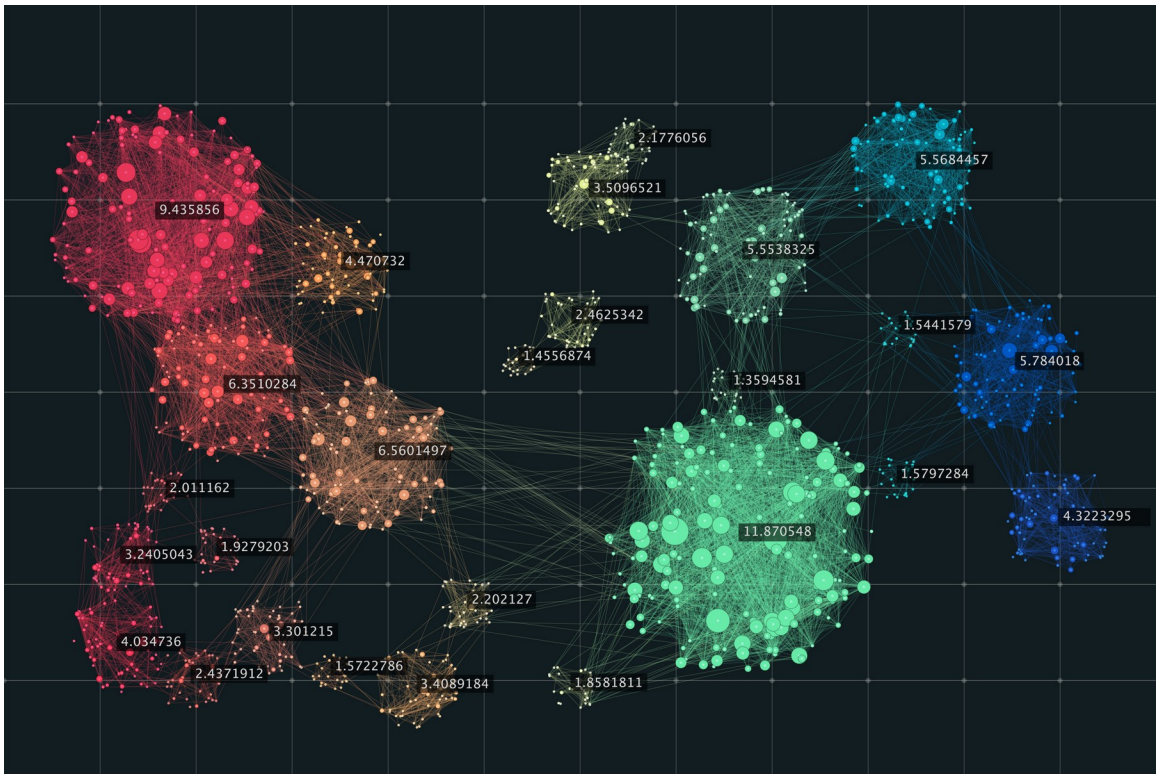


Music Recommender System

DATA MINING PROJECT

Atharva Shekatkar | 20BDS0067
Arnab Banik | 20BDS0038
Parshva Maniar | 20BDS0058
Tejas Rokade | 20BDS0033

Abstract

Recommendation systems are widely used in various fields, whether it be on music and video streaming websites or for businesses. The inspiration for this project came from having an overview of clustering machine learning algorithms and the curiosity about how various recommendations systems worked on various streaming sites. Our team's goal is to analyze and create such a recommendation system, specifically for music, using unsupervised clustering machine learning algorithms like BIRCH, K-means, Mean-shift and KMedoids algorithms. The most accurate model will be used to give recommendations based on a few songs the user has already listened to. The data being used will be Spotify's dataset available on Kaggle, which gives information about a large number of songs that are available on Spotify. The data will be preprocessed, visualized and then classified into clusters while training the model.

Contents

INTRODUCTION

This document lays out a project plan for the development of “Music Recommender System”.

The plan will include a summary of the system functionality, scope of the project from the perspective of the team, the approach to developing the project and metrics and measurements that will be recorded throughout the project.

This document will also cover a detailed analysis of the procedure of development of the project, including obtaining the dataset, preprocessing, analysis of various algorithms, evaluating the best algorithm and then obtaining the final output.

PURPOSE

The goal of this project is to make a music recommender system, which will recommend new songs to the user based on a few songs the user has listened to previously.

SCOPE

The “Music Recommender System” is a machine learning model which will help users find new music which is similar to the user’s preference in music. The model will be hosted on a website online.

A user must call the model and provide the names of the songs they listened to. The model then identifies and finds each song in the database and then recommends songs similar to it based on various calculations.

The model should be free to call by anyone.

