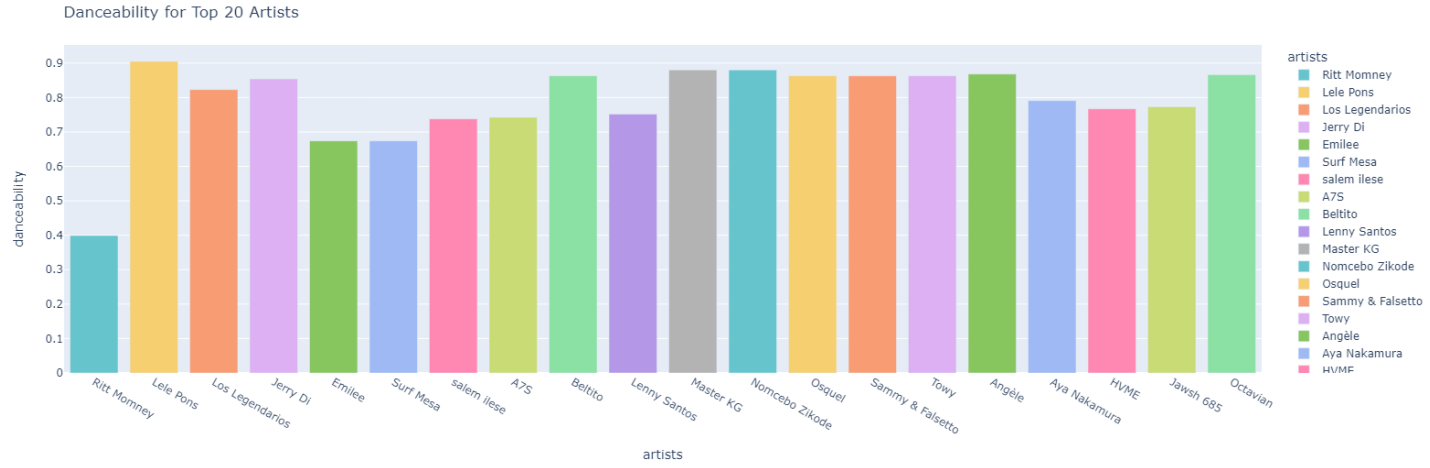




Danceability: describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity; measurement ranges from 0.0 to 1.0

