MUSIC POPULARITY PREDICTION

PHASE 2

NAME	ROLL NUMBER
JOSHITA MALLA	AM.EN. U4CSE20032
M SAMHITA	AM.EN. U4CSE20040
N SREE DIVYA	AM.EN. U4CSE20047
T SAMHITHA	AM.EN. U4CSE20072
V HARSHINI	AM.EN. U4CSE20075

1. Problem Definition:

In the last 30 years if we look it brought many changes in the way how we access music. There is often too much when it comes to enjoying the music. Hence the **core function** mainly depends on **how the modern platforms recommendations matching the user requirements**. Taking this into account we have to know **what models can be used to predict which songs would become popular**. Hence in this project we will find that models can be used to predict the music popularity.

2. Data sets:

→ spotify-2000: (*worked by T. Samhitha and Harshini*)

Spotify-2000.csv.xls

This dataset contains audio statistics of the top 2000 tracks on Spotify. The data contains about 15 columns each describing the track and it's qualities. Songs released from 1956 to 2019 are included from some notable and famous artists like *Queen*, *The Beatles*, *Guns N' Roses*, etc. This is a very fun dataset to explore and find out unique links which land songs in the Top 2000s.

This data contains audio features like Danceability, BPM, Liveness, Valence(Positivity) and many more.
Each feature's description has been given in detail below.

Content

- Index: ID
- Title: Name of the Track
- Artist: Name of the Artist

- Top Genre: Genre of the track
- Year: Release Year of the track
- Beats per Minute(BPM): The tempo of the song
- Energy: The energy of a song - the higher the value, the more energtic. song
- Danceability: The higher the value, the easier it is to dance to this song.

- Loudness: The higher the value, the louder the song.
- Valence: The higher the value, the more positive mood for the song.
- Length: The duration of the song.
- Acoustic: The higher the value the more acoustic the song is.
- Speechiness: The higher the value the more spoken words the song contains
- Popularity: The higher the value the more popular the song is.

→ top 50: (*worked by M. Samhita*)

top50popular.csv.xls

The top 50 most listened songs in the world by Spotify. This dataset has several variables about the songs. There are 50 songs and 13 variables to be explored. This dataset contains audio statistics of the top 50 tracks on Spotify. Contains popular songs like senorita, shape of you etc…

Content

- 50 songs
- 13 variables

→ Songs:

song_data.csv, song_info.csv

Consists of 2 csv files namely Songs_data.csv and songs_info.csv. Dataset contains 19.000 songs and has 15 features like duration ms, key, audio mode, acoustic Ness, danceability, energy and so on.

Content

- duration_ms: The duration of the track in milliseconds.
- key: The estimated 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.
- audio_mode: 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.
- time_signature: An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).
- acousticness: A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
- danceability: Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
- energy: Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale.

- instrumentalness: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
- loudness: 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 typical range between -60 and 0 db.
- 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.
- audio_valence: A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).
- 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.
- song_popularity: Song ratings of spotify audience.
- liveness: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live.

3. PREPARE DATA

→ Pre-processing

1. For the first data set(spotify-2000):

We first checked for null values and found that there were none.

Since duplicates will not aid the model, they will be dropped. We will also check for any null values that may effect the one-hot encoding process later on.

```
3]: df.isnull().sum()
    #no nulls, allgood
```

```
3]: Index                     0
    Title                     0
    Artist                    0
    Top Genre                 0
    Year                      0
    Beats Per Minute (BPM)    0
    Energy                    0
    Danceability              0
    Loudness (dB)             0
    Liveness                  0
    Valence                   0
    Length (Duration)         0
    Acousticness              0
    Speechiness               0
    Popularity                0
    dtype: int64
```

```
4]: print(len(df.index))
    1994
```

We found that in the duration column there were some non-numerical values which was rectified

```
In [69]: df[df["Length (Duration)"].str.contains(",")==True]
         #There were some non-numeric values contained in the duration column, that should have been measured in milliseconds
```

Out[69]:

	Index	Title	Artist	Top Genre	Year	Beats Per Minute (BPM)	Energy	Danceability	Loudness (dB)	Liveness	Valence	Length (Duration)	Acousticness	Speechiness	Popularity
842	843	Echoes	Pink Floyd	album rock	1971	134	32	28	-17	11	14	1,412	37	4	58
904	905	Close to the Edge (i. The Solid Time of Change...	Yes	album rock	1972	81	60	22	-11	41	25	1,121	27	6	47
951	952	Autobahn - 2009 Remaster	Kraftwerk	dance rock	1974	82	50	33	-16	13	11	1,367	11	4	48
1982	1983	Get Ready	Rare Earth	blues rock	1969	127	87	41	-6	83	65	1,292	0	4	45

```
In [70]: df["Length (Duration)"] = df["Length (Duration)"].replace(["1,412","1,121","1,367","1,292"],['1412','1121','1367','1292'])
         #These were corrected manually

In [71]: df["Length (Duration)"] = df["Length (Duration)"].astype(np.int64)
         #Casting all values to integers
```