ACQUIRING THE DATASET

The dataset used for training various models for the music recommender system is taken from Kaggle. The dataset contains multiple files, which have data arranged according to genre and year and have the main training dataset. Each one contains various attributes for each entry such as the name of the song, the artist, whether the song is explicit, whether the song is suitable for dancing, etc.

The dataset has both categorical as well as continuous variables and is thus suitable for making a recommendation system which the team is aiming for.

The dataset has been uploaded on the GitHub repository for this project, the link for which can be found in the Links section.

EXPLORATORY DATA ANALYSIS

Under this section, various stages of exploratory data analysis are discussed. We first discuss data cleaning, followed by data visualization.

CHECKING THE DATASET

Here, we first check the dataset to see what type of values it contains and whether there are any null entries in any column. This can easily be done using the **info** function from **pandas** library.

In [5]:
```
#EDA
#getting info about the datasets
song_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 174389 entries, 0 to 174388
Data columns (total 19 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   acousticness      174389 non-null  float64
 1   artists           174389 non-null  object 
 2   danceability      174389 non-null  float64
 3   duration_ms       174389 non-null  int64  
 4   energy            174389 non-null  float64
 5   explicit          174389 non-null  int64  
 6   id                174389 non-null  object 
 7   instrumentalness  174389 non-null  float64
 8   key               174389 non-null  int64  
 9   liveness          174389 non-null  float64
 10  loudness          174389 non-null  float64
 11  mode              174389 non-null  int64  
 12  name              174389 non-null  object 
 13  popularity        174389 non-null  int64  
 14  release_date      174389 non-null  object 
 15  speechiness       174389 non-null  float64
 16  tempo             174389 non-null  float64
 17  valence           174389 non-null  float64
 18  year              174389 non-null  int64  
dtypes: float64(9), int64(6), object(4)
memory usage: 25.3+ MB
```

In [6]:
```
genre_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3232 entries, 0 to 3231
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   genres            3232 non-null   object 
 1   acousticness      3232 non-null   float64
 2   danceability      3232 non-null   float64
 3   duration_ms       3232 non-null   float64
 4   energy            3232 non-null   float64
 5   instrumentalness  3232 non-null   float64
 6   liveness          3232 non-null   float64
 7   loudness          3232 non-null   float64
 8   speechiness       3232 non-null   float64
 9   tempo             3232 non-null   float64
 10  valence           3232 non-null   float64
 11  popularity        3232 non-null   float64
 12  key               3232 non-null   int64  
 13  mode              3232 non-null   int64  
dtypes: float64(11), int64(2), object(1)
memory usage: 353.6+ KB
```

In [7]:
```
year_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 102 entries, 0 to 101
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   year              102 non-null    int64  
 1   acousticness      102 non-null    float64
 2   danceability      102 non-null    float64
 3   duration_ms       102 non-null    float64
 4   energy            102 non-null    float64
 5   instrumentalness  102 non-null    float64
 6   liveness          102 non-null    float64
 7   loudness          102 non-null    float64
 8   speechiness       102 non-null    float64
 9   tempo             102 non-null    float64
 10  valence           102 non-null    float64
 11  popularity        102 non-null    float64
 12  key               102 non-null    int64  
 13  mode              102 non-null    int64  
dtypes: float64(11), int64(3)
memory usage: 11.3 KB
```

As we can see, out of the total entries in each dataset, none are null. Therefore, we do not need to drop any rows or replace any entry for **NA** values.

DATA CLEANING

We saw that there are string values present in the **song_df** dataset, which will pose a hindrance while training the dataset. Therefore, we make a copy of the dataset and drop those columns.

```
In [15]: #We don't need columns like 'id', 'artists', etc for visualizing the data or training models.
         #Therefore, we shall drop them from all datasets
         non_numeric_cols = ['artists', 'id', 'release_date', 'name']

         song_df_num = song_df.copy()
         song_df_num.drop(columns=non_numeric_cols, inplace=True)
```

We do not drop any columns from the **genre_df** and **year_df** since those dataframes are only going to be used for visualization.

DATA VISUALIZATION

We try to visualize the data in an attempt to understand various trends in music over the period of time and how various features can be correlated to each other.