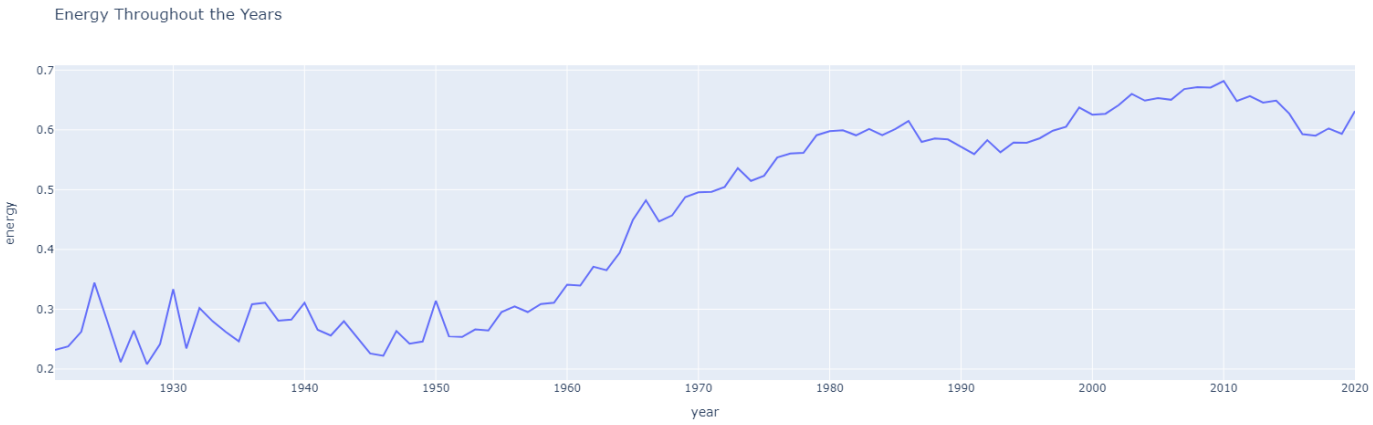


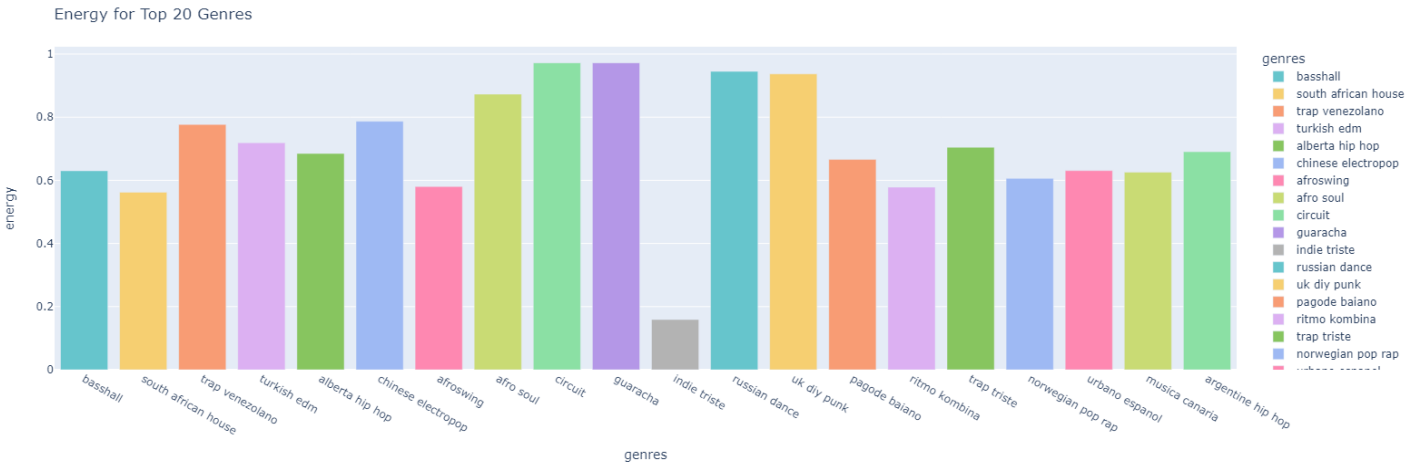


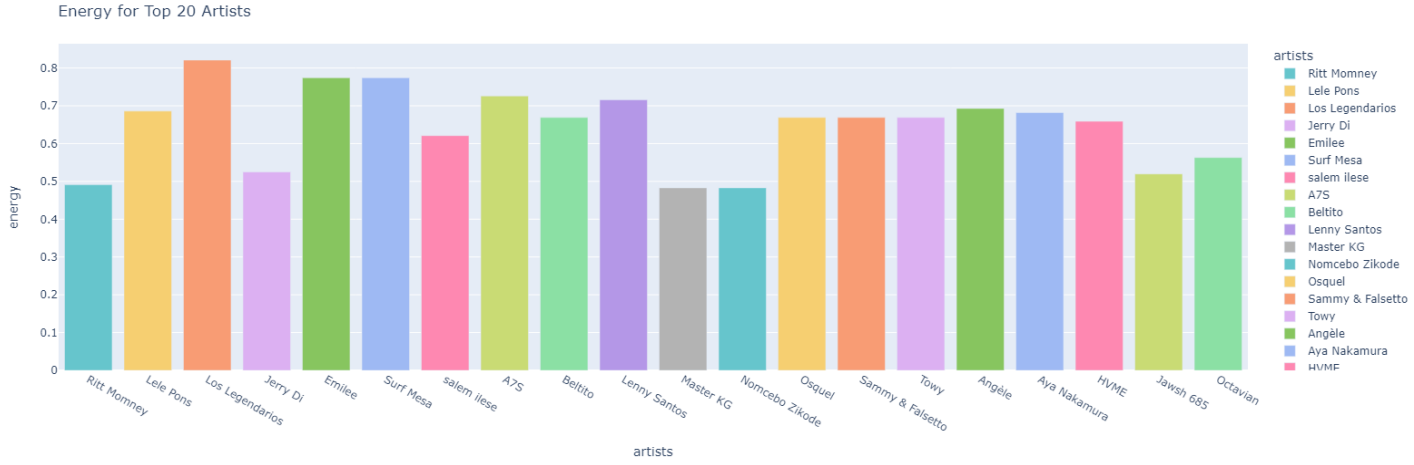


Duration_ms: describes the length of song in microseconds

Energy: describes the perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy; measurement ranges from 0.0 to 1.0; 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.