We dropped all the values that will not be associated with the classifier and then we rechecked for null and duplicate values and found that there were none

```
As the purpose of this exercise is to find out which ML-models best classify the popularity of a given song, we need to only
include columns in the dataframe that are actually useful for classification. Without strong evidence but intuition
withstanding, song titles were dropped from the classifier.
```

```
In [ ]: #Remove values which we will not be associating in the classifier

In [74]: df = df.drop(labels=['Index', 'Title'], axis=1)


         df.shape
Out[74]: (1994, 13)

In [75]: df.duplicated().sum()
         #No duplicates
Out[75]: 0

In [76]: df.isnull().sum().sum()
         #No null values
Out[76]: 0
```

2. Second data set(top50):

We first checked and dropped all null values. Then rechecked for missing or duplicate values and if any column had a different data type. We found nothing.

```
In [171]: data = data.drop(['Unnamed: 0', 'Track.Name'], axis=1)

In [173]: data['Popularity'] = pd.qcut(data['Popularity'], q=2, labels=[0, 1])

In [174]: data.shape
Out[174]: (50, 12)
```

3. Third data set:(song_data, song_info):

We found no null, duplicate or missing values in the data set; we also did not find any column with a value of different data type.

In [81]:
```
song_data.columns[song_data.isnull().any()]
```
Out[81]:
```
Index([], dtype='object')
```

In [82]:
```
song_data.isnull().sum()
```
Out[82]:
```
song_name           0
song_popularity     0
song_duration_ms    0
acousticness        0
danceability        0
energy              0
instrumentalness    0
key                 0
liveness            0
loudness            0
audio_mode          0
speechiness         0
tempo               0
time_signature      0
audio_valence       0
dtype: int64
```

In [85]:
```
song_data["popularity"]= [ 1 if i>=66.5 else 0 for i in song_data.song_popularity ]
song_data["popularity"].value_counts()
```
Out[85]:
```
0    13386
1     5449
Name: popularity, dtype: int64
```

- Checked popularity rating of songs that have been popular in the last 10 years in Spotify and took the mean value of them (66.5) . According to this value, the songs has above this rating could remain on the top lists for a long time. If song_popularity is higher than 66.5 (this is about 30% percent of data) we labeled it "1" and if is not we labeled it "0". So we have "1" for the popular songs and "0" for the unpopular ones.

In [86]:
```
#popular songs' data
a=song_data[song_data["popularity"]==1]
a.describe()
```
Out[86]:

	song_popularity	song_duration_ms	acousticness	danceability	energy	instrumentalness	key	liveness	loudness	audio_mode	spee
count	5449.000000	5449.000000	5449.000000	5449.000000	5449.000000	5449.000000	5449.00000	5449.000000	5449.000000	5449.000000	5449
mean	76.992292	218539.555515	0.210354	0.659758	0.658601	0.022390	5.11782	0.174400	-6.624852	0.618829	0
std	8.068717	48620.048311	0.246079	0.147652	0.187495	0.115572	3.65752	0.137557	3.139341	0.485719	0
min	67.000000	67000.000000	0.000009	0.072200	0.002890	0.000000	0.00000	0.021500	-34.255000	0.000000	0
25%	71.000000	190185.000000	0.026300	0.562000	0.541000	0.000000	1.00000	0.092000	-7.906000	0.000000	0
50%	75.000000	212429.000000	0.106000	0.668000	0.680000	0.000000	5.00000	0.121000	-5.985000	1.000000	0
75%	82.000000	240533.000000	0.300000	0.765000	0.802000	0.000118	8.00000	0.203000	-4.626000	1.000000	0
max	100.000000	547733.000000	0.996000	0.978000	0.997000	0.968000	11.00000	0.978000	-0.739000	1.000000	0