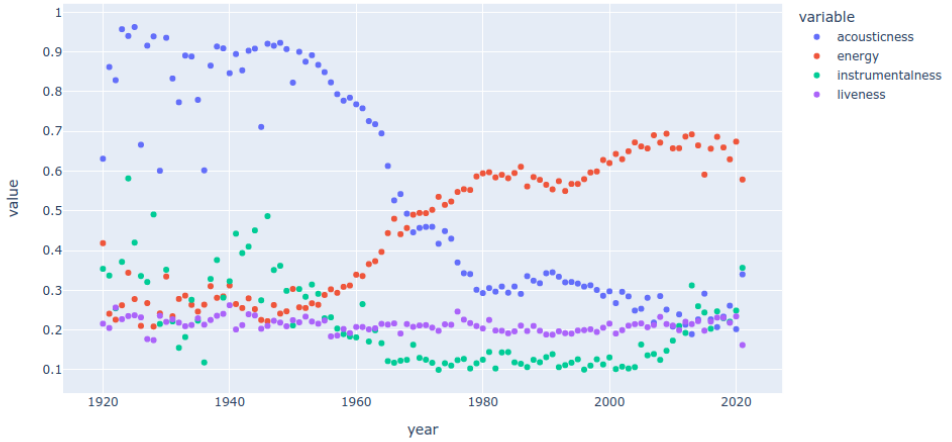
First, let us see how **acousticness**, **energy**, **instrumentalness** and **liveness** have varied in the past 100 years.

In [17]:
```
#Data visualization
import plotly.express as px

observe_features = ['acousticness', 'energy', 'instrumentalness', 'liveness']

#observing trend in above features as a function of year
fig = px.scatter(year_df, x = 'year', y = observe_features)

fig.show()
```

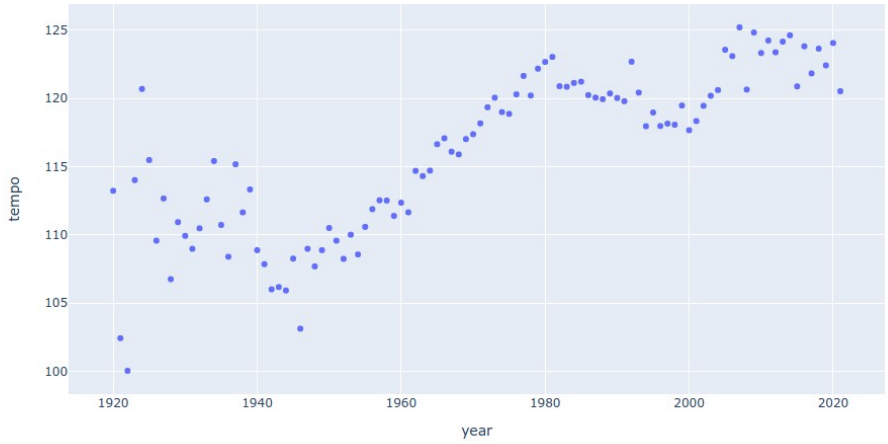


We see that in the past 100 years, **acousticness** in music has gone down, with a rise in **energy** in music. This may be because of advancements in audio engineering and various software available to create and edit music digitally. We also notice that **instrumentalness** and **liveness** have remained the same in music.

Next, let us see how **tempo** has varied in the past 100 years

```
In [18]: #observing trend in 'tempo' as a function of year
fig2 = px.scatter(year_df, x = 'year', y = 'tempo')

fig2.show()
```



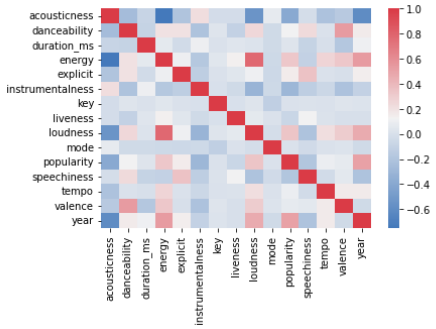
We see that **tempo** of music has increased considerably as well. This is in accordance with the increase in **energy** of the music, as usually energizing songs have higher tempo.

Finally, let's see the correlation matrix as a heatmap to see which features are correlated with each other.

```
In [19]: #Checking correlation for the dataset
corr_mat = song_df_num.corr()

cmap_set = sns.diverging_palette(250, 10, as_cmap=True)

figure = sns.heatmap(corr_mat, cmap=cmap_set)
```



From the above figure, we can see that **acousticness** is inversely correlated with **energy**, **loudness** as well as **year**. We can also make out that **energy** and **loudness** are positively correlated with each other.

This concludes the exploratory data analysis section. We shall now move onto training the models for song recommendation.

TRAINING THE MODELS

In this section, we first scale our dataset. Then, we train various models based on different algorithms on the scaled dataset and test them.

SCALING THE DATASET

We scale the dataset using **StandardScaler()** function available in the **sklearn.preprocessing** library.

```
In [85]: #Scaling the dataset
         from sklearn.preprocessing import StandardScaler
         scaler = StandardScaler()

In [86]: scaled_song_data = scaler.fit_transform(song_df_num)

In [87]: sample_song_df=song_df.sample(frac=0.1, random_state=1566)
         sample_song_df_num=sample_song_df.copy()
         sample_song_df_num.drop(columns=non_numeric_cols,inplace=True)
         scaled_sample_data=scaler.fit_transform(sample_song_df_num)
```

We also make a smaller dataset out of the larger dataset as some of the algorithms that we test do not scale well for larger datasets. Hence, we take a random sample of 10% of the total dataset and also use a seed value in **random_state** variable so the outcomes can be reproduced and store it in **sample_song_df**. We store the columns with numerical data as before and then, scale both the sample song dataset as well as the original song dataset.