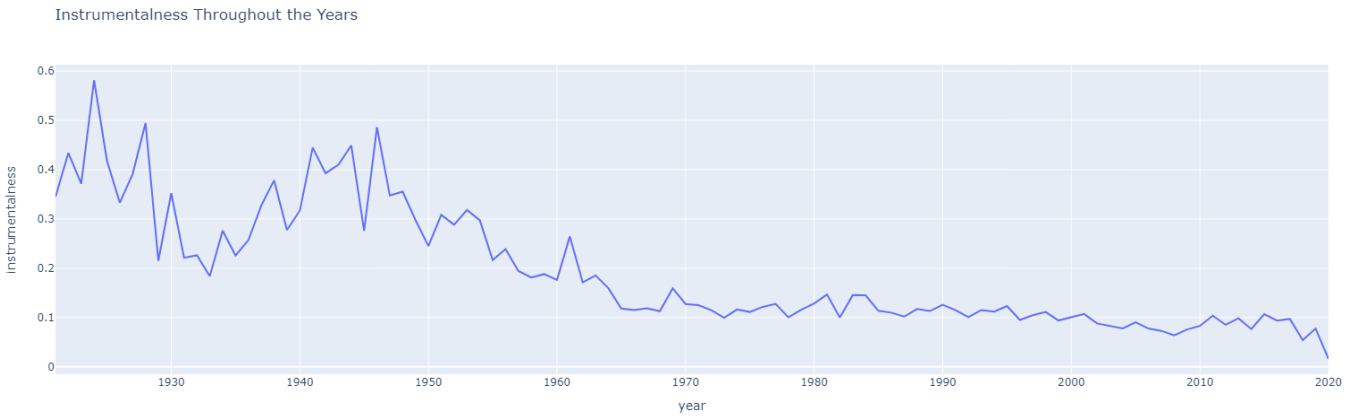


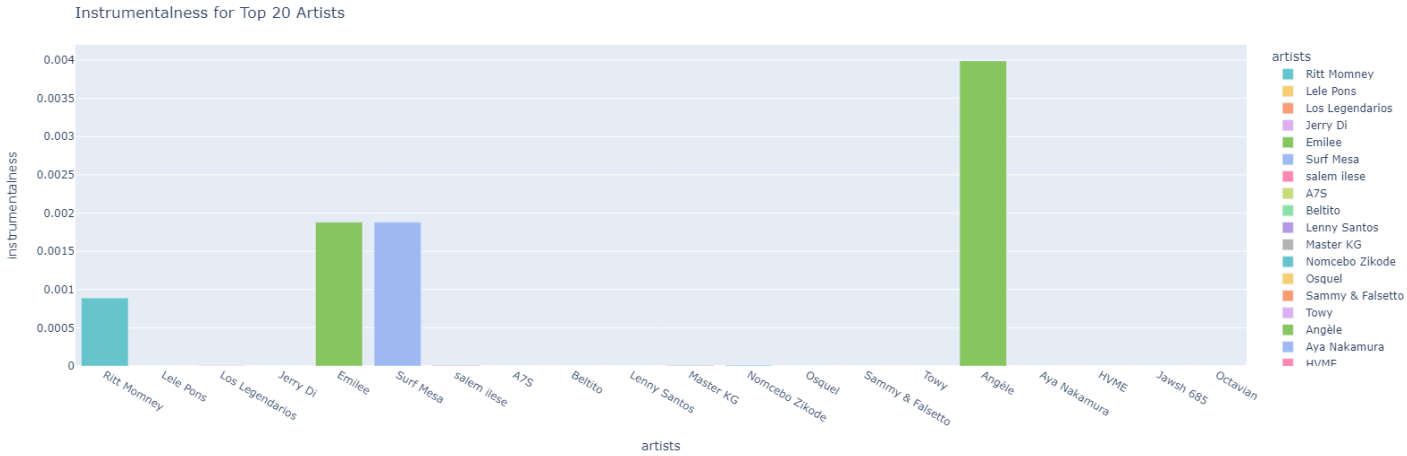


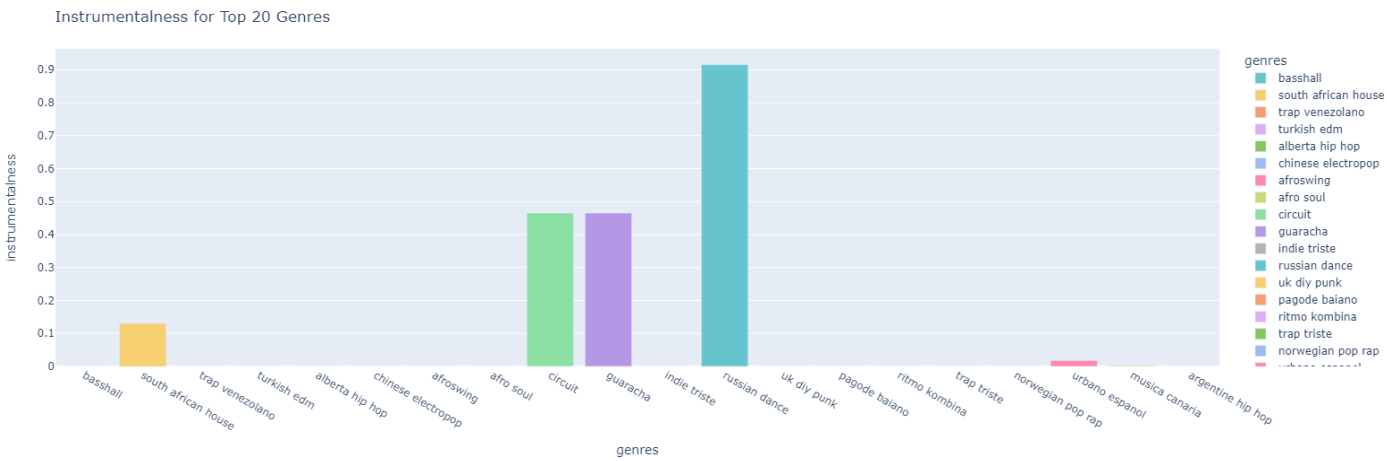


Explicit: describes if the track is explicit; boolean measurement values: 1 = Explicit; 0 = Not Explicit

Instrumentalness: predicts if a track has no vocals (including “Ooh” and “aah” sounds). Rap or spoken word tracks are clearly “vocal”. The closer the instrumentalness value is to 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: describes the overall key of the track. 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.

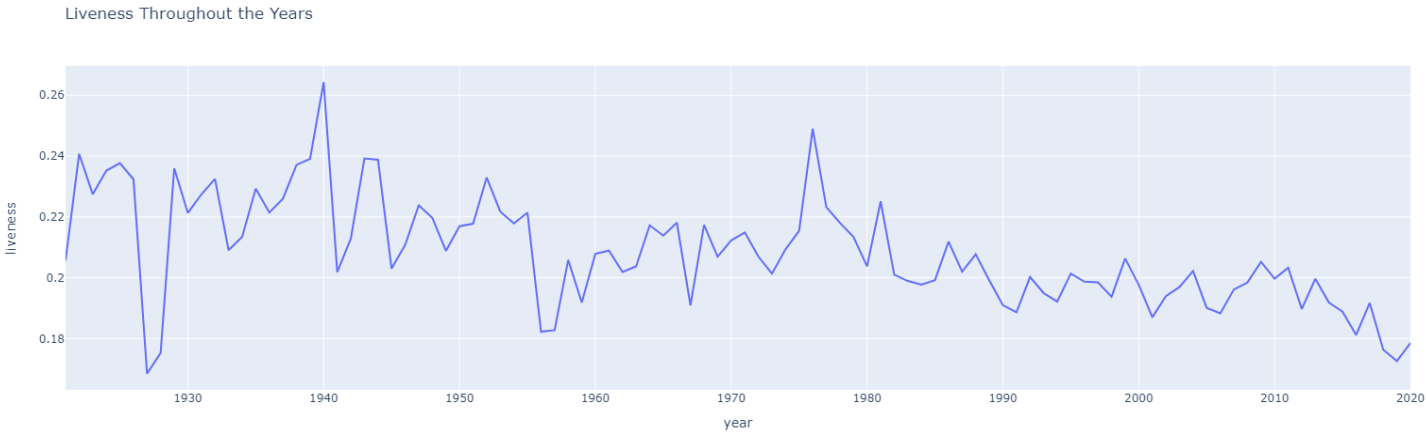
Loudness: The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically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

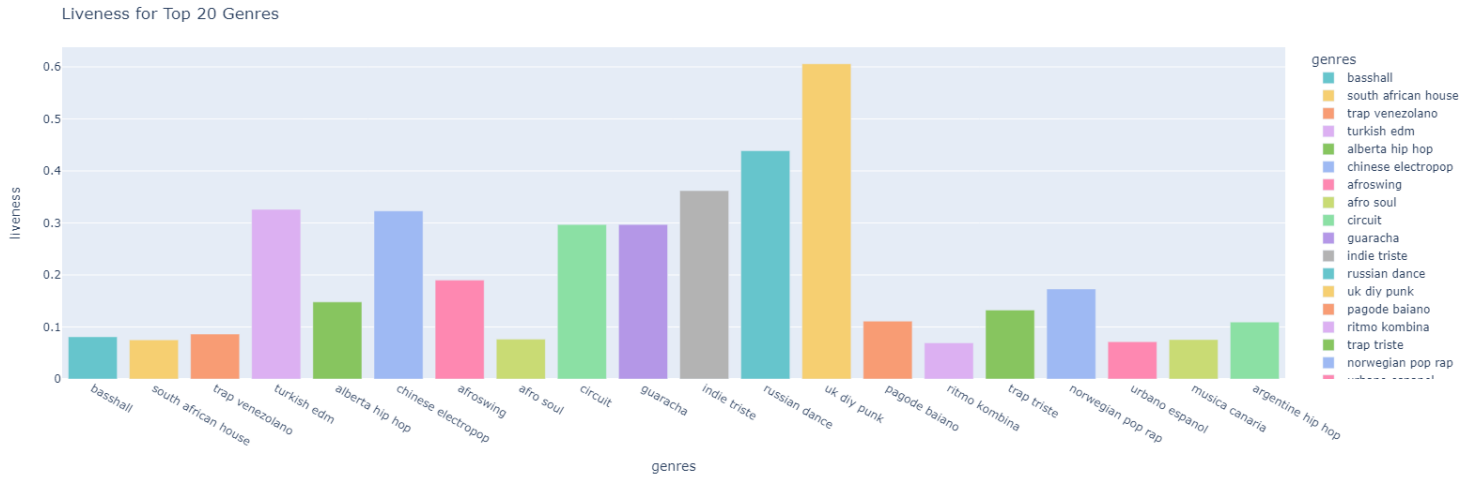
Mode: Indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.

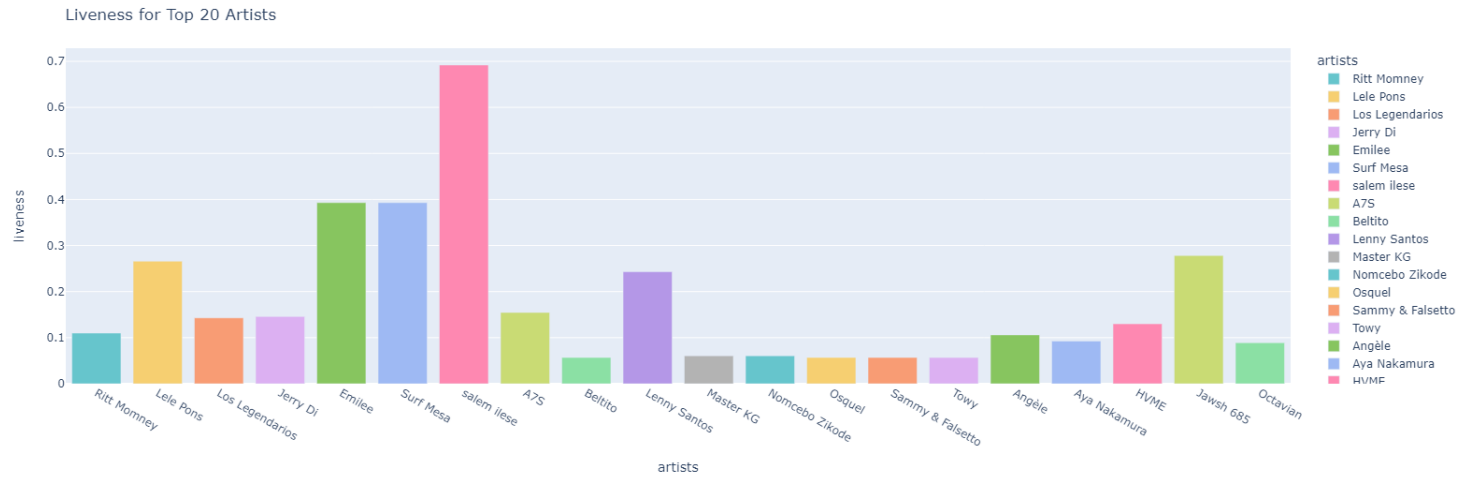
Popularity: The popularity of the track. The value will be between 0 and 100, with 100 being the most popular. The popularity is calculated by algorithm and is based, in the most part, on the total number of plays the track has had and how recent those plays are. Generally speaking, songs that are being played a lot now will have a higher popularity than songs that were played a lot in the past. Artist and album popularity is derived mathematically from track popularity. Note that the popularity value may lag actual popularity by a few days: the value is not updated in real time.

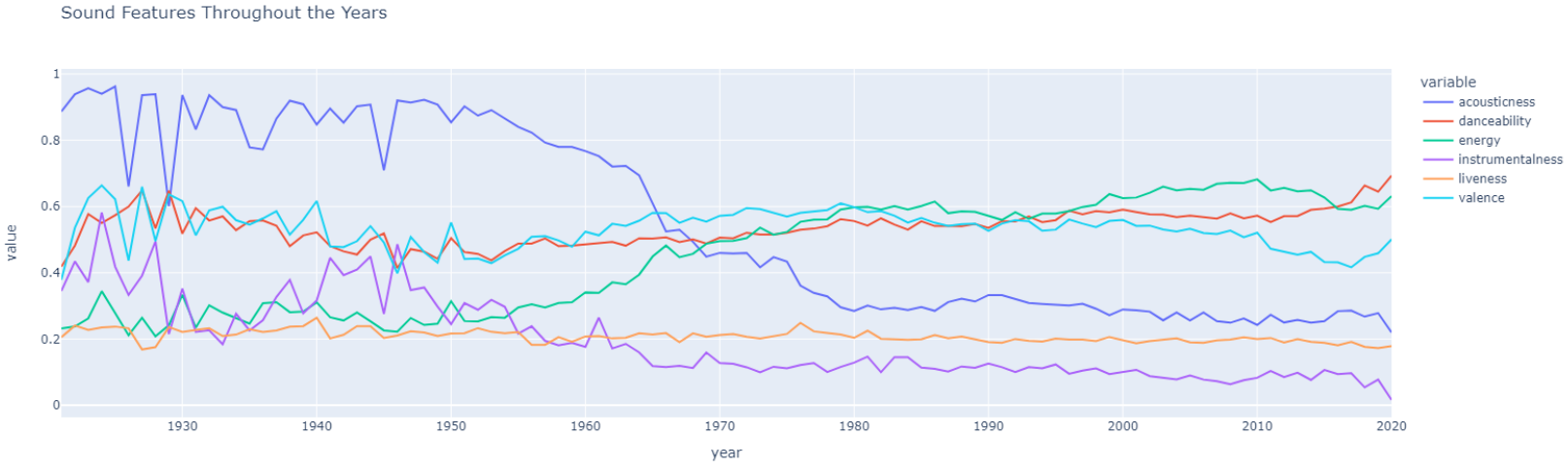
Speechiness: Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 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.

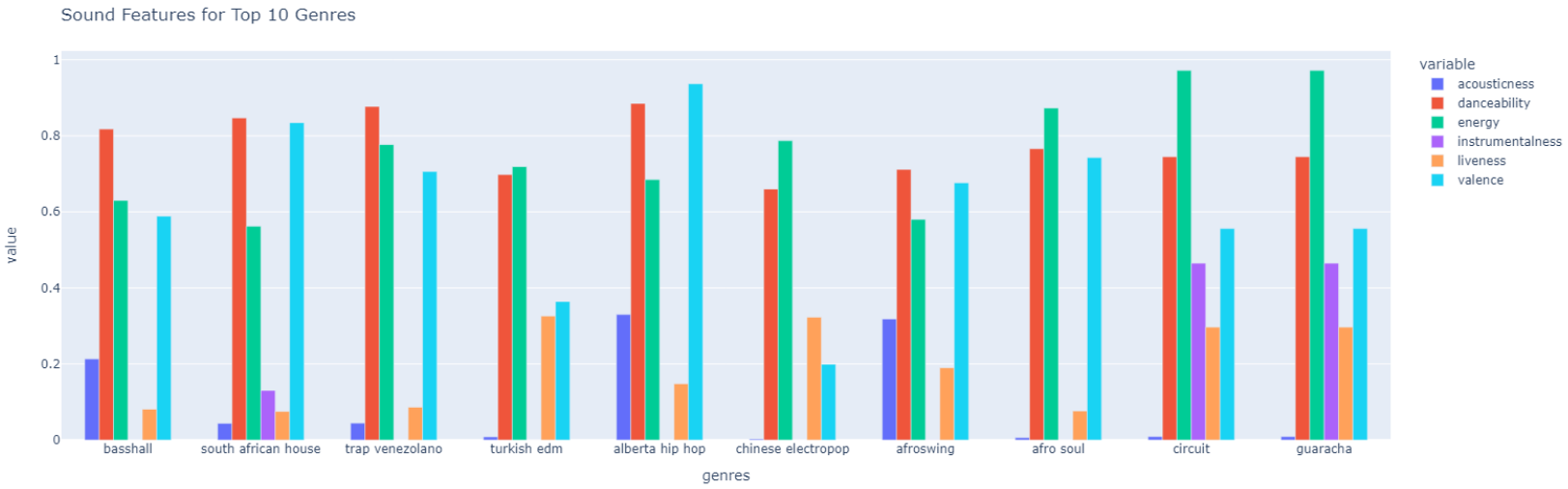
Tempo: The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.

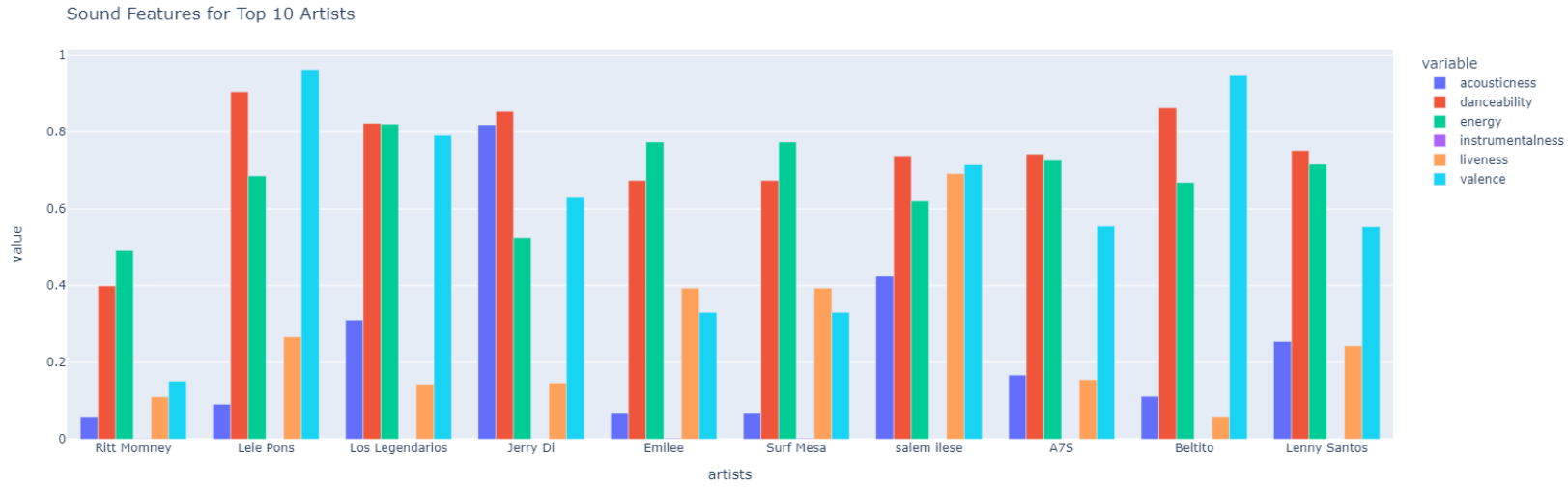



Liveness for Top 20 Genres
liveness
genres
basshall
south african house
trap venezolano
turkish edm
alberta hip hop
chinese electropop
afroswing
afro soul
circuit
guaracha
indie triste
russian dance
uk diy punk
pagode baiano
ritmo kombina
trap triste
norwegian pop rap
urbano espanol
musica canaria
argentine hip hop


Liveness for Top 20 Artists
liveness
artists
Ritt Momney
Lele Pons
Los Legendarios
Jerry Di
Emilee
Surf Mesa
salem ilese
A7S
Beltito
Lenny Santos
Master KG
Nomcebo Zikode
Osquel
Sammy & Falsetto
Towy
Angèle
Aya Nakamura
HVME
Jawsh 685
Octavian


Sound Features Throughout the Years
value
year
variable
acousticness
danceability
energy
instrumentalness
liveness
valence





Our column for our models will vary here, which you can read about right below this in the analysis sections for the dimension reduction analysis.

FeatureAgglomeration Reduction - Konrad

The FeatureAgglomeration class in scikit-learn is a tool for unsupervised dimensionality reduction that focuses on clustering together similar features in a dataset. It is particularly useful when dealing with high-dimensional data, as it uses “agglomerative clustering to group together features that look very similar, thus decreasing the number of features.” as provided by the scikit learn User Guide on Hierarchical Clustering. This brings a reduction in the number of features, making the data more manageable and potentially improving the performance of machine learning algorithms, especially when dealing with multicollinearity, which can potentially be the case with some of our features as they can potentially overlap somewhat in what they represent, such as with energy and tempo in our dataset. For the purpose of this model, which is called nn_FAR on our project's github, I kept all of the parameters as the default except the n_clusters parameter, which denotes the amount of features we want the algorithm to reduce our current ones to.

Since we are doing dimensionality reduction through a sklearn class, we can simply get started by doing the basic preprocessing of our dataset that was mentioned previously in this report, where we get rid of any existing duplicate rows or null values in our dataset and then drop all of the columns which are not found in the audio features we want to extract from the spotify song input. We do this for both the dataset and the song input values, so that the columns in each are the same. We then standardize the scale of our data using StandardScaler() so that we can scale to our unit variance, which can help with features that have different ranges, as it helps prevent certain features from dominating the others. We can see the before and after of applying the StandardScaler() to our data and song input feature below. Keep in mind the data is only displaying the first 10 entries:

Before Scaling:

Data and Song Input:

	valence	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	0.0594	0.982	0.279	831667	0.211	0.878000	10	0.665	-20.096	1	0.0366	80.954
1	0.9630	0.732	0.819	180533	0.341	0.000000	7	0.160	-12.441	1	0.4150	60.936
2	0.0394	0.961	0.328	500062	0.166	0.913000	3	0.101	-14.850	1	0.0339	110.339
3	0.1650	0.967	0.275	210000	0.309	0.000028	5	0.381	-9.316	1	0.0354	100.109
4	0.2530	0.957	0.418	166693	0.193	0.000002	3	0.229	-10.096	1	0.0380	101.665
5	0.1960	0.579	0.697	395076	0.346	0.168000	2	0.130	-12.506	1	0.0700	119.824
6	0.4060	0.996	0.518	159507	0.203	0.000000	0	0.115	-10.589	1	0.0615	66.221
7	0.0731	0.993	0.389	218773	0.088	0.527000	1	0.363	-21.091	0	0.0456	92.867
8	0.7210	0.996	0.485	161520	0.130	0.151000	5	0.104	-21.508	0	0.0483	64.678
9	0.7710	0.982	0.684	196560	0.257	0.000000	8	0.504	-16.415	1	0.3990	109.378

	valence	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	0.0884	0.379	0.545	218708	0.635	0.0579	0	0.158	-7.874	1	0.0259	90.005

After Scaling:

Data and Song Input:

	valence	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	-1.782825	1.276187	-1.467013	4.763146	-1.013988	2.268102	1.365588	2.626719	-1.514237	0.643912	-0.379706	-1.169307
1	1.650688	0.611347	1.598779	-0.399747	-0.528270	-0.532771	0.512123	-0.262229	-0.170766	0.643912	1.945481	-1.821180
2	-1.858821	1.220340	-1.188820	2.133824	-1.182122	2.379754	-0.625830	-0.599749	-0.593551	0.643912	-0.396297	-0.212404
3	-1.381564	1.236296	-1.489722	-0.166101	-0.647832	-0.532682	-0.056853	1.002043	0.377680	0.643912	-0.387080	-0.545537
4	-1.047180	1.209703	-0.677855	-0.509485	-1.081242	-0.532765	-0.625830	0.132499	0.240788	0.643912	-0.371104	-0.494867
5	-1.263770	0.204465	0.906137	1.301381	-0.509589	0.003159	-0.910318	-0.433849	-0.182173	0.643912	-0.174471	0.096469
6	-0.465809	1.313418	-0.110116	-0.566464	-1.043879	-0.532771	-1.479295	-0.519660	0.154265	0.643912	-0.226701	-1.649077
7	-1.730767	1.305440	-0.842500	-0.096539	-1.473552	1.148391	-1.194806	0.899071	-1.688862	-1.553007	-0.324403	-0.781368
8	0.731133	1.313418	-0.297470	-0.550503	-1.316628	-0.051072	-0.056853	-0.582587	-1.762046	-1.553007	-0.307812	-1.699324
9	0.921124	1.276187	0.832331	-0.272668	-0.842119	-0.532771	0.796611	1.705688	-0.868212	0.643912	1.847164	-0.243698

	valence	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	-1.67263	-0.327406	0.043174	-0.097054	0.5702	-0.348066	-1.479295	-0.27367	0.630754	0.643912	-0.445456	-0.874567

Now that our data is scaled, we can confidently fit and transform our standardized data to the FeatureAgglomeration() class, set to 4 clusters, resulting in a set of data with 4 columns in it. We also transform our song input features with the class, resulting in both now having 4 columns instead of 12:

```
agglo = FeatureAgglomeration(n_clusters = 4)

X_agglo = agglo.fit_transform(X_standard)
song_agglo = agglo.transform(song_standard)
```

Data (Left) and Song Input (Right)

	0	1	2	3
0	1.803932	-1.232511	1.772144	-1.624919
1	0.487908	-0.840072	0.039288	1.624734
2	0.231172	-0.662692	1.800047	-1.523821
3	0.207184	-0.271896	0.351807	-1.435643
4	-0.146002	-0.445107	0.338469	-0.862518
5	0.085331	-0.198431	0.103812	-0.178816
6	-0.429641	-0.846230	0.390324	-0.287962
7	-0.453937	-1.314594	1.226915	-1.286633
8	-0.610153	-1.592666	0.631173	0.216832
9	0.944142	-0.651343	0.371708	0.876727

	0	1	2	3
0	-0.330313	0.108796	-0.337736	-0.814728

While we can see that our features have been reduced, however, it is a bit confusing to know what exactly these 4 new columns represent, so taking a closer look at this would be useful for helping us come to some conclusions. We can use the labels_ attribute to return an array of cluster labels for each feature, which essentially tells us which feature was clustered into which agglomerated column. We can use np.where() to extract the index positions of each of the cluster labels in the labels_ array through a for loop, which we can then use to append all of the column names associated with said indexes into a new array, and in this way we can fairly simply see the column names that were agglomerated into each cluster, as seen below:

```
Cluster 0:  Feature Names Index(['duration_ms', 'key', 'liveness', 'mode', 'speechiness'], dtype='object')
Cluster 1:  Feature Names Index(['energy', 'loudness', 'tempo'], dtype='object')
Cluster 2:  Feature Names Index(['acousticness', 'instrumentalness'], dtype='object')
Cluster 3:  Feature Names Index(['valence', 'danceability'], dtype='object')
```

Here we can see which of our original features were agglomerated into which cluster, our agglomerated columns. Clusters 1-3 seem to be fairly accurate, the features paired together have a fair amount of potential overlap, so we can see the functionality of FeatureAgglomeration in action. Cluster 0, however, seems to contain quite a few features that do not have much related to each other. It is difficult to think of a significant overlap between duration_ms and mode or speechiness for instance. This could potentially mean that FeatureAgglomeration did not have as much confidence in combining the features in Cluster 0 with any other features.

To do some more testing will perform the same process for a FeatureAgglomeration of 5 clusters instead of 4, to see if the class will maybe more evenly spread out the features with another cluster to work with:

```
Cluster 0: Original Features Index(['duration_ms', 'key', 'liveness', 'speechiness'], dtype='object')
Cluster 1: Original Features Index(['energy', 'loudness', 'tempo'], dtype='object')
Cluster 2: Original Features Index(['acousticness', 'instrumentalness'], dtype='object')
Cluster 3: Original Features Index(['valence', 'danceability'], dtype='object')
Cluster 4: Original Features Index(['mode'], dtype='object')
```

As we can see, the agglomeration did not really improve for the same features, the only change was that mode was separated from Cluster 0, and instead was put into Cluster 4 on its own. At this point, it has become relatively clear that the features duration_ms, key, liveness, speechiness, and mode are probably not as influential in the dataset as the remaining features. Getting rid of them could potentially end up improving the model. After making a new dataframe, with the features mentioned dropped, and performing the previous steps we have done to prepare our new dataframe, we get the following 4 cluster agglomeration:

```
Cluster 0: Original Features Index(['acousticness', 'instrumentalness'], dtype='object')
Cluster 1: Original Features Index(['energy', 'loudness'], dtype='object')
Cluster 2: Original Features Index(['valence', 'danceability'], dtype='object')
Cluster 3: Original Features Index(['tempo'], dtype='object')
```

We can see that our new agglomerated data is much more balanced then the previous iterations, and because of this, it will be this version of data and dimensionality reduction that we perform for my final KNN model with FeatureAgglomeration.

Factor Analysis - Alan

Factor Analysis is a data analysis method that is used to see which variables are influential and look for variance between variables. Here factor analysis was used for one of the K-nearest neighbors models, specifically the model called nn_model on our github. Factor analysis works by reducing the number of observed variables and helps find unobserved variables. It does this in 2 steps; datacamp has a good explanation to this

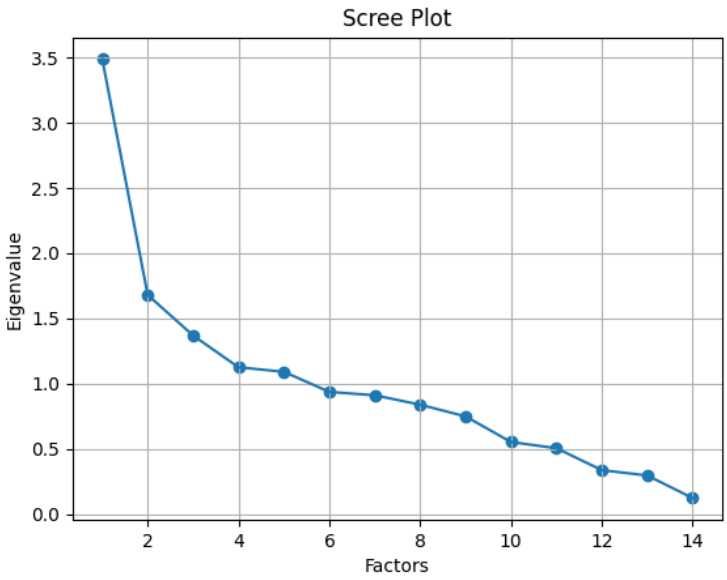
- **Factor Extraction:** In this step, the number of factors and approach for extraction selected using variance partitioning methods such as principal components analysis and common factor analysis.
- **Factor Rotation:** In this step, rotation tries to convert factors into uncorrelated factors — the main goal of this step to improve the overall interpretability. There are lots of rotation methods that are available such as: Varimax rotation method, Quartimax rotation method, and Promax rotation method.

Figure: Introduction to Factor Analysis on datacamp

A "factor" in factor analysis is a variable that describes the associations in the given observed variables. Each factor shows variance between observed variables, and the factor that has the lowest variance can be dropped. First we need to drop redundant columns and once that is done we need to use an adequacy test. I used the Bartlett's test. Bartlett's test checks if the observed variables matrix have a connection to the identity matrix, if they do we can't use factor analysis. In python I use calculate_bartlett_sphericity in the factor_analyzer library.

The result from that is:
```
687141.2317599686 0.0
```
The second value 0.0 known as the p-value is 0 which indicates that the observed matrix is not the identity matrix, thus we can proceed. Next we need to use the Kaiser-Meyer-Olkin (KMO) test. This will show us if the data is suitable for factor analysis. It is going to determine if the data is adequate for the model. We are looking for a number between 0.60 and 1 When running calculate_kmo in the factor analysis library we get:
```
kmo model:  0.658586753152299
```
So a model of 0.65 is not the best but barely enough. Now we need to figure out the number of factors needed for factor analysis. We will use both Kaiser criterion and a scree plot.
The result from Kaiser criterion:
```
[3.48900429 1.68056238 1.36923886 1.12531066 1.09070791 0.9365853
 0.91085168 0.83863172 0.74842385 0.55202029 0.50474024 0.33644305
 0.29544306 0.12203673]
```

Observe that we have 5 values that are greater than 1.This means that we only need 5 factors for our analysis. For the scree plot:



We see that after 5 it begins to repeat so no significance after 5. We can now begin our factor analysis.

	Factor 1	Factor 2	Factor 3	Factor 4	Factor 5
valence	0.265486	0.826311	-0.146579	0.080849	0.127846
acousticness	-0.768905	0.027210	-0.214750	0.071448	-0.195947
danceability	0.218744	0.652108	0.287320	-0.299167	0.200966
duration_ms	0.052299	-0.260115	-0.054599	0.003735	0.078729
energy	0.942459	0.068490	-0.055115	0.228511	0.241349
explicit	0.141708	0.015112	0.515034	-0.038500	0.178235
instrumentalness	-0.434187	-0.231666	-0.324773	0.056685	0.415625
key	0.016274	0.019074	0.015699	-0.000229	0.063166
liveness	0.066350	-0.044447	0.059381	0.312146	-0.033709
loudness	0.793997	0.154717	0.007610	0.010927	-0.039472
mode	-0.014069	0.033434	-0.065769	0.028370	-0.169248
popularity	0.628643	-0.139185	0.148731	-0.388788	0.034208
speechiness	-0.190751	0.188636	0.810948	0.392740	0.202433
tempo	0.258325	0.063238	-0.037101	0.099374	-0.007580

We can see here that factor 1 has high loadings with energy and loudness which could mean factor 1 is representing energetic loud music. Factor 2 has high loadings with valence and danceability which can mean a relation between these 2 variables. Factor 3 has high loadings with speechness and explicit. Factor 4 does not have any and because of that cant be interpreted. Factor 5 also does not have any high loadings. With this in mind I will reduce the factors to 3

	Factor 1	Factor 2	Factor 3
valence	0.283363	0.931353	-0.222717
acousticness	-0.783412	-0.007509	-0.282643
danceability	0.247447	0.561840	0.230198
duration_ms	0.058942	-0.248287	-0.021900
energy	0.879001	0.097635	0.066656
explicit	0.116998	0.092808	0.602437
instrumentalness	-0.374618	-0.183465	-0.197311
key	0.018198	0.027694	0.027677
liveness	0.026095	-0.012898	0.091867
loudness	0.802578	0.130911	0.029733
mode	-0.019855	0.007469	-0.101653
popularity	0.629220	-0.116833	0.131961
speechiness	-0.220042	0.281125	0.685615
tempo	0.252291	0.066904	-0.025291

Now we get the variance of each factor

	Factor 1	Factor 2	Factor 3
SS Loadings	2.838929	1.411876	1.097809
Proportion Var	0.202781	0.100848	0.078415
Cumulative Var	0.202781	0.303629	0.382044

This tells us that we have a total cumulative variance of 38% using 3 factors. This is on the low side, so let's examine our variables a bit. We can see that in factor 1 duration_ms, key,liveness, and mode have low loadings. In factor 2 acousticness has low loadings but we believe this variable is important to the dataset so it will remain. Finally factor 3 has low loadings with duration. Let’s try dropping these columns and see what we get:

	Factor 1	Factor 2	Factor 3
valence	0.151163	0.989951	-0.048941
acousticness	-0.801597	-0.061937	-0.198022
danceability	0.215366	0.525852	0.334551
energy	0.861427	0.206079	-0.019908
explicit	0.182168	-0.019937	0.612734
instrumentalness	-0.380475	-0.173486	-0.195800
loudness	0.784448	0.221903	-0.030226
popularity	0.653894	-0.067013	0.054228
speechiness	-0.157467	0.114898	0.706771
tempo	0.240920	0.099333	-0.041350

	Factor 1	Factor 2	Factor 3
SS Loadings	2.757567	1.410122	1.072798
Proportion Var	0.275757	0.141012	0.107280
Cumulative Var	0.275757	0.416769	0.524049

Explicit has low loadings for both Factor 1 and Factor 2 so we may consider dropping it. Tempo and valence also have low loadings for factor 1 and factor 3, may consider dropping tempo. As you can see however out cumulative variance increased to 52%

	Factor 1	Factor 2	Factor 3
valence	0.123131	0.968273	0.210081
acousticness	-0.812357	-0.042949	-0.126769
danceability	0.237761	0.397934	0.624625
energy	0.864038	0.287226	-0.129531
instrumentalness	-0.405214	-0.088389	-0.276292
loudness	0.768549	0.264219	-0.047078
popularity	0.654337	-0.073386	0.034050
speechiness	-0.086170	-0.003926	0.413492
tempo	0.232472	0.143592	-0.089822

	Factor 1	Factor 2	Factor 3
SS Loadings	2.722669	1.283891	0.725895
Proportion Var	0.302519	0.142655	0.080655
Cumulative Var	0.302519	0.445173	0.525828

A slight increase in cumulative variance after dropping the explicit column.

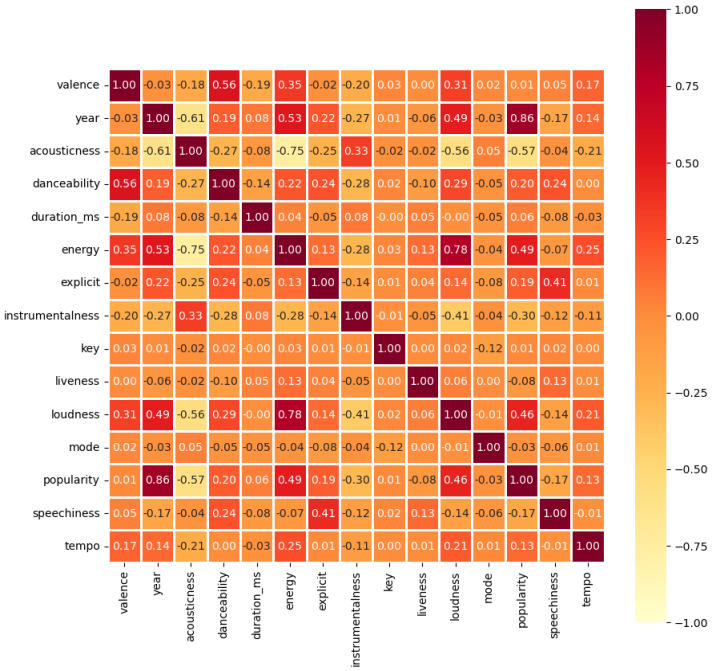
Feature Extraction and utilizing K-means clustering - Anthony

What is k-means clustering?

- K-means clustering is a form of unsupervised learning technique in machine learning, which allows an algorithm to make inferences about data using only the input vectors without knowing the labeled outcomes.
- The objective for K-means is to group similar data points together and since this project is a Spotify song recommendation application, we can utilize K-means to select songs from the same input cluster.
- In K-means there is a fixed number of clusters that must be defined in order to group each data point into their respective clusters.
- The more clusters that there are in the model the more accurate the model will be.

For my K-means clustering model, I selected all the features such as valence, acousticness, danceability, energy, instrumentalness, loudness, popularity, speechiness, and tempo. With these features that were extracted from the dataset, I fit them into a K-means model and classified all the songs into which cluster they belonged in. For the application I only defined 10 clusters for the sake of performance, as the current structure of the application requires the model to be re-run each time the user inputs a new song into the application for the model to predict. With a larger amount of clusters we can expect greater accuracy however, since the current state of the application requires a re-run each time. In order to counteract this downside of the application, I can utilize a package called pickle to save a model that I have already trained and prevent the performance hit each time I were to run a K-means model that has increased numbers of clusters.

When I was doing my analysis for the data values in the dataset, I took a sample of the data and created a correlation matrix to determine the possible relationships between each of the columns of data.



From the heatmap we can see that some of the values have higher correlation such as year and popularity. This suggests that for a future model, the year in which a song is released could be weighted higher for when creating the clusters that categorizes the songs in the data set.

Low Variance Filter - Daphne

I used the dimensionality reduction technique known as low variance filter to help filter for feature selection. This method filters the list of features I will use in my K-Means clustering model.

Variance describes the spread of the data and tells us how far the points are from the mean. This is the formula to calculate variance:

$$\sigma^2 = \frac{\Sigma(x-\bar{x})^2}{n}$$

To implement low variance filter we start with importing the necessary libraries:

```
from sklearn.preprocessing import normalize
```

I also have to check for the nulls in our dataset (shown on the right). I dropped some unnecessary features from the dataframe.

```
# dropping unnecessary columns
df_features = df.drop(columns=['id','name','release_date','explicit','artists','popularity','year','duration_ms'], axis=1)
```

Then I have to normalize the dataset. If the variables in a dataset have a high range, it could skew the results.

```
normalized_df = normalize(df_features)
```

I saved the normalized values of the dataset and created a dataframe. Then I used the .var() function to calculate the variance values:

```
# save the result in a data frame called data_scaled
df_scaled = pd.DataFrame(normalized_df)
# .var() function calculates the variance
variance = df_scaled.var()
```

This is the resulting variance values for the features:

```
0     0.000006
1     0.000017
2     0.000004
3     0.000007
4     0.000012
5     0.001282
6     0.000004
7     0.005072
8     0.000023
9     0.000003
10    0.000977
```

```
Nulls?:
valence             0
year                0
acousticness        0
artists             0
danceability        0
duration_ms         0
energy              0
explicit            0
id                  0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
name                0
popularity          0
release_date        0
speechiness         0
tempo               0
```

I used a for loop that will traverse through the variance values and filter out the variance value lower than 0.000005 and the corresponding feature. Features with variance values greater than or equal to 0.000005 will be added to a list called variable.

```
variable = [ ]

for i in range(0,len(variance)):
    if variance[i]>=0.000005: #sett
        variable.append(columns[i])
```

The resulting list (below) are the features selected from low variance filter.

```
['valence',
 'acousticness',
 'energy',
 'instrumentalness',
 'key',
 'loudness',
 'mode',
 'tempo']
```

I created a dataframe containing only those features to use in my K-means model.

```
# creating a new dataframe using the above variables (after low variance filter)
df_new = df_features[variable]
```

This shows the comparison of before and after the low variance filtering technique:

```
    # compare shape of new and original data
    df_new.shape, df_features.shape
```

```
((170653, 8), (170653, 11))
```

Summary

In summary, we discussed a problem and our approach to solving this problem. We showed how we pre-processed our dataset and how we picked which columns to keep and which to drop, discussed 2 machine learning models; k-means and k-nearest neighbors, which we are using from the sklearn library in python and we used and explained 4 different dimensionality reduction techniques that we use on our datasets before using the models.

Conclusion

In conclusion, our problem that we seeked to solve is finding music with similar characteristics to the ones we listen to. We solved this by using a dataset with information on the aesthetic of songs and using dimension reduction techniques and machine learning models; k-nearest neighbors and kmeans, we are able to obtain songs with similar characteristics thus solving our problem. In the course of this project we have learned the following:

- Models: Kmeans-Clustering, and K-Nearest Neighbors
- Dimension reduction techniques: Factor Analysis, Feature Agglomeration Reduction, Feature Extraction/Selection, and Low Variance Filter

Going forward, what we could do with this project is extend our datasets and use a more extensive dataset on determining song outputs, thus taking a more supervised approach to this problem.

References

Introduction to Factor Analysis in Python
Plotly Documentation on Discrete Colors
Beginner's Guide to Low Variance Filter
Spotify Music Recommendation Kaggle Notebook (for inspiration on visualizations)
Sklearn Kmeans Documentation
Sklearn Kmeans Description from Documentation
Principal Component Analysis in Python
Ultimate Guide to Dimensionality Reduction Techniques
Guide to Building Song Recommendation with Spotify
Another Guide to Building Song Recommendation with Spotify
NearestNeighbors Class Documentation by scikit learn
NearestNeighbors User Guide by scikit learn
Hierarchical Clustering User Guide by scikit learn