We checked if any outliers are present and then removed if so.

```
In [93]: from collections import Counter
def detect_outliers(df,features):
    outlier_indices = []

    for c in features:
        # 1st quartile
        Q1 = np.percentile(df[c],25)
        # 3rd quartile
        Q3 = np.percentile(df[c],75)
        # IQR
        IQR = Q3 - Q1
        # Outlier step
        outlier_step = IQR * 1.5
        # detect outlier and their indeces
        outlier_list_col = df[(df[c] < Q1 - outlier_step) | (df[c] > Q3 + outlier_step)].index #filtre
        # store indeces
        outlier_indices.extend(outlier_list_col) #The extend() extends the list by adding all items of a list (passed as an argum

    outlier_indices = Counter(outlier_indices)
    multiple_outliers = list(i for i, v in outlier_indices.items() if v > 2)

    return multiple_outliers
```

```
In [26]: detect_outliers(song_data,["song_popularity","song_duration_ms","danceability","energy","instrumentalness","liveness","loudness",
```

Out[26]:

	song_name	song_popularity	song_duration_ms	acousticness	danceability	energy	instrumentalness	key	liveness	loudness	audio_mode	speechines
232	La Maza	58	351400.0	0.6520	0.555	0.331	0.000012	9	0.235	-17.718	0.0	0.270
253	Whole Lotta Love	77	333893.0	0.0484	0.412	0.902	0.131000	9	0.405	-11.600	1.0	0.405

```
In [94]: # drop outliers
song_data = song_data.drop(detect_outliers(song_data,["song_popularity","song_duration_ms","danceability","energy","instrumentalı
```

```
In [95]: song_data[song_data["audio_mode"].isnull()]
```

Out[95]:

song_name	song_popularity	song_duration_ms	acousticness	danceability	energy	instrumentalness	key	liveness	loudness	audio_mode	speechiness	te

→ <u>Summarization:</u>

1. For the first data set(spotify-2000):

<u>Dimensions of the data set</u>

```
In [196]: df.head(1000)
```

Out[196]:

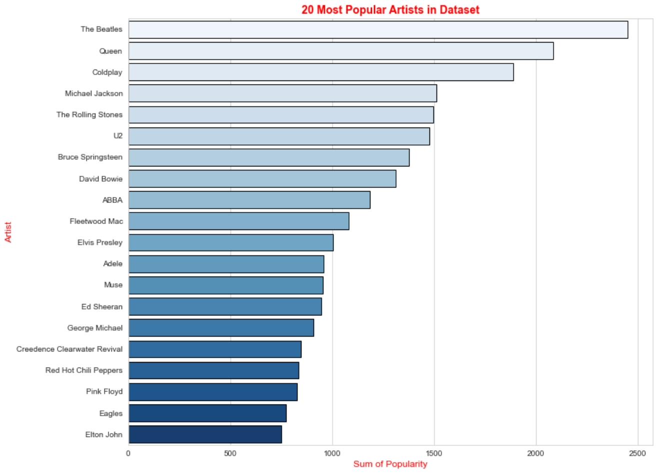
	Index	Title	Artist	Top Genre	Year	Beats Per Minute (BPM)	Energy	Danceability	Loudness (dB)	Liveness	Valence	Length (Duration)	Acousticness	Speechiness	Popı
0	1	Sunrise	Norah Jones	adult standards	2004	157	30	53	-14	11	68	201	94	3	
1	2	Black Night	Deep Purple	album rock	2000	135	79	50	-11	17	81	207	17	7	
2	3	Clint Eastwood	Gorillaz	alternative hip hop	2001	168	69	66	-9	7	52	341	2	17	
3	4	The Pretender	Foo Fighters	alternative metal	2007	173	96	43	-4	3	37	269	0	4	
4	5	Waitin' On A Sunny Day	Bruce Springsteen	classic rock	2002	106	82	58	-5	10	87	256	1	3	

```
In [197]: df.shape
```

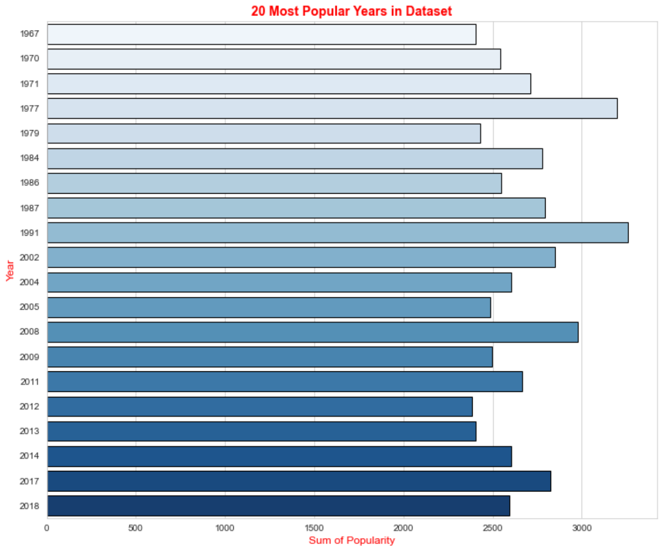
Out[197]: (1994, 15)