IMPLEMENTING THE ALGORITHMS

1. KMeans:

The KMeans algorithm is a clustering algorithm which clusters data by trying to separate samples in n groups of equal variance. It requires the number of clusters to be specified. It also scales well to large number of samples and hence is used across a large number of applications. It is one of the most popular clustering algorithms.

For our project, we use the **KMeans** module from **sklearn.clustering** library and feed it our dataset using the **fit** function. We use the randomly sampled scaled dataset for fitting as it would give an unfair advantage to KMeans while evaluation if we feed it the entire dataset. We then use the model to make predictions on our entire dataset and add the predicted cluster numbers, in a new column, to the **song_df** dataframe.

Following is the code:

```
In [130]: from sklearn.cluster import KMeans
          kmeans = KMeans(n_clusters = 20, max_iter = 500, algorithm='auto')
          model = kmeans.fit(scaled_sample_data)

In [131]: kmeans_song_cluster_labels = model.predict(scaled_song_data)
          song_df['kmeans_cluster_labels'] = kmeans_song_cluster_labels
          song_df.head()
```

Out[131]:

id	instrumentalness	key	liveness	loudness	mode	name	popularity	release_date	speechiness	tempo	valence	year	kmeans_cluster_labels
/W3FcF8AEI	0.000522	5	0.3790	-12.628	0	Keep A Song In Your Soul	12	1920	0.0936	149.976	0.6340	1920	12
5H8ZI9w30f	0.026400	5	0.0809	-7.261	0	I Put A Spell On You	7	1920-01-05	0.0534	86.889	0.9500	1920	12
qI3oYzuhne	0.000018	0	0.5190	-12.098	1	Golfing Papa	4	1920	0.1740	97.600	0.6890	1920	4
oaxY0fpwfh	0.801000	2	0.1280	-7.311	1	True House Music - Xavier Santos & Carlos Gomi...	17	1920-01-01	0.0425	127.997	0.0422	1920	9
nAHfdsLejp	0.000246	10	0.4020	-6.036	0	Xuniverxe	2	1920-10-01	0.0768	122.076	0.2990	1920	0

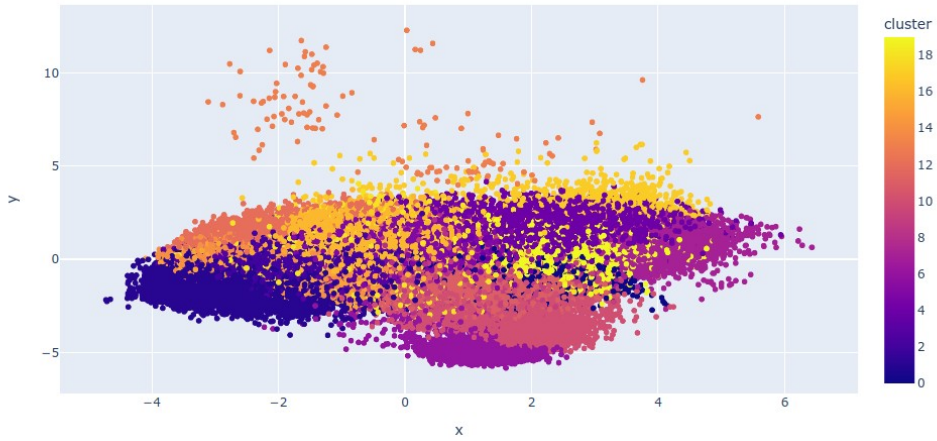
We can visualize the clusters using **PCA** (Principal Component Analysis) and **plotly** modules.

Following is the plot we get:

```
In [92]: #Visualizing Kmeans clustering

In [93]: from sklearn.decomposition import PCA
         from sklearn.pipeline import Pipeline
         pca_pipeline = Pipeline([('scaler', StandardScaler()), ('PCA', PCA(n_components=2))])
         song_embedding = pca_pipeline.fit_transform(song_df_num)
         projection = pd.DataFrame(columns=['x', 'y'], data=song_embedding)
         projection['title'] = song_df['name']
         projection['cluster'] = song_df['kmeans_cluster_labels']

         fig = px.scatter(projection, x='x', y='y', color='cluster', hover_data=['x', 'y', 'title'])
         fig.show()
```



It should be noted that the plot is interactive and displays the song name and cluster number when the mouse pointer hovers over the dot.

However, one must run the notebook to make the plot visible as saving the plot in the file increases its size exponentially due to its interactive nature. This is the case for all clustering plots in this **ipnyb** file.

2. BIRCH:

Birch is a clustering algorithm which builds a tree called the Clustering Feature Tree (CFT) for the given dataset. The data is compressed to a set of Clustering Feature nodes (CF nodes). The CF Nodes have a number of subclusters called Clustering Feature subclusters and these subclusters can have CF nodes as children. The CF Subclusters hold the necessary information for clustering which prevents the need to hold the entire input data in memory.

For our project, we use the **Birch** module from the **sklearn.clustering** library and feed our sampled scaled dataset to it. We then predict the cluster labels for the entire song data using the Birch model and add the predicted cluster labels to the **song_df** dataframe under the column **birch_cluster_labels**

Following is the code for the same:

```
In [137]: from sklearn.cluster import Birch
model = Birch(branching_factor = 50,threshold = 1.5, n_clusters = 20)
model.fit(scaled_sample_data)
birch_song_cluster_labels = model.predict(scaled_song_data)
song_df['birch_cluster_labels'] = birch_song_cluster_labels
```

```
In [138]: song_df.head()
```

Out[138]:

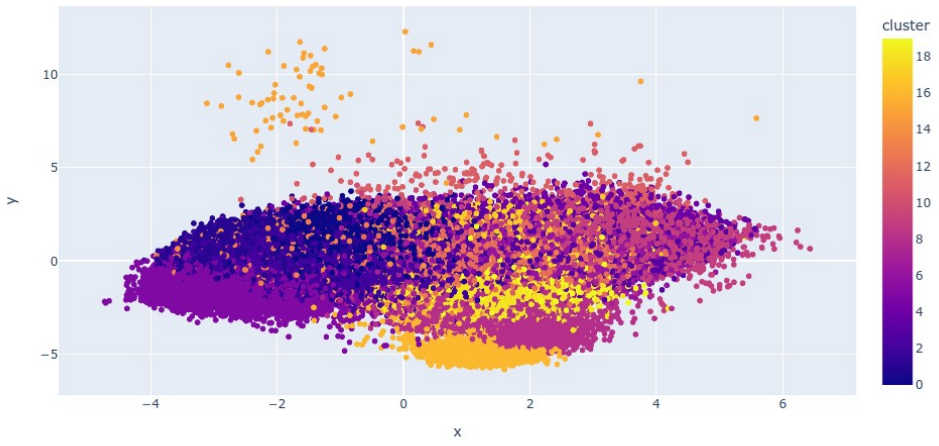
strumentalness	key	liveness	...	mode	name	popularity	release_date	speechiness	tempo	valence	year	kmeans_cluster_labels	birch_cluster_labels
0.000522	5	0.3790	...	0	Keep A Song In Your Soul	12	1920	0.0936	149.976	0.6340	1920	12	7
0.026400	5	0.0809	...	0	I Put A Spell On You	7	1920-01-05	0.0534	86.889	0.9500	1920	12	3
0.000018	0	0.5190	...	1	Golfing Papa	4	1920	0.1740	97.600	0.6890	1920	4	18
0.801000	2	0.1280	...	1	True House Music - Xavier Santos & Carlos Gomi...	17	1920-01-01	0.0425	127.997	0.0422	1920	9	14
0.000246	10	0.4020	...	0	Xuniverxe	2	1920-10-01	0.0768	122.076	0.2990	1920	0	5

Similar to KMeans, we can also visualize the Birch clustering using PCA.

We get the following plot:

```
In [139]: pca_pipeline = Pipeline([('scaler', StandardScaler()), ('PCA', PCA(n_components=2))])
song_embedding = pca_pipeline.fit_transform(song_df_num)
projection = pd.DataFrame(columns=['x', 'y'], data=song_embedding)
projection['title'] = song_df['name']
projection['cluster'] = song_df['birch_cluster_labels']

fig = px.scatter(projection, x='x', y='y', color='cluster', hover_data=['x', 'y', 'title'])
fig.show()
```



Once again, the plot is interactive and hence the notebook must be run to generate it each time. We see that we get similar clustering to that of KMeans here.

3. Mean Shift:

Mean-shift clustering is a clustering algorithm which aims to discover blobs in a smooth density of samples. It is a centroid based algorithm, which works by updating the candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate the near-duplicates to form the final set of centroids.

For our project, we use the **MeanShift** module available in the **sklearn.clustering** library. We feed it the scaled sampled dataset for training and then make predictions on the entire scaled dataset. However, the Mean-shift algorithm does not scale well and hence even for the sampled dataset, it takes quite an amount of time (up to 1-2 minutes) for the training to complete.

Following is the code for the same:

In [142]:
```
from sklearn.cluster import MeanShift
model = MeanShift(n_jobs=-1)
model.fit(scaled_sample_data)
meanshift_song_cluster_labels = model.predict(scaled_song_data)
song_df['meanshift_cluster_labels'] = meanshift_song_cluster_labels
song_df.head()
```

Out[142]:

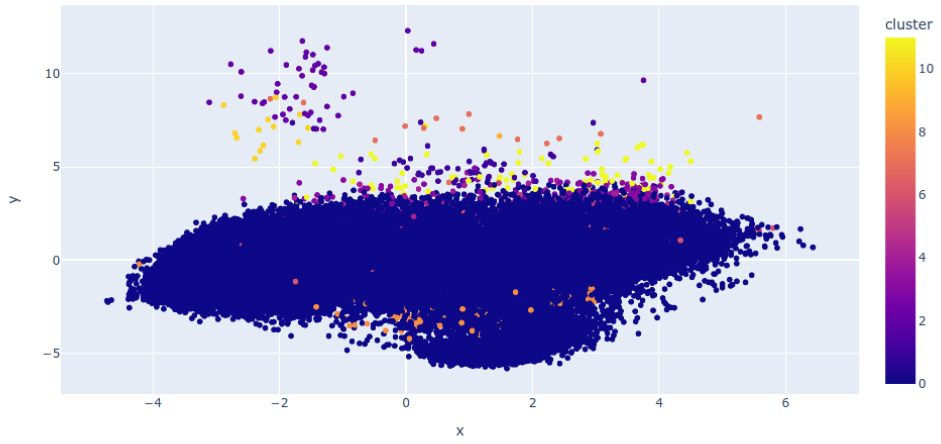
ıy	liveness	...	name	popularity	release_date	speechiness	tempo	valence	year	kmeans_cluster_labels	birch_cluster_labels	meanshift_cluster_labels
5	0.3790	...	Keep A Song In Your Soul	12	1920	0.0936	149.976	0.6340	1920	12	7	0
5	0.0809	...	I Put A Spell On You	7	1920-01-05	0.0534	86.889	0.9500	1920	12	3	0
0	0.5190	...	Golfing Papa	4	1920	0.1740	97.600	0.6890	1920	4	18	0
2	0.1280	...	True House Music - Xavier Santos & Carlos Gomi...	17	1920-01-01	0.0425	127.997	0.0422	1920	9	14	0
0	0.4020	...	Xuniverxe	2	1920-10-01	0.0768	122.076	0.2990	1920	0	5	0

We can see that all the first 5 songs are in cluster 0, which indicates that Mean-shift was not able to cluster the data properly. We can confirm our suspicions by visualization in the same way that we have done earlier.

We visualize in the following way:

```
In [143]: pca_pipeline = Pipeline([('scaler', StandardScaler()), ('PCA', PCA(n_components=2))])
          song_embedding = pca_pipeline.fit_transform(song_df_num)
          projection = pd.DataFrame(columns=['x', 'y'], data=song_embedding)
          projection['title'] = song_df['name']
          projection['cluster'] = song_df['meanshift_cluster_labels']

          fig = px.scatter(projection, x='x', y='y', color='cluster', hover_data=['x', 'y', 'title'])
          fig.show()
```



We can see that indeed, Mean-shift was not able to form proper clusters and most of the songs are grouped in a single cluster.

4. KMedoids:

KMedoids is a clustering algorithm which is related to the KMeans algorithm. While KMeans tries to minimize the within cluster sum-of-squares, KMedoids tries to minimize the sum of distances between each point and the medoid of it's cluster. The medoid is a datapoint, unlike the centroid, which has the least total distance to the other members of its cluster.

For our project, we use the **KMedoids** module available in **sklearn_extra.cluster** library and pass it our scaled sampled dataset. We then use the trained model to predict the clusters for the entire dataset, and add the predicted labels to the **song_df** dataframe.

Following is the code for the same:

In [146]:
```
from sklearn_extra.cluster import KMedoids
model = KMedoids(n_clusters=20)
model.fit(scaled_sample_data)
kmedoids_song_cluster_labels = model.predict(scaled_song_data)
song_df["kmedoids_cluster_labels"] = kmedoids_song_cluster_labels
song_df.head()
```

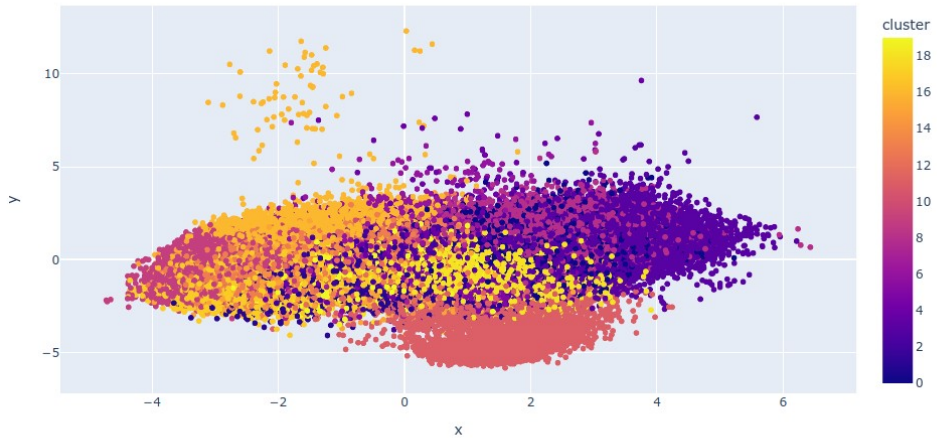
Out[146]:

speechiness	tempo	valence	year	kmeans_cluster_labels	birch_cluster_labels	meanshift_cluster_labels	KMediods_cluster_labels	kmedoids_cluster_labels
0.0936	149.976	0.6340	1920	12	7	0	2	2
0.0534	86.889	0.9500	1920	12	3	0	5	5
0.1740	97.600	0.6890	1920	4	18	0	5	5
0.0425	127.997	0.0422	1920	9	14	0	16	16
0.0768	122.076	0.2990	1920	0	5	0	4	4

We then try to visualize the data using PCA in the same way that we have done earlier.

In [147]:
```
pca_pipeline = Pipeline([('scaler', StandardScaler()), ('PCA', PCA(n_components=2))])
song_embedding = pca_pipeline.fit_transform(song_df_num)
projection = pd.DataFrame(columns=['x', 'y'], data=song_embedding)
projection['title'] = song_df['name']
projection['cluster'] = song_df['kmedoids_cluster_labels']

fig = px.scatter(projection, x='x', y='y', color='cluster', hover_data=['x', 'y', 'title'])
fig.show()
```



MODEL EVALUATION

We evaluate each model based on the Calinski-Harabasz Index, which is also known as the Variance Ratio Criterion. A higher Calinski-Harabasz score relates to a model with better defined clusters. The index is the ratio of the sum of between-clusters dispersion and of within-cluster dispersion for all clusters, where dispersion is defined as the sum of distances squared.

For calculating the score, we use the **metrics** module from the **sklearn** library. We store the Calinski-Harabasz scores in a dictionary and then print the scores to compare which is the best model for our data.

Following is the code:

```
In [224]: #evaluating the models
          from sklearn import metrics
          evaluation_scores = {}
          evaluation_scores['kmeans'] = metrics.calinski_harabasz_score(scaled_song_data, song_df['kmeans_cluster_labels'])
          evaluation_scores['birch'] = metrics.calinski_harabasz_score(scaled_song_data, song_df['birch_cluster_labels'])
          evaluation_scores['meanshift'] = metrics.calinski_harabasz_score(scaled_song_data, song_df['meanshift_cluster_labels
          evaluation_scores['kmedoids'] = metrics.calinski_harabasz_score(scaled_song_data, song_df['kmedoids_cluster_labels'

In [227]: #printing the evaluated score for each model
          #higher is better
          for key, value in evaluation_scores.items():
              print(key, ':', value)

          kmeans : 12657.830713318253
          birch : 8921.553256445315
          meanshift : 618.0836570244139
          kmedoids : 6992.881052682993
```

From the output, we can see that KMeans dominates all the algorithms, with Birch and KMedoids being within close range of each other. The meanshift algorithm, however, lags behind with a very low score compared to the other algorithms. This matches our previous results as meanshift was not able to form proper clusters for our data.

Thus, given the evaluation, we select KMeans as our best algorithm and hence we shall use KMeans for the recommendation of songs, and hence we will use KMeans to interpret the final results of the project.

OUTPUT INTERPRETATION

For getting the output, which in this case is song recommendations based on songs we have listened to in the past, we must first write a few helper functions so that we can get our output properly

Following are the functions and their explanations in short:

FLATTEN_LIST_OF_DICT

This function gets a list of dictionaries, flattens (makes it a single dictionary with multiple key-value pairs, instead of nesting dictionaries) it and returns this flattened dictionary.

GET_SONG_DATA

For a given song and the song database, returns the features of the given song if it is present in the database. If not present, it returns **None** value.

GET_MEAN_VECTOR

For a given list of songs and of the song database, it creates a list of song features for each song. It then gets the mean for each feature and then returns this vector of mean feature value to the calling function.

RECOMMEND_SONGS

The main function which calls the helper functions mentioned above to return a list of recommended songs. It takes a song list, the song database and number of songs to be recommended, which is defaulted to 10.

It first flattens the song list, then gets the mean vector for the songs provided. This mean vector is then scaled and we use the KMeans model to predict the cluster label for this scaled mean vector. We then shortlist the songs in the songs database having the same cluster label as that of the mean vector. We get the cosine distances between

the mean vector and each of the shortlisted songs and sort the distances. We then get the closest n songs which were not present in the songs list past to the function initially and return their **name**, **artist** and **year** value to the user.

Test run 1:

```
In [110]: recommend_songs([{'name': 'Come As You Are', 'year':1991},
                           {'name': 'Smells Like Teen Spirit', 'year': 1991},
                           {'name': 'Lithium', 'year': 1992},
                           {'name': 'All Apologies', 'year': 1993},
                           {'name': 'Stay Away', 'year': 1993}],  song_df)

          Warning: Stay Away does not exist in our database

Out[110]: [{'name': 'Low Gravy',
            'year': 1924,
            'artists': '[\'The Chenille Sisters\', "James Dapogny\'s Chicago Jazz Band"]'},
           {'name': 'La Mina del Ford - Remasterizado',
            'year': 1924,
            'artists': "['Ignacio Corsini']"},
           {'name': "Toodles - My Baby's Baby Blue Eyes",
            'year': 1924,
            'artists': "['George Olsen']"},
           {'name': 'Pero Hay Una Melena - Remasterizado',
            'year': 1924,
            'artists': "['Francisco Canaro']"},
           {'name': "I've Got My Love to Keep Me Warm",
            'year': 1933,
            'artists': "['Billie Holiday']"},
           {'name': "You Run Your Mouth, I'll Run My Business",
            'year': 1926,
            'artists': "['Fats Waller']"},
           {'name': 'Un Agent Courait', 'year': 1921, 'artists': "['Georgius']"},
           {'name': "It's Time To Jump And Shout",
            'year': 1939,
            'artists': "['Jimmie Lunceford']"},
           {'name': 'Lost Love', 'year': 1926, 'artists': "['Fats Waller']"},
           {'name': "The Lion's Confession - Outtake; Alternate Arrangement",
            'year': 1939,
            'artists': "['MGM Studio Orchestra']"}]
```

Test Run 2:

```
In [112]: recommend_songs([{'name': 'Aa Chal Ke Tujhe', 'year': 1964},
                           {'name': 'Ek Main Aur Ek Tu', 'year': 1975},
                           {'name': 'Yeh Kya Hua', 'year': 1971},
                           {'name': 'Is Mod Se Jate Hain - Duet', 'year': 1975},
                           {'name': 'Roop Tera Mastana', 'year': 1969}], song_df)

Out[112]: [{'name': "Don't Let The Sun Go Down On Me",
            'year': 1974,
            'artists': "['Elton John']"},
           {'name': 'Часть 5.2 - Обратный путь',
            'year': 1931,
            'artists': "['Эрих Мария Ремарк']"},
           {'name': 'When the Morning Comes',
            'year': 1973,
            'artists': "['Daryl Hall & John Oates']"},
           {'name': 'Someday My Prince Will Come',
            'year': 1960,
            'artists': "['Bill Evans', 'Scott LaFaro', 'Paul Motian']"},
           {'name': 'Beautiful Girl (with Lenny Hayton & His Orchestra)',
            'year': 1932,
            'artists': "['Bing Crosby', 'Lenny Hayton & His Orchestra']"},
           {'name': 'Se Acabo la Yeta', 'year': 1930, 'artists': "['Pedro Maffia']"},
           {'name': 'So Amazing', 'year': 1986, 'artists': "['Luther Vandross']"},
           {'name': 'Sophisticated Savage', 'year': 1951, 'artists': "['Les Baxter']"},
           {'name': 'Goodbye Blue Sky', 'year': 1979, 'artists': "['Pink Floyd']"},
           {'name': 'When I Think Of You', 'year': 1986, 'artists': "['Janet Jackson']"}]
```

Thus, it recommends appropriate songs to us based on the given input songs. Though the songs may not be of the same language, they would share similar attributes like **energy**, **acousticness**, etc.

CONCLUSION

Spotify stores the metadata and audio features for songs which we can use to build music recommendation systems. We have successfully accomplished our goal, which was to build a music recommendation system by implementing various ML models, evaluating and comparing their performance for our project, and choosing the best model to use for the recommendation system.

LINKS

Github repository – all the code as well as the datsets used can be found on the github repository for this project.

BIBLIOGRAPHY

KMeans Algorithm paper – IEEE, 1982

Birch Algorithm paper

MeanShift Algorithm paper – IEEE, 2002

KMedoids Algorithm paper – 1990

PCA (Principal Component Analysis) algorithm paper – 1901