Final Report - Group 69

02807 Computational Tools for Data Science

Authors

Cristobal Donoso - s222508
Oscar Parsholt Beck - s223103
Elysia Livia Gao - s222445
Ifigeneia Tziola - s222569

Technical University of Denmark (DTU)

Contents

1 Defining a Spotify Recommendation System

1.1 Problem Definition & Motivation

With the vast increase in data availability and users online, recommendations systems have become a significant scientific, economic, and industry interest. Recommendation systems provide users with personalized content, increasing user satisfaction on platforms. Most notably, recommendation systems have become an important form of content filtering, reducing content overload on the end of the users and alleviating the challenge of "paralysis of choice" for the user. However, issues such as low efficiency or time consuming algorithms, the cold start problem (difficulty in initially analyzing data with low number of users), and user privacy concerns exist [Isinkaye, 2015]. Further, we made consideration for balancing exploration vs. exploitation as in we wanted song recommendations that were both similar to the user's last liked song but were not as to be so similar so that the user could discover new music. We also crafted our algorithm so that newer item profiles or unknown songs can be recommended just as easily or often as newer songs.

Although Spotify's algorithm is most likely a combination of content based and collaborative based filtering (evaluating user interactions), we chose to focus our project on a content based approach. The content based filtering evaluates similarity in song features rather than similarity in interactions/with other users. Our motivation in doing so was based on available data (user data may be difficult to obtain) and the various advantages of content based filtering such as recommendations may be more relevant to the user and the opportunity to evaluate many attributes related to each song [Team, 2021].

In this project, we propose a Spotify song recommendation system using an ensemble of four main content based methods: (1) K-Means Clustering, (2) Decision Trees, (3) Cosine Similarity, and (4) Locality Sensitivity Hashing. We obtained two Spotify datasets from Kaggle with the first consisting of 170.653 songs with 19 song features and a second dataset containing the lyrics data for 18.454 songs [Mavani], [Nakhaee]. We chose to focus on 14 various quantitative song features (valence, acousticness, danceability, duration, energy, explicit, instrumentalness, key, liveness, loudness, mode, popularity, speechiness, and tempo) which were the inputs of the first three methods. The locality sensitivity hashing (LSH) used lyrical content to determine similarity between songs and suggest a song recommendation. Firstly, we performed data pre-processing (removing non-quantitative data and standardization of attribute values) as well as PCA (principal component analysis) for dimensionality reduction before using our data for the three mentioned methods. We chose the first two methods in order to obtain song recommendations with a greater song diversity as songs may be recommended from the closest cluster whereas the last two methods are based on similarity measures which would should result in much closer recommendations.

We developed a parallel hybrid recommendation system that takes song recommendations from four different types of data science techniques (K-Means Clustering, Decision Trees, Cosine Similarity, and Locality Sensitivity Hashing) and combines them using ensemble majority voting which takes the most commonly recommended songs between all methods to make the final song recommendations. See figure 1.1 [Vatsal, 2022] as reference. This combination of methods creates an overall more robust system which will reduce the weaknesses of individual models.

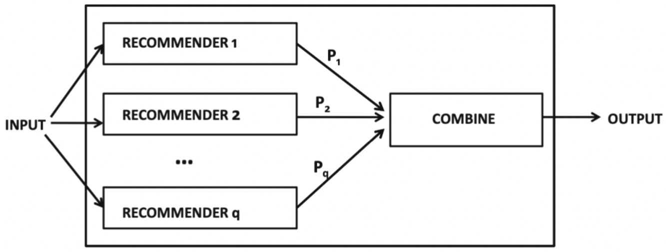


Figure 1.1: The figure depicts a parallel hybrid recommendation system. We will have four recommenders (K-Means Clustering, Decision Tree, Cosine Similarity, Locality-Sensitivity Hashing) to be combined using an ensemble majority voting.

1.2 Pre-processing & PCA

We normalized all attribute values which included subtracting the mean and dividing by the standard deviation for easier and more reliable graphical comparison. Since all values for the provided attributes were available and valid, no transformations were necessary to account for missing data. We also removed columns consisting of non-quantitative data as all of our computations require numeric data. We also removed these columns as to give less weight to songs from the same decade or from the song artists and similar titles which generally result in a more diverse song recommendation model. We also performed principal component analysis to reduce dimensionality of our dataset that contained 14 quantitative attributes, thus making our data easier to visualize and faster to run. PCA showed that the first 8 principal components account for more than 80% of the variance in the data as shown in figure 1.2. Therefore, we used PCA decomposed data as inputs for the methods.

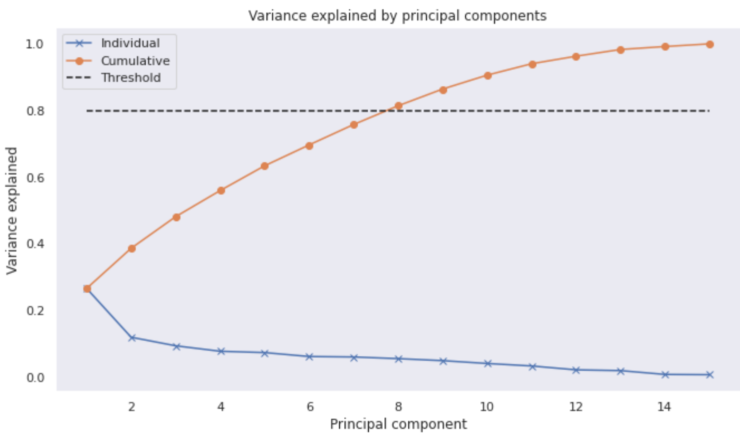


Figure 1.2: The figure depicts the proportion of variance for increasing number of components where the dotted line shows the cut off value of 80%.

1.3 K-Means Clustering

We chose to predict song recommendations with the K-Means clustering to increase our diversity in song recommendations. We also used the K-Means clustering as a way to label our data for further classification methods / Decision Tree mentioned below as K-Means clustering is an unsupervised machine learning algorithm that is both very popular and simple to implement.

In the K-Means Clustering method, we fit the data to be split into 10 clusters which was determined by testing different cluster sizes or values of k and plotting them against the Davies-Bouldin (DB) Index. The DB Index is the ratio of cluster scatter to the cluster's separation, therefore, a lower DB Index is more desirable. Figure 1.3 depicts a cluster of 10 results in approximately the lowest DB Index. Cluster results can be seen in figure 1.4. In order to the obtain song recommendations, we take the last liked song or song input, find the closest cluster centroid with euclidean distance, then we recommend songs contained in that cluster.

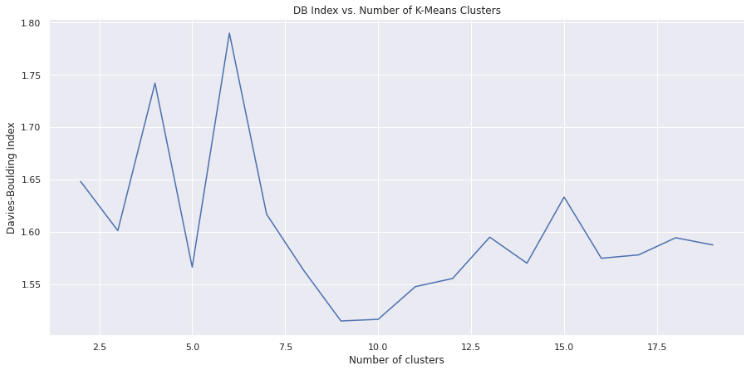


Figure 1.3: The figure depicts the number of clusters vs. DB index where the optimal clusters is approximately 10.

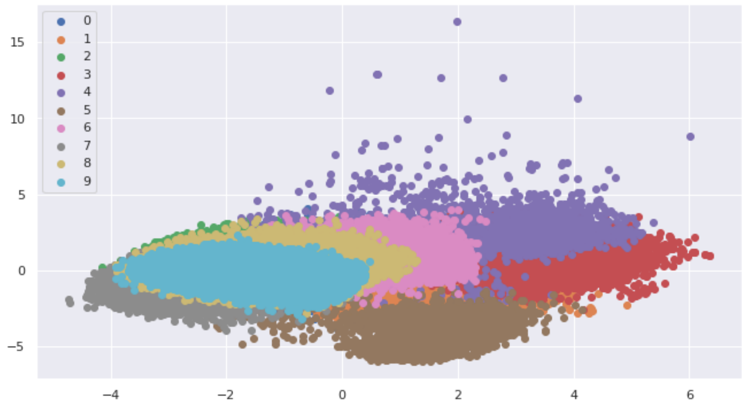


Figure 1.4: The figure depicts the results of our K-Means Clustering result for 10 clusters.

1.4 Decision Tree

The decision tree is a classifier that consists of several nodes which form a directed tree with a single root node that has no incoming edges. The rest of the nodes in the decision tree all have exactly one incoming edge, and if that edge is outgoing, it is called an interior node, while the rest are leaf nodes, which are also the decision nodes. Each interior node splits the node into two or more sub-nodes based on the input values of the given node. The leaf node at the end then holds the most appropriate target value for the criteria of the previous interior nodes [Rokach and Maimon, 2005]. The decision tree should therefore be able to classify which cluster a song most likely would belong to based on the attributes of the song. The decision tree was chosen over other machine learning classification methods as it gives the desired information of the cluster number in an accessible way in form of the leaf nodes. We decided to implement the decision tree from scratch.

1.4.1 Our Decision Tree Algorithm

The decision tree starts by checking whether the data is pure or not, as this is a requirement for it to be classified. The data is pure when there is one class in the interior node. If it is not pure then the data needs to be split into further interior nodes, but there is a limit to the number of interior nodes that can be created, which is the max depth. If the max depth has been reached, more interior nodes will not be created. The potential split happens by iterating over all attributes and finding all the unique values for that attribute. The split for each attribute would then be the midpoint of these unique values. After finding all potential splits, the optimal split has been found by calculating the entropy for each split. Entropy is being used to determine the purity of the different classes. If the entropy is high, then the purity is low. The lower the entropy the better chance the model has of predicting the classes as the purity is high. The formula for the entropy is:

$$E[X] = -\sum xp(x) * log_2 * p(x) \tag{1.1}$$

where x is a class and p(x) is the probability of class in the interior node. The potential splits are divided into a left or right value. The split with the lowest entropy will be selected. This continues until the left or right interior node is empty which then created a leaf node from the non-empty interior node. The nodes consist of a question based on the name of the attributes and a value found from the best spilt. If the answer is yes, the next node will be on the left side and if the answer is no it will be on the right side.

We have used the decision tree to classify the cluster number of a song, and with that cluster number, we are recommending 5 (arbitrarily chosen but code is written with song number as an input) different songs from the cluster. The decision tree uses the PCA features and the cluster number and the tree is trained on 50.000 songs, due to time constraints. For this project, we have selected a random song that will be used in the decision tree song recommender. The decision tree can be seen in figure 1.5.

```
{'PCA0 <= -0.020539867249146096': [{'PCA2 <= 1.1661138449661386': [{'PCA4 <= 0.8290177729975929':
[{'PCA3 <= 1.1243309081477864': [1.0,
        9.0]},
      {'PCA7 <= 0.5617595365787885': [6.0, 1.0]}]},
    {'PCA1 <= 0.8251347731124794': [{'PCA0 <= -1.7634859814456654': [4.0,
        4.0]},
      {'PCA3 <= 1.3597137680740041': [6.0, 9.0]}]}]},
  {'PCA1 <= -0.2262897119577551': [{'PCA2 <= 0.6887394691719135': [{'PCA0 <= 2.583408333102769':
[3.0,
        3.0]},
      {'PCA1 <= -2.6121014460523106': [5.0, 7.0]}]},
    {'PCA0 <= 2.031983318022677': [{'PCA3 <= 0.944400852541201': [8.0, 9.0]},
      {'PCA7 <= 1.262793588327254': [7.0, 0.0]}]}]}]}
```

Figure 1.5: The figure depicts the final Decision Tree which branches between principal components called "PCA"

1.5 Cosine Similarity

We chose to predict song recommendations with cosine similarity since we are evaluating data with high dimensionality as well as data where the actual magnitude of the vectors is not important. Evaluating using cosine similarity also provides more depth than evaluating based on euclidean distance [Grootendorst, 2021].

In the Cosine Similarity method, we constructed a matrix of cosine similarities. Cosine similarity is defined as the cosine of the angle between the two input song vectors (the song selected and another song randomly sampled in our dataset) as shown by the equation below [Karabiber, 2022]. Our method then returns a given number of songs with the highest cosine similarity to the input song.

$$\cos(\mathbf{a},\mathbf{b}) = \frac{\mathbf{a}\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} = \frac{\sum_{i=1}^{n} \mathbf{a}_i \mathbf{b}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{a}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{b}_i)^2}} \tag{1.2}$$

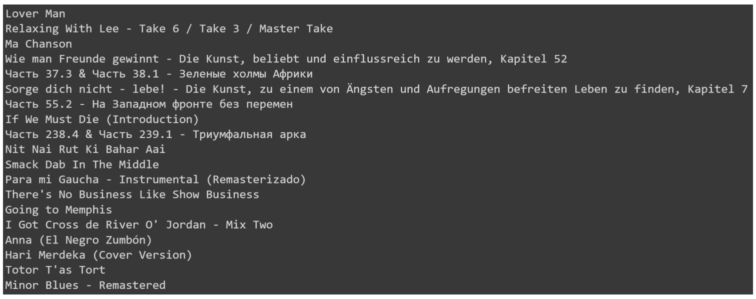
1.6 Locality Sensitivity Hashing

We chose to incorporate local sensitivity hashing to analyze similar song lyrics for a few reasons. Most notably, LSH is effective for high dimensionality problems / large datasets with high accuracy. LSH also has overall improved efficiency as the method of creating signatures only requires n iterations through the dataset as opposted to n^2 iterations [Hari, 2018].

In our locality-sensitivity hashing method, we take our signatures from each set of lyrics and break them into b blocks of length r (where k = br is the length of the signatures). For each block, a hash function is applied. If two songs get hashed to the same value for at least one block, they will be considered a candidate pair. Then, we evaluate the Jaccard similarity of each candidate pair, and if the candidate pairs are above a threshold s defined by $(1/b)^{1/r} \approx s$ [Leskovec, 2022] . The signatures algorithm converts a list of k-shingles from each set of lyrics (any substring of length k found within the document) to a sequence of minhash values. We used the following values for k, b, r, and s as follows: 5, 20, 5, and 0.6 since k, b, and r values seemed appropriate for lyric analysis while s is computed as mentioned above. The LSH method, therefore, takes the song input and recommends songs similar in lyrical content above the Jaccard similarity threshold.

1.7 Ensemble of Methods

We used a majority vote prediction to obtain the final list of song recommendations since this is a simple way of combining our methods into a classification voting ensemble. Specifically, we used a hard voting method that predicts the class or song with the largest sum of votes from the models. We chose to create an ensemble of methods because generally, predictions will be better than any individual method [Harrison, 2022]. With creating an ensemble, we accumulate the following benefits of multiple models which may include faster efficiency in terms of using LSH, high dimensionality problem with cosine similarity, and more diverse song results with clustering and the decision tree etc. Example final output for the song 'Tears' with 5 songs recommended per method give the following output:



```
Lover Man
Relaxing With Lee - Take 6 / Take 3 / Master Take
Ma Chanson
Wie man Freunde gewinnt - Die Kunst, beliebt und einflussreich zu werden, Kapitel 52
Часть 37.3 & Часть 38.1 - Зеленые холмы Африки
Sorge dich nicht - lebe! - Die Kunst, zu einem von Ängsten und Aufregungen befreiten Leben zu finden, Kapitel 7
Часть 55.2 - На Западном фронте без перемен
If We Must Die (Introduction)
Часть 238.4 & Часть 239.1 - Триумфальная арка
Nit Nai Rut Ki Bahar Aai
Smack Dab In The Middle
Para mi Gaucha - Instrumental (Remasterizado)
There's No Business Like Show Business
Going to Memphis
I Got Cross de River O' Jordan - Mix Two
Anna (El Negro Zumbón)
Hari Merdeka (Cover Version)
Totor T'as Tort
Minor Blues - Remastered
```

Figure 1.6: The figure depicts list of 20 songs where input song is 'Tears'

1.8 Technical Issues

We ran into various technical challenges while working with our dataset. The original dataset found did not have lyrics data. However, we wanted to analyze lyrics as part of a similarity measure for our overall song recommendation system and found a smaller dataset which had many songs in common with the larger dataset which did have data for lyrics. Thus, we randomly chose a song input that belonged to both datasets and produced song recommendations for each of the four methods in order to have song recommendations based on the same input. Another issue was deciding a method to create an ensemble. We implemented the hard voting system, however, since there may not be a lot of overlap of song recommendation between methods especially since the number of songs recommendations is set to a small number as an example, the final predictions are essentially a combination of song recommendations from all four methods.

2 Appendix

2.1 A: Contribution Table

	Elysia	Oscar	Chris	Ifi
Data Visualization & Meeting Attendence	25%	25%	25%	25%
Data Pre-processing	50%	50%		
K-Means Clustering	80%	20%		
Decision Tree		100%		
Cosine Similarity	100%			
Locality-Sensitivity Hashing	90%	10%		
Ensemble of Methods	70%	30%		
Report Writing	70%	30%		
Overall	61%	33%	3%	3%

Figure 2.1: Contribution Table

2.2 B: Jupyter Notebook

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import plotly.express as px
import matplotlib.pyplot as plt
import xlrd
from scipy.linalg import svd
from matplotlib.pyplot import figure, plot, title, xlabel, ylabel, show, legend
from scipy.linalg import svd
from sklearn.impute import SimpleImputer
from sklearn import model_selection
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import davies_bouldin_score
from sklearn.metrics.pairwise import cosine_similarity
import sys
import os
#!pip install mmh3
import mmh3
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import VotingClassifier
from random import randrange
from collections import Counter
```

In [2]:

```
#from google.colab import drive
#drive.mount('/content/drive')
```

Data investigation

In [3]:

```
#data = pd.read_csv('/content/drive/MyDrive/Computational tools for DS/data.csv')
data = pd.read_csv('data/data.csv')
```

In [4]:

```
data.info()
songNames = data.loc[:,'name'];
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170653 entries, 0 to 170652
Data columns (total 19 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   valence           170653 non-null  float64
 1   year              170653 non-null  int64  
 2   acousticness      170653 non-null  float64
 3   artists           170653 non-null  object 
 4   danceability      170653 non-null  float64
 5   duration_ms       170653 non-null  int64  
 6   energy            170653 non-null  float64
 7   explicit          170653 non-null  int64  
 8   id                170653 non-null  object 
 9   instrumentalness  170653 non-null  float64
 10  key               170653 non-null  int64  
 11  liveness          170653 non-null  float64
 12  loudness          170653 non-null  float64
 13  mode              170653 non-null  int64  
 14  name              170653 non-null  object 
 15  popularity        170653 non-null  int64  
 16  release_date      170653 non-null  object 
 17  speechiness       170653 non-null  float64
 18  tempo             170653 non-null  float64
```

```
dtypes: float64(9), int64(6), object(4)
memory usage: 24.7+ MB
```

In [5]:
```
data.T
```

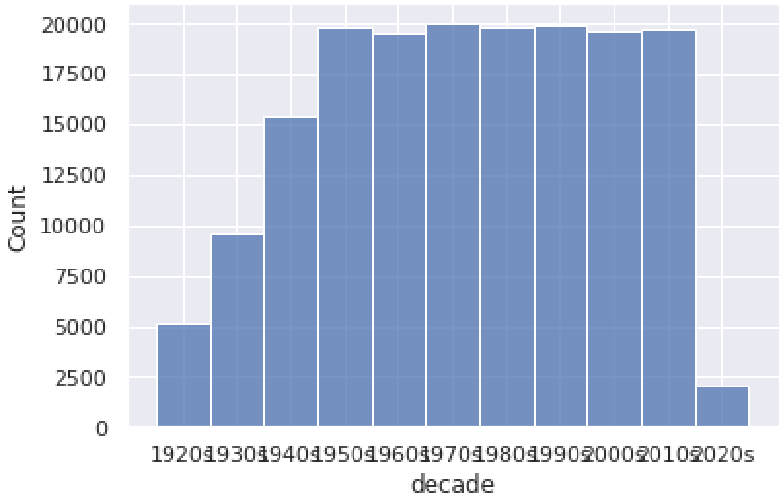
Out[5]:

	0	1	2	
valence	0.0594	0.963	0.0394	
year	1921	1921	1921	
acousticness	0.982	0.732	0.961	
artists	['Sergei Rachmaninoff', 'James Levine', 'Berli...	['Dennis Day']	['KHP Kridhamardawa Karaton Ngayogyakarta Hadi...	['Frank
danceability	0.279	0.819	0.328	
duration_ms	831667	180533	500062	
energy	0.211	0.341	0.166	
explicit	0	0	0	
id	4BJqT0PrAfrxzMOxytFOIz	7xPhfUan2yNtyFG0cUWkt8	1o6I8BglA6ylDMrIELygv1	3ftBPsC5vPBKxYSe
instrumentalness	0.878	0.0	0.913	0
key	10	7	3	
liveness	0.665	0.16	0.101	
loudness	-20.096	-12.441	-14.85	
mode	1	1	1	
name	Piano Concerto No. 3 in D Minor, Op. 30: III. ...	Clancy Lowered the Boom	Gati Bali	Da
popularity	4	5	5	
release_date	1921	1921	1921	
speechiness	0.0366	0.415	0.0339	
tempo	80.954	60.936	110.339	

19 rows × 170653 columns

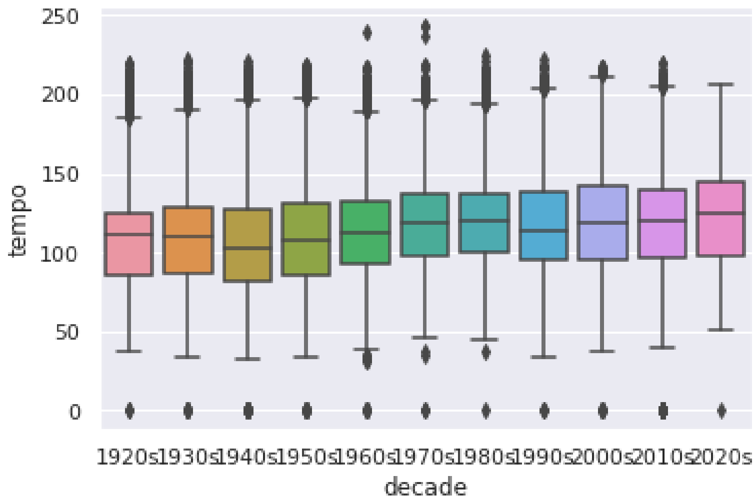
In [6]:
```python
# Visualizing the amount of songs by the decades in the dataset
def get_decade(year):
    start = int(year/10) * 10
    decade = '{}s'.format(start)
    return decade

data['decade'] = data['year'].apply(get_decade)

sns.set(rc={'figure.figsize':(11 ,6)})
sns.histplot(data['decade'])
```

Out[6]:
```
<AxesSubplot:xlabel='decade', ylabel='Count'>
```



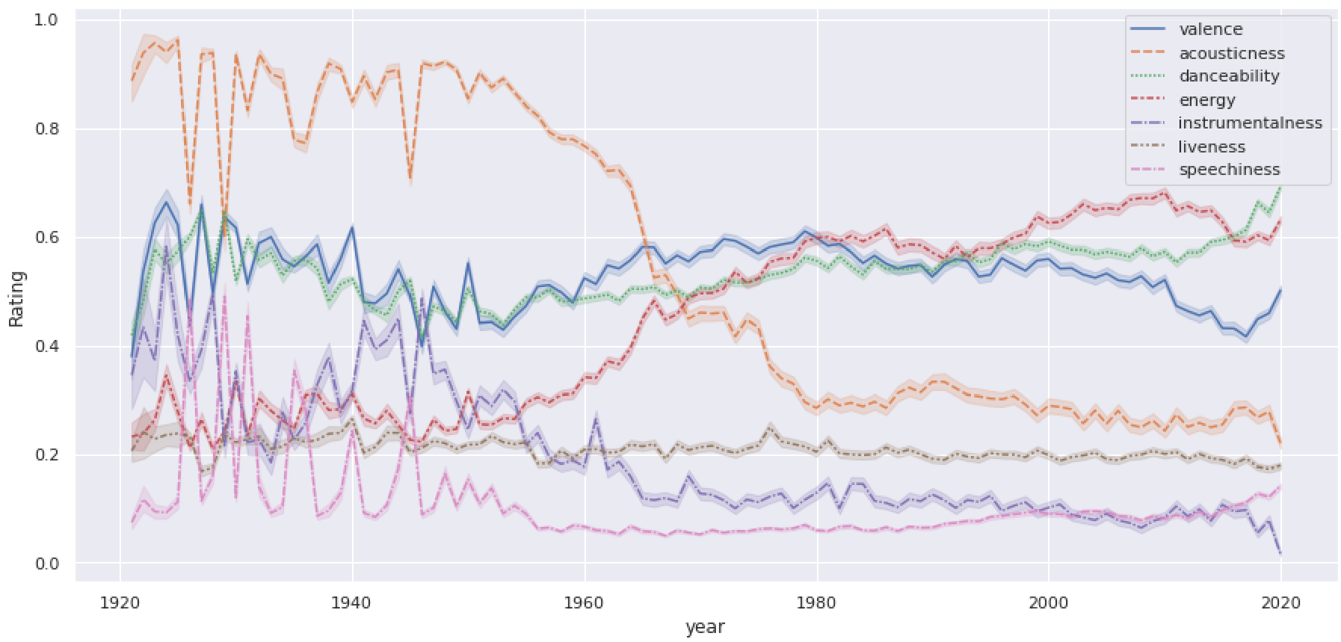
In [7]:
```
sns.boxplot(x = data['decade'], y= data['tempo'])
#sns.boxplot(x = data['decade'], y= data['energy'])
#sns.boxplot(x = data['decade'], y= data['acousticness'])
```

Out[7]: <AxesSubplot:xlabel='decade', ylabel='tempo'>



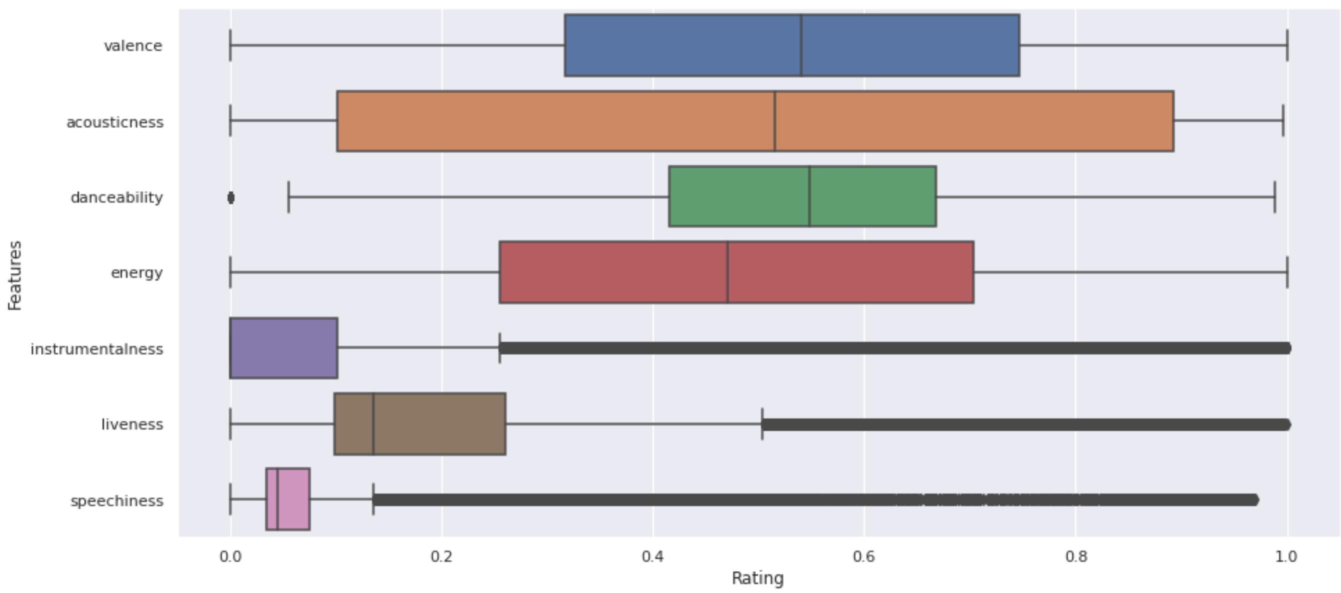
In [8]:
```
# Visualizing different songs features to see how the evolve
features = ['year','valence','acousticness','danceability','energy','instrumentalness','
data_melted = data[features].melt("year",var_name="Features",value_name="Rating")
sns.set(rc={'figure.figsize':(15 ,7)})
sns.lineplot(data=data_melted, x="year", y="Rating", hue="Features", style = 'Features')
plt.legend(loc='upper right')
```

Out[8]: <matplotlib.legend.Legend at 0x7fd0d85b4590>



In [9]:
```python
# Boxplot of the rating for the features
sns.boxplot(data=data_melted, x="Rating", y ='Features')
```

Out[9]: <AxesSubplot:xlabel='Rating', ylabel='Features'>



In [10]:
```python
corr = data.corr()
corr.style.background_gradient(cmap='RdYlGn')
```

Out[10]:

	valence	year	acousticness	danceability	duration_ms	energy	explicit	instrume
valence	1.000000	-0.028245	-0.184101	0.558946	-0.191813	0.353876	-0.018613	-(
year	-0.028245	1.000000	-0.614250	0.188515	0.079713	0.530272	0.220881	-(
acousticness	-0.184101	-0.614250	1.000000	-0.266852	-0.076373	-0.749393	-0.246007	(
danceability	0.558946	0.188515	-0.266852	1.000000	-0.139937	0.221967	0.241757	-(
duration_ms	-0.191813	0.079713	-0.076373	-0.139937	1.000000	0.042119	-0.048880	(
energy	0.353876	0.530272	-0.749393	0.221967	0.042119	1.000000	0.132723	-(
explicit	-0.018613	0.220881	-0.246007	0.241757	-0.048880	0.132723	1.000000	-(
instrumentalness	-0.198501	-0.272371	0.329819	-0.278063	0.084770	-0.281101	-0.140987	1
key	0.028473	0.007540	-0.020550	0.024439	-0.004266	0.027705	0.005432	-(

liveness	0.003832	-0.057318	-0.024482	-0.100193	0.047168	0.126192	0.039640	-0
loudness	0.313512	0.487697	-0.561696	0.285057	-0.003037	0.782362	0.140300	-0
mode	0.015641	-0.032385	0.047168	-0.045956	-0.046085	-0.039260	-0.078872	-0
popularity	0.014200	0.862442	-0.573162	0.199606	0.059597	0.485005	0.191543	-0
speechiness	0.046381	-0.167816	-0.043980	0.235491	-0.084604	-0.070555	0.414070	-0
tempo	0.171689	0.141048	-0.207120	0.001801	-0.025472	0.250865	0.011969	-0

In [11]:
```
# Finding unique combinations of artists on a song
artists = []
for i in data['artists'].values:
    if i in artists:
        continue
    else:
        artists.append(i)

# Printing the unique combinations
len(artists)
```

Out[11]:
```
34088
```

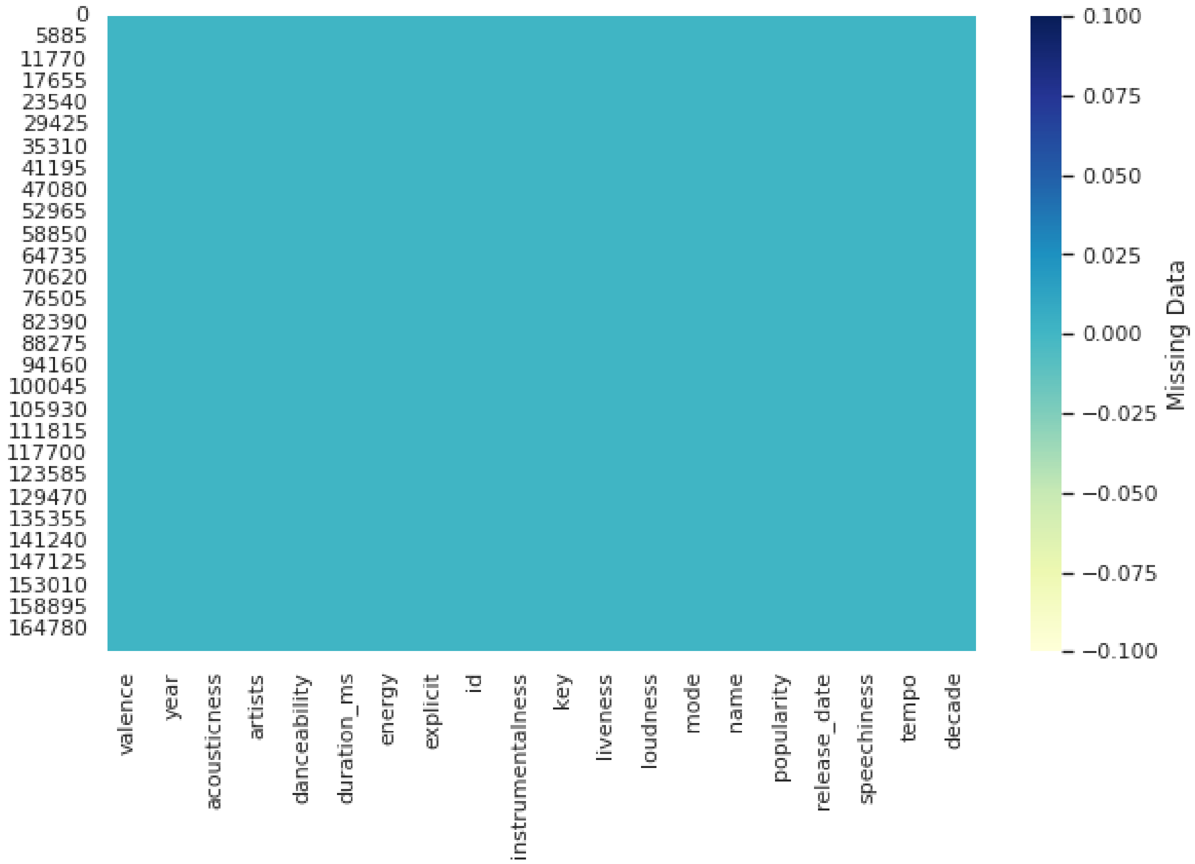
In [12]:
```
# Finding the average duration of the songs
print("The average duration of the songs is", round(data['duration_ms'].mean()/1000), "s
print("The average tempo of the songs is", round(data['tempo'].mean()))
```

```
The average duration of the songs is 231 seconds
The average tempo of the songs is 117
```

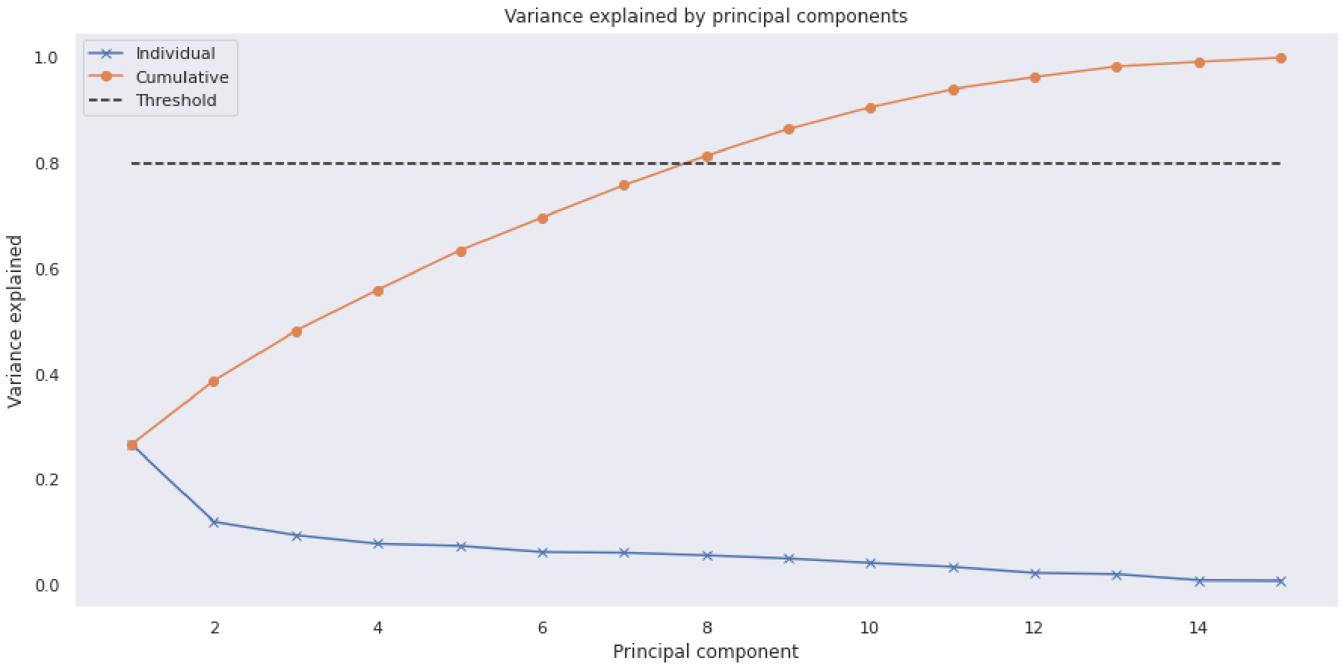
Data Preparation/Pre-processing:

In [13]:
```
# Remove qualitative rows for easier computation, standardize and normalize data
data2 = data.drop(columns = ['artists','id','name', 'release_date','decade'])
data2 = SimpleImputer(strategy='mean').fit_transform(data2)
data2 = StandardScaler().fit_transform(data2)
N, M = data2.shape
```

In [14]:
```
# Visualize any missing data
plt.figure(figsize=(10,6))
sns.heatmap(data.isna(), cmap="YlGnBu", cbar_kws={'label': 'Missing Data'})
plt.savefig("visualizing_missing_data_with_heatmap_Seaborn_Python.png", dpi=100)
```

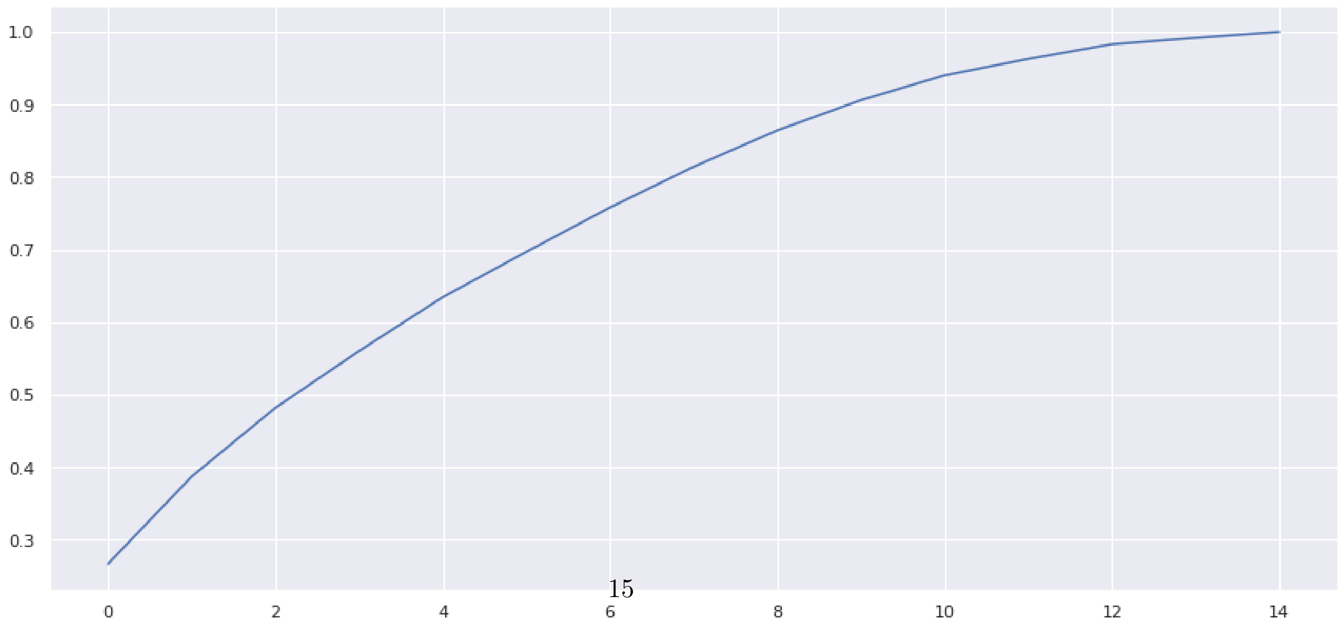


In [15]:
```
# PCA Analysis
Y = data2 - np.ones((N,1))*data2.mean(axis=0)
U,S,V = svd(Y,full_matrices=False)
rho = (S*S) / (S*S).sum()
threshold = 0.8

plt.figure()
plt.plot(range(1,len(rho)+1),rho,'x-')
plt.plot(range(1,len(rho)+1),np.cumsum(rho),'o-')
plt.plot([1,len(rho)],[threshold, threshold],'k--')
plt.title('Variance explained by principal components')
plt.xlabel('Principal component')
plt.ylabel('Variance explained')
plt.legend(['Individual','Cumulative','Threshold'])
plt.grid()
plt.show()

Y = data2 - np.ones((N,1))*data2.mean(0)
U,S,Vh = svd(Y,full_matrices=False)
V = Vh.T
Z = Y @ V
```



In [16]:
```
# PCA Analysis using library in python
from sklearn.decomposition import PCA
pca = PCA()
pca.fit(data2)
pca.explained_variance_ratio_

# Plt cumulative graph of total variance
expl = pca.explained_variance_ratio_
cdf = [sum(expl[:i+1]) for i in range(len(expl))]
plt.plot(range(len(expl)), cdf);

# Finding the number of principal components needed
i = 0
threshold = 0.80
while cdf[i] < threshold:
    i += 1
print(i+1, cdf[i])
```

```
8 0.8142233124132674
```



In [22]:
```
# Decomposition of the data
from sklearn import decomposition
```

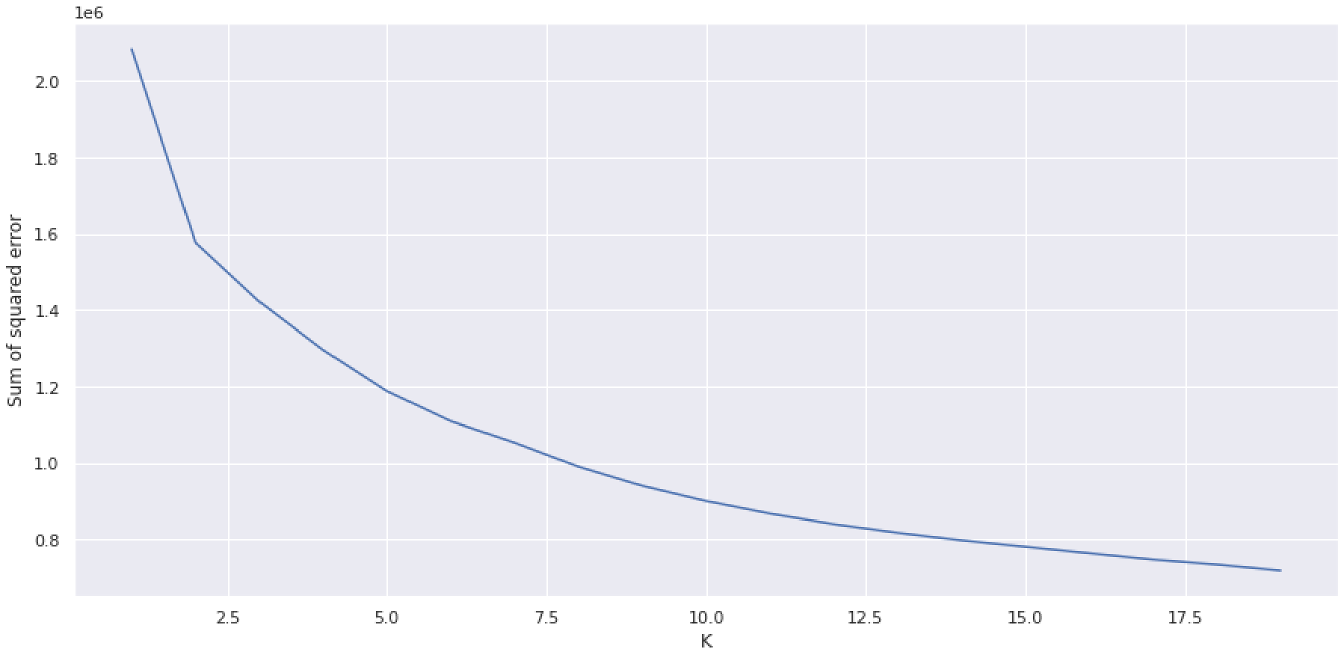
```
pca_new=decomposition.PCA(n_components=8)
pca_new.fit(data2)
X_reduced=pca_new.transform(data2)
X_reduced.shape
```

Out[22]:
```
(170653, 8)
```

K-Means Clustering

In [23]:
```
# Determine number of clusters for K-Means by plotting clusters vs. SSE
sse = []
k_rng = range(1,20)
for k in k_rng:
    km = KMeans(n_clusters=k)
    km.fit(X_reduced)
    sse.append(km.inertia_)

plt.xlabel('K')
plt.ylabel('Sum of squared error')
plt.plot(k_rng,sse)
```

Out[23]:
```
[<matplotlib.lines.Line2D at 0x7fd0d8604510>]
```

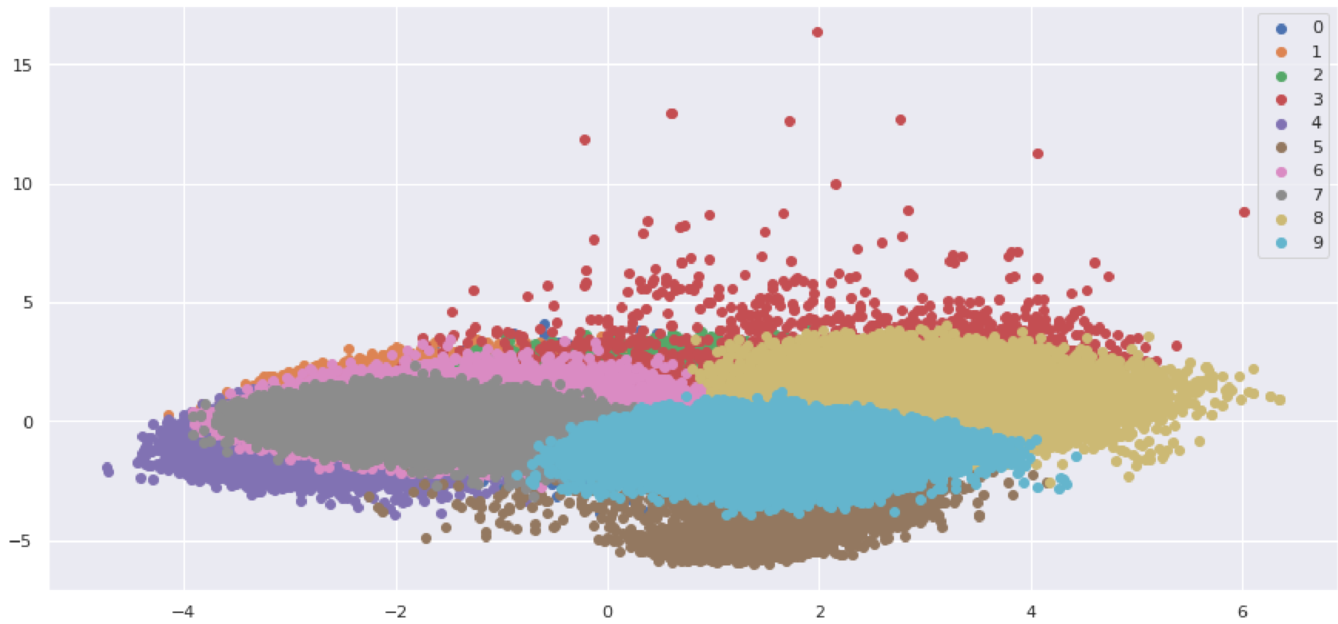


In [24]:
```
# Performs K-Means Clustering on dataset, k = 10
clt = KMeans(n_clusters=10)
model = clt.fit(X_reduced)
centroids = clt.cluster_centers_
clusters = pd.DataFrame(model.fit_predict(X_reduced))
label = clt.fit_predict(X_reduced)
clusters['cluster'] = model.labels_
```

In [25]:
```
# Plots DB Index Graph to determine number of clusters
results = {}
for i in range(2,20):
    kmeans = KMeans(n_clusters=i, random_state=30)
    labels = kmeans.fit_predict(X_reduced)
    db_index = davies_bouldin_score(X_reduced, labels)
    results.update({i: db_index})

plt.plot(list(results.keys()), list(results.values()))
plt.xlabel("Number of clusters")
```

```
plt.ylabel("Davies-Boulding Index")
plt.title("DB Index vs. Number of K-Means Clusters")
plt.show()
```

In [26]:
```
# Plotting the K-Means Cluster results:
u_labels = np.unique(label)

for i in u_labels:
    plt.scatter(X_reduced[label == i , 0] , X_reduced[label == i , 1] , label = i)
plt.legend()
plt.show()
```



In [27]:
```
# Classification by finding euclidean distance to the nearest centroid
def closest_centroid(X):
    num_clusters = len(centroids)
    L = len(centroids[0])
    diff=np.zeros(shape = (num_clusters,L),dtype = 'float')
    id = 0;    #id is index of the closest centroid
    diff[0, :] = X - centroids[0,:]
    min_distance = sum(np.multiply(diff[0,:],diff[0,:]))
    for j in range(1,num_clusters):
        diff[j, :] = X - centroids[j,:]
```

```
        distance = sum(np.multiply(diff[j,:],diff[j,:]))
        if (distance<min_distance):
            min_distance = distance
            id = j
    return id
```

In [98]:
```
# Predicts song recommendations using closest centroid method given a song input
def cluster_pred(song_id,numsongs):
    song_id_arr = X_reduced.iloc[3,:].to_list()
    clust_num = closest_centroid(song_id_arr)
    clust = clusters[clusters.cluster == clust_num ]
    song_list = list()
    for song in range(numsongs):
        id_list = clust.sample().index;
        song_list.append(data.iloc[id_list,14].tolist())
        song_list_flattened  = [val for sublist in song_list for val in sublist]
    return song_list_flattened
```

In [97]:
```
# Test - recommends 10 songs based on K-Means clustering method
print(cluster_pred(3,10))
```

```
Kapitel 219 - Der Page und die Herzogin
Часть 91.3 - По ком звонит колокол
Часть 85.2 - По ком звонит колокол
Часть 104.2 - По ком звонит колокол
Часть 86.4 & Часть 87.1 - Зеленые холмы Африки
Toss of the Coin
Kapitel 389 - Der Page und die Herzogin
Das ist bei uns nicht möglich, Kapitel 101
Capítulo 24.4 - la Sombra Fuera del Tiempo
Часть 2.12 - Обратный путь
['Kapitel 219 - Der Page und die Herzogin', 'Часть 91.3 - По ком звонит колокол', 'Часть
85.2 - По ком звонит колокол', 'Часть 104.2 - По ком звонит колокол', 'Часть 86.4 & Част
ь 87.1 - Зеленые холмы Африки', 'Toss of the Coin', 'Kapitel 389 - Der Page und die Herz
ogin', 'Das ist bei uns nicht möglich, Kapitel 101', 'Capítulo 24.4 - la Sombra Fuera de
l Tiempo', 'Часть 2.12 - Обратный путь']
```

Decision Tree From Scratch

In [31]:
```
# A function for checking for the amount of unique types, purity check. The data is pure
def purity_check(data):
    types = data[:,-1]
    unique_types = np.unique(types)
    if len(unique_types) == 1:
        return True
    else:
        return False
```

In [32]:
```
# Making a classifier that will also make the leaf nodes in the decision tree
def classifier(data):
    types = data[:,-1]
    unique_types,unique_types_count = np.unique(types,return_counts=True)
    index = unique_types_count.argmax()
    classification = unique_types[index]
    return classification
```

In [33]:
```
# Getting the potentiel splits by iterating over the different columns(attributes) in th
def get_potential_splits(data):
    potential_splits={}
    _,column_no=data.shape
    for column_index in range(column_no-1):
        values=data[:,column_index] #Finding all values in the column
        unique_values=np.unique(values) #Unique values
```

```
        mid_points=(unique_values[1:]+unique_values[:-1])/2 #calculating the midpoints
        potential_splits[column_index]=mid_points #potential splits
    return potential_splits
```

In [34]:
```
# A function to split the data. It takes the best split column and value and split them i
def data_split(data,split_col,split_val):
    split_col_val = data[:,split_col] #finding the values from the columns after the nod
    left = data[split_col_val <= split_val] #left split
    right = data[split_col_val > split_val] #right split
    return left,right
```

In [35]:
```
# Calculating the entropy which is used to determine the purity of the splits.
def cal_entropy(data):
    types = data[:,-1]
    _,count = np.unique(types,return_counts=True)
    prob = count/count.sum() #calculating the probability from  by dividing the count of
    entropy = -sum(prob*np.log2(prob)) #Using the formula for calculating entropy
    return entropy
```

In [36]:
```
# Calculating the overall entropy by adding the entropy from the left and right node
def overall_entropy(left,right):
    data_points = len(left) + len(right)
    p_left = len(left) / data_points
    p_right = len(right) / data_points
    overall_entropy = (p_left * cal_entropy(left)+p_right * cal_entropy(right))
    return overall_entropy
```

In [37]:
```
# Finding the best splits
def best_split(data,pot_splits):
    fixed_entropy = 999 # Seeting a high fixed entropy that will be used
    for col_index in pot_splits: #Iterating over column index from the potential splits
        for val in pot_splits[col_index]: #Iterating over the values from the column ind
            left,right = data_split(data,split_col=col_index,split_val=val) # splitting
            current_entropy = overall_entropy(left,right) #setting the overall entropy o
            if current_entropy <= fixed_entropy:
                fixed_entropy= current_entropy #If the current entropy is lower than the
                best_s_col = col_index # saving the column index as the best split colum
                best_s_val = val # saving the corresponding value as the best split valu
    return best_s_col, best_s_val
```

In [38]:
```
# Making the decision tree
def decision_tree(df,counter, max_depth):
    if counter == 0:
        global col_header #setting as a global value so it keeps it name throughout the
        col_header = df.columns #retriving the name of the columns
        data = df.values #saving the values from the dataframe with pandas
    else:
        data = df
    #base case
    if purity_check(data) or (counter == max_depth): #Base for the function to stop recu
        leaf = classifier(data)
        return leaf #returns the a leaf node also known as the leaf node
    #Recursion
    else:
        counter += 1
        pot_splits = get_potential_splits(data) #Finding the potential splits for the da
        split_col,split_val = best_split(data,pot_splits) #Getting the best split based
        left,right = data_split(data,split_col,split_val) #Splitting the data into inter
```



```
        # Checking for empty data and if empty returns a leaf node
        if len(left) == 0 or len(right) == 0:
            leaf = classifier(data)
            return leaf
        #nodes
```

```
        name = col_header[split_col] # Finding the name based on the colums name and col
        question = "{} <= {}".format(name,split_val) #Forming the question for each inte
        node = {question:[]} #Making the interior node

        #Answer the question
        yes = decision_tree(left,counter,max_depth) #Finding the yes answers
        no = decision_tree(right,counter,max_depth) #Finding the no answers

        node[question].append(yes) #Appending the yes answers
        node[question].append(no) #Appending the no answers

        return node
```

In [73]:

```
# Making a classify example that will be used the prediction
def classify_ex(example,tree):

    #if the tree is just a root node
    if not isinstance(tree,dict):
        return tree

    question = list(tree.keys())[0] #Finding just the questions
    name,operator,val = question.split() #Splitting the question into the name, operator

    #Finding the answer that will be used to make the predictions
    if example[name] <= float(val):
        answer = tree[question][0]
    else:
        answer = tree[question][1]

    if not isinstance(answer,dict):
        return answer

    else:
        residual_tree = answer
        return classify_ex(example,residual_tree)
```

In [40]:

```
def recommend_dict(songs, tree, data):
    tmp = []
    artist = []
    song_recom = []
    df = pd.DataFrame()
    for i in songs:
        if data['name'].str.contains(i).any():
            index = data['name'][data['name'] == i].index[0]
            df = df.append(data[['danceability','duration_ms','energy','popularity','clu
    for i in range(0,len(df)):
        g = []
        hell = df[i:i+1]
        m = classify_ex(hell,tree)
        sim_songs = data['name'][data['cluster'] == m].sample(n= 10)
    for i in sim_songs:
        song_recom.append(i)
    for i in song_recom:
        if data['name'].str.contains(i).any():
            index = data['name'][data['name'] == i].index[0]
            tmp.append(data['artists'][index:index+1])
    for i in tmp:
        for x in i:
            artist.append(x)
    res = dict(zip(artist, song_recom))
    return res
```


In [41]:

```
def recommend_songs(songs, tree, data):
    song_recom = []
```

```
    df = pd.DataFrame()
    for i in songs:
        if data['name'].str.contains(i).any():
            index = data['name'][data['name'] == i].index[0]
            df = df.append(data[['danceability','duration_ms','energy','popularity','clu
    for i in range(0,len(df)):
        g = []
        hell = df[i:i+1]
        m = classify_ex(hell,tree)
        sim_songs = data['name'][data['cluster'] == m].sample(n= 10)
    for i in sim_songs:
        song_recom.append(i)
    return song_recom
```

In [42]:

```
# recommend on PCA data
def recom(index, tree, data, original_data):
    song_list = list();
    sim_of_indiv_song = data.iloc[index];
    m = classify_ex(sim_of_indiv_song,tree)
    tmp = original_data['name'][original_data['cluster'] == m].sample(n= 10)

    #sim_of_indiv_song = sim_of_indiv_song.sort_values(ascending=False);
    #sim_of_indiv_song = sim_of_indiv_song[0:num_songs];
    #song_id = sim_of_indiv_song.index;
    for i in tmp:
        song_list.append(i)
    return song_list;
```

In [76]:

```
# Adding the cluster number to the data to use for the decision tree
X_reduced_tree = pd.DataFrame(X_reduced)
X_reduced_tree = X_reduced.rename(columns={0: "PCA0", 1: "PCA1", 2: "PCA2", 3: "PCA3", 4
X_reduced_tree['cluster'] = clusters[0]
data['cluster'] = clusters[0]
```

In [44]:

```
# Getting training data for the decision tree
train = X_reduced.sample(n= 50000)
```

In [45]:

```
# Setting the max depth for the decision tree
counter = 0
max_depth = 4
```

In [46]:

```
# Running the decision tree
tree = decision_tree(train,counter,max_depth)
```

In [91]:

```
tree
```

Out[91]:

```
{'PCA0 <= -0.020539867249146096': [{'PCA2 <= 1.1661138449661386': [{'PCA4 <= 0.829017772
9975929': [{'PCA3 <= 1.1243309081477864': [1.0,
        9.0]},
      {'PCA7 <= 0.5617595365787885': [6.0, 1.0]}]},
    {'PCA1 <= 0.8251347731124794': [{'PCA0 <= -1.7634859814456654': [4.0,
        4.0]},
      {'PCA3 <= 1.3597137680740041': [6.0, 9.0]}]}]},
  {'PCA1 <= -0.2262897119577551': [{'PCA2 <= 0.6887394691719135': [{'PCA0 <= 2.583408333
102769': [3.0,
        3.0]},
      {'PCA1 <= -2.6121014460523106': [5.0, 7.0]}]},
    {'PCA0 <= 2.031983318022677': [{'PCA3 <= 0.944400852541201': [8.0, 9.0]},
      {'PCA7 <= 1.262793588327254': [7.0, 0.0]}]}]}]}
```

In [47]:

```
# Recommend songs based on the song from the list
#recommend_songs(song,tree,data)
```

In [48]:
```
# Recommend songs + artists based on the song from the list
#recommend_dict(song,tree,data)
```

Cosine Similarity

In [95]:
```
# Creates a matrix that computes cosine similarity between songs
X_reduced = pd.DataFrame(X_reduced)
df_percent = X_reduced.sample(frac=0.1)

def cosine_sim_pred(index,num_songs):
    similarities = cosine_similarity(df_percent)
    cosine_sim = pd.DataFrame(similarities)
    song_list = list()

    sim_of_indiv_song = cosine_sim.iloc[index]
    sim_of_indiv_song = sim_of_indiv_song.sort_values(ascending=False)
    sim_of_indiv_song = sim_of_indiv_song[0:num_songs]
    song_id = sim_of_indiv_song.index

    for song in song_id:
        data.iloc[song,14]
        song_list.append(data.iloc[song,14])

    return song_list;
```

In [50]:
```
# Test - recommends 10 songs using Cosine Similarity Method
print(cosine_sim_pred(3,10))
```

```
Danny Boy
Ants Marching
No Milk Today
Make It Last Forever (with Jacci McGhee)
Munchkinland
Lady
You And I
I Am A Pilgrim
Rubin And Cherise
You Might Think
['Danny Boy', 'Ants Marching', 'No Milk Today', 'Make It Last Forever (with Jacci McGhe
e)', 'Munchkinland', 'Lady', 'You And I', 'I Am A Pilgrim', 'Rubin And Cherise', 'You Mi
ght Think']
```

Locality-Sensitivity Hashing - Find similar songs based on lyrical content

In [51]:
```
# Read in dataset and create a dataframe for lyrics only
#song_lyrics = pd.read_csv('/content/drive/MyDrive/Computational tools for DS/spotify_so
song_lyrics = pd.read_csv('data/spotify_songs_lyrics.csv')
lyrics_data = song_lyrics.iloc[:,3]
lyrics_percent = lyrics_data.sample(frac=1)
```

In [52]:
```
# Finding songs in both datasets
data['both'] = data.name.isin(song_lyrics['track_name']).astype(str)
tmp = data[data['both']=='True']
```

In [53]:
```
# Hashes a list of strings
def listhash(l,seed):
        val = 0
        for e in l:
                val = val ^ mmh3.hash(e, seed)
```

```
        return val

q = 5 # length of shingle
k = 100 # number of minhashes
docs = {} #dictionary mapping document id to document contents
```

In [54]:
```
# Produces a list of shingles where each shingle is a list of q words
def shingle(aString, q, delimiter=' '):
    """
    Input:
        - aString (str): string to split into shingles
        - q (int)
        - delimiter (str): string of the delimiter to consider to split the input string
    Return: list of unique shingles
    """
    all_shingles = []
    if delimiter != '':
        words_list = aString.split(delimiter)
    else:
        words_list = aString
    for i in range (len(words_list)-q+1):
        all_shingles.append(delimiter.join(words_list[i:i+q]))
    return list(set(all_shingles))
```

In [55]:
```
# Takes list of shingles and seed for the hash function mapping the shingles and outputs
def minhash(shingles_list, seed):
    """
    Input:
        - shingles_list (list of str): set of hashes
        - seed (int): seed for listhash function
    Return: minhash of given shingles
    """
    minhash_value = None
    for aShingle in shingles_list:
        hashcode = listhash([aShingle], seed)
        if minhash_value == None or hashcode < minhash_value:
            minhash_value = hashcode
    return minhash_value
```

In [56]:
```
# Outputs k different minhases in an array
def minhash2(shingles_list, k):
    """
    Input:
        - shingles_list (list of str): set of hashes
        - k (int): seed for listhash function
    Return: sequence of k minhashes
    """
    all_minhash = []
    for i in range(k):
        all_minhash.append(minhash(shingles_list, i))
    return all_minhash
```

In [57]:
```
# Cleans Text
def clean_text(aString):
    output = aString.replace('\n','')
    output_list = output.split()
    output_list = [''.join(ch for ch in aWord if ch.isalnum()) for aWord in output_list]
    output_list = [s.lower() for s in output_list]
    output = ' '.join(output_list)
    return " ".join(output.split())
```


In [58]:
```
# Takes a dictionary and outputs a new dictionary consisting of song id's as keys and si
def signature(dict_docs, q = q, num_hashes = k):
    """
```

```
    Input:
        - dict_docs (dict of str:str): dictionary of {title:document}
        - q (int)
        - num_hashes (int)
    Return: dictionary consisting of document id’s as keys and signatures as values
    """
    dict_signatures = {}
    total_texts = len(list(dict_docs.keys()))
    counter = 1
    for key,text in dict_docs.items():
#         print(f'{counter}/{total_texts} - {key} - Processing...')
        doc_shingles = shingle(text, q)
        minhash_values = minhash2(doc_shingles, num_hashes)
        dict_signatures[key] = minhash_values
        counter += 1
    return dict_signatures
```

In [59]:

```
# Computes Jaccard similarity between two vectors
def jaccard(name1, name2, signatures_dict):
    """
    Input:
        - name1 (str): key of the first document S
        - name2 (str): key of the second document T
        - signatures_dict (dict of str:list): dictionary of signatures
    Return: Jaccard similarity between S and T
    """
    signatures_doc1 = np.array(signatures_dict[name1])
    signatures_doc2 = np.array(signatures_dict[name2])
    return len(np.intersect1d(signatures_doc1, signatures_doc2))/len(np.union1d(signatur
```

In [60]:

```
# Implement locality-sensitivity hashing which finds all pairs of documents whose estima
b,r = 20, 5
#b,r = 16, 4
assert k == b*r

def lsh(signatures_dict, jaccard_threshold=0.6, seed=42):
    lsh_dict = {}
    for key, values in signatures_dict.items():
        blocks = np.split(np.array(values), b)
        blocks_hash_values = []
        for aBlock in blocks:
            blocks_hash_values.append(mmh3.hash(aBlock, seed))
        lsh_dict[key] = blocks_hash_values
    list_keys = list(lsh_dict.keys())
    similar_items = {}

    for i in range (len(list_keys)-1):
        for j in range (i+1, len(list_keys)):
            common_values = np.intersect1d(lsh_dict[list_keys[i]], lsh_dict[list_keys[j]]
            if len(common_values) > 0:
                # we found a candidate
                similarity_score = jaccard(list_keys[i], list_keys[j], signatures_dict)
                if similarity_score >= jaccard_threshold:
                    similar_items[(list_keys[i], list_keys[j])] = similarity_score

    return similar_items
```

In [61]:

```
lyrics = lyrics_percent.astype(str)
i = 0
for song in lyrics:
    docs[song] = str(clean_text(song))
lyrics = lyrics.to_dict() #dictionary mapping document id to document contents
```

In [62]:

```
dict_signatures_lyrics = signature(lyrics)
```

In [63]:
```
keysNone = [k for k, v in dict_signatures_lyrics.items() if (None in v or 'None' in v or
keysNotNone = [x for x in dict_signatures_lyrics.keys() if x not in keysNone]
NoneRem = {key: dict_signatures_lyrics[key] for key in keysNotNone}
```

In [123…
```
found_similar_items_with_lsh = lsh(NoneRem)
found_similar_items_keys = found_similar_items_with_lsh.keys()
```

In [65]:
```
def LSH_find_sim_song(song_id,dict):
    for key, value in dict.items():
        if key[0] == song_id:
            return key[1]
        elif key[1] == song_id:
            return key[0]
        else:
            return 'Not Found'
```

In [66]:
```
def LSH_pred(song_id,dict):
    song_pair = LSH_find_sim_song(song_id,dict)
    LSH_list = []
    if song_pair == 'Not Found':
        return LSH_list
    LSH_list.append(data.iloc[song_pair,14])
    return LSH_list
```

In [89]:
```
# Testing - find similar song given song input using LSH
#print(LSH_find_sim_song(14959,found_similar_items_with_lsh))
#print(LSH_pred(14959,found_similar_items_with_lsh))
#print(LSH_pred(59,found_similar_items_with_lsh))
```

Ensemble Methods - Combining multiple models

Example 1

In [116…
```
data['name'][249:250]
```

Out[116]:
```
249    Tears
Name: name, dtype: object
```

In [115…
```
# Ensemble to combine predictions from all models and count and list the most common son
predictions = list()
rand_song_id = 249 # Song is in both datasets
num_songs = 5
decision_pred = recom(rand_song_id,tree,X_reduced_tree,data) # decision tree prediction
cosine_pred = cosine_sim_pred(rand_song_id,num_songs)
clus_pred = cluster_pred(rand_song_id,num_songs)
LSH_song_pred = LSH_pred(rand_song_id,found_similar_items_with_lsh)
predictions = cosine_pred + clus_pred+ decision_pred + LSH_song_pred
def hard_voting(predictions):
    c = Counter(predictions)
    return [k for k, v in c.items() if v == c.most_common(1)[0][1]]
print(hard_voting(predictions))
print(len(hard_voting(predictions)))
```

```
['Tears', 'Lover Man', 'Relaxing With Lee - Take 6 / Take 3 / Master Take', 'Ma Chanso
n', 'Wie man Freunde gewinnt - Die Kunst, beliebt und einflussreich zu werden, Kapitel 5
2', 'Часть 37.3 & Часть 38.1 - Зеленые холмы Африки', 'Sorge dich nicht - lebe! - Die Ku
nst, zu einem von Ängsten und Aufregungen befreiten Leben zu finden, Kapitel 7', 'Часть
55.2 - На Западном фронте без перемен', 'If We Must Die (Introduction)', 'Часть 238.4 &
Часть 239.1 - Триумфальная арка', 'Nit Nai Rut Ki Bahar Aai', 'Smack Dab In The Middle',
'Para mi Gaucha - Instrumental (Remasterizado)', "There's No Business Like Show Busines
```

```
s", 'Going to Memphis', "I Got Cross de River O' Jordan - Mix Two", 'Anna (El Negro Zumb
ón)', 'Hari Merdeka (Cover Version)', "Totor T'as Tort", 'Minor Blues - Remastered']
20
```

Example 2

In [107…

```python
data['name'][8465:8466]
```

Out[107]:

```
8465    Good Times
Name: name, dtype: object
```

In [108…

```python
# Ensemble to combine predictions from all models and count and list the most common son
predictions = list()
rand_song_id = 8465 # Song is in both datasets
num_songs = 5
decision_pred = recom(rand_song_id,tree,X_reduced_tree,data) # decision tree prediction
cosine_pred = cosine_sim_pred(rand_song_id,num_songs)
clus_pred = cluster_pred(rand_song_id,num_songs)
LSH_song_pred = LSH_pred(rand_song_id,found_similar_items_with_lsh)
predictions = cosine_pred + clus_pred+ decision_pred + LSH_song_pred
def hard_voting(predictions):
    c = Counter(predictions)
    return [k for k, v in c.items() if v == c.most_common(1)[0][1]]
print(hard_voting(predictions))
print(len(hard_voting(predictions)))
```

```
['Good Times', "Be Careful, It's My Heart", 'Here Comes My Baby - Stereo Version', 'What
Child Is This/The Holly And The Ivy - Medley / Remastered 2006', "Bringin' On The Heartb
reak", 'Часть 22.2 - Обратный путь', 'Das ist bei uns nicht möglich, Kapitel 132', 'Kapi
tel 322 - Die drei Ehen der Grand Sophy', 'Часть 30.3 - По ком звонит колокол', 'Kapitel
9 - Dschungelbuch', 'Listen to the Sirens', 'After The Storm', 'This Boy - Remastered 20
09', 'En El Último Rincón', 'Soul Provider', "I Just Ain't Been Able", "I'm Not The Only
One", "Missin' You Crazy", 'Occapella', 'Move Over']
20
```

Example 3

In [109…

```python
data['name'][2718:2719]
```

Out[109]:

```
2718    Am I Blue?
Name: name, dtype: object
```

In [110…

```python
# Ensemble to combine predictions from all models and count and list the most common son
predictions = list()
rand_song_id = 2718 # Song is in both datasets
num_songs = 5
decision_pred = recom(rand_song_id,tree,X_reduced_tree,data) # decision tree prediction
cosine_pred = cosine_sim_pred(rand_song_id,num_songs)
clus_pred = cluster_pred(rand_song_id,num_songs)
LSH_song_pred = LSH_pred(rand_song_id,found_similar_items_with_lsh)
predictions = cosine_pred + clus_pred+ decision_pred + LSH_song_pred
def hard_voting(predictions):
    c = Counter(predictions)
    return [k for k, v in c.items() if v == c.most_common(1)[0][1]]
print(hard_voting(predictions))
print(len(hard_voting(predictions)))
```

```
['Am I Blue?', 'Pretty Thing', 'Mere Samnewali Khidki Mein - Instrumental', 'Violent Por
nography', 'Slob On My Nob (feat. Project Pat)', 'Voice-Over Intro Quincy Jones Intervie
w #2/Quincy Jones Interview #2 / Voice-Over Intro Billie Jean (Demo)', 'Pregnancy - Liv
e', 'Часть 85.2 - Фиеста', 'Часть 195.4 & Часть 196.1 - По ком звонит колокол', 'Часть 2
41.2 - Триумфальная арка', 'Goldberg Variations, BWV 988: Variation 7 a 1 ovvero 2 Cla
v.', 'I Will Say Goodbye', 'Till We Meet Again', 'Die Lustige Witwe (2001 - Remaster), A
ct II: Dialog: Valencienne, bitte geben...Mein Freund, Vernunft! (Valencienne/Camille)',
```

```
"na sera 'e maggio", 'Te Quiero Asi (If I Love You So)', 'Far From Home', 'Serenade In B
lue', 'Lieder und Gesänge aus der Jugendzeit (Excerpts): Book 2, No. 2, Ich ging mit Lus
t durch einen grünen Wald', 'Descriptions automatiques: II. Sur une lanterne']
20
```

Example 4

In [111…

```
data['name'][612:613]
```

Out[111]:

```
612    Heat Wave
Name: name, dtype: object
```

In [122…

```
# Ensemble to combine predictions from all models and count and list the most common son
predictions = list()
rand_song_id = 612 # Song is in both datasets
num_songs = 5
decision_pred = recom(rand_song_id,tree,X_reduced_tree,data) # decision tree prediction
cosine_pred = cosine_sim_pred(rand_song_id,num_songs)
clus_pred = cluster_pred(rand_song_id,num_songs)
LSH_song_pred = LSH_pred(rand_song_id,found_similar_items_with_lsh)
predictions = cosine_pred + clus_pred+ decision_pred + LSH_song_pred
def hard_voting(predictions):
    c = Counter(predictions)
    return [k for k, v in c.items() if v == c.most_common(1)[0][1]]
print(hard_voting(predictions))
print(len(hard_voting(predictions)))
```

```
['Heat Wave', 'Yesterdays', 'Old Hippie', 'Chief Rocka', 'Sousedská / Furiant / Však Nám
Tak Nebude / Hezká Jsi Andulko', 'Часть 88.2 - На Западном фронте без перемен', 'Von der
Renaissance bis heute, Kapitel 40', 'Acte 1, scène 9', 'Von der Renaissance bis heute, K
apitel 2', 'Часть 220.2 - Триумфальная арка', 'Sansar Ke Aadhar Daya Humpe Dikhao', 'Roz
ika', 'Itália e Abissínia', "Let's Do It", 'Roundhouse', 'Ghir Ghir Ke Aai Badariya', 'V
ouna mou xalilosete', 'Hotel Hispaniola', "The Soldier's Tale Suite: I. The Soldier's Ma
rch", 'Venganza - Remasterizado']
20
```

In [ ]:

References

[1] Maarten Grootendorst. “nine distance measures in data science.”, 2021. URL https://towardsdatascience.com/9-distance-measures-in-data-science-918109d069fa.

[2] Santhosh Hari. "locality sensitive hashing for similar item search", 2018. URL https://towardsdatascience.com/locality-sensitive-hashing-for-music-search-f2f1940ace23.

[3] Graham Harrison. “how to attain a deep understanding of soft and hard voting in ensemble machine leaning methods.”, 2022. URL https://towardsdatascience.com/how-to-attain-a-deep-understanding-of-soft-and-hard-voting-in-ensemble-machine-lean

[4] F.O. Isinkaye. "recommendation systems: Principles, methods and evaluation". *Egyptian Informatics Journal*, 16(3):261–273, 2015. doi: 10.1016/j.eij.2015.06.005. URL https://doi.org/10.1016/j.eij.2015.06.005.

[5] Fatih Karabiber. “cosine similarity.”, 2022. URL https://www.learndatasci.com/glossary/cosine-similarity/.

[6] Jurij Leskovec. Locality-sensitivity hashing. pages 99–103. Cambridge University Press, 2022.

[7] Vatsal Mavani. URL https://www.kaggle.com/code/vatsalmavani/music-recommendation-system-using-spotify-dataset.

[8] Muhammad Nakhaee. URL https://www.kaggle.com/datasets/imuhammad/audio-features-and-lyrics-of-spotify-songs?resource=download.

[9] Lior Rokach and Oded Maimon. Decision trees. pages 165–166. Springer, 2005.

[10] The Upwork Team. What content-based filtering is amp; why you should use it., 2021. URL https://www.upwork.com/resources/what-is-content-based-filtering.

[11] Vatsal. “recommendation systems explained.”, 2022. URL https://towardsdatascience.com/recommendation-systems-explained-a42fc60591ed.