Statistical summary of all attributes:

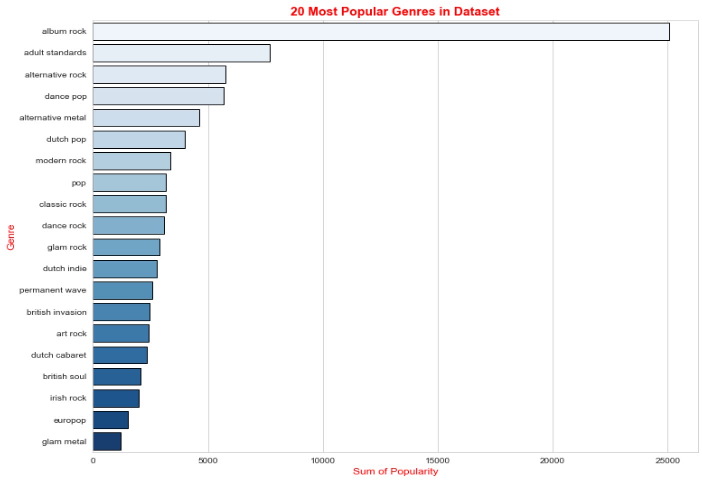
```
In [26]: fig, ax = plt.subplots(figsize = (12, 10))
         lead_artists = df.groupby('Artist')['Popularity'].sum().sort_values(ascending=False).head(20)
         ax = sns.barplot(x=lead_artists.values, y=lead_artists.index, palette="Blues", orient="h", edgecolor='black', ax=ax)
         ax.set_xlabel('Sum of Popularity', c='r', fontsize=12)
         ax.set_ylabel('Artist', c='r', fontsize=12)
         ax.set_title('20 Most Popular Artists in Dataset', c='r', fontsize=14, weight = 'bold')
         plt.show()
```




```
In [31]: fig, ax = plt.subplots(figsize = (12, 10))
         lead_artists = df.groupby('Year')['Popularity'].sum().sort_values(ascending=False).head(20)
         ax = sns.barplot(x=lead_artists.values, y=lead_artists.index, palette="Blues", orient="h", edgecolor='black', ax=ax)
         ax.set_xlabel('Sum of Popularity', c='r', fontsize=12)
         ax.set_ylabel('Year', c='r', fontsize=12)
         ax.set_title('20 Most Popular Years in Dataset', c='r', fontsize=14, weight = 'bold')
         plt.show()
```



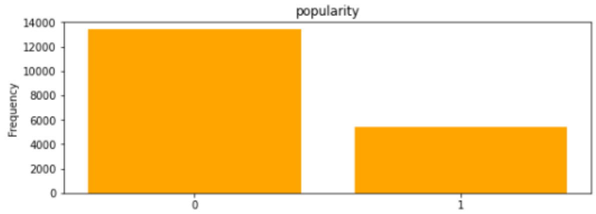

```
]: fig, ax = plt.subplots(figsize = (12, 10))
   lead_artists = df.groupby('Top Genre')['Popularity'].sum().sort_values(ascending=False).head(20)
   ax = sns.barplot(x=lead_artists.values, y=lead_artists.index, palette="Blues", orient="h", edgecolor='black', ax=ax)
   ax.set_xlabel('Sum of Popularity', c='r', fontsize=12)
   ax.set_ylabel('Genre', c='r', fontsize=12)
   ax.set_title('20 Most Popular Genres in Dataset', c='r', fontsize=14, weight = 'bold')
   plt.show()
```



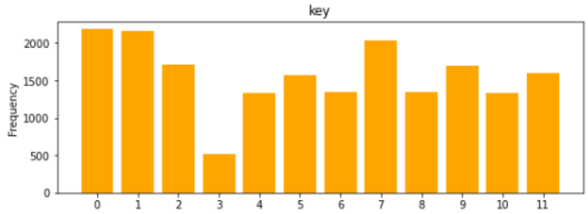
Breakdown of the data by the class variable:

```
In [89]: def bar_plot(variable):

             var=song_data[variable]
             var_value= var.value_counts()

             #visualize
             plt.figure(figsize=(9,3))
             plt.bar(var_value.index,var_value,color="orange")
             plt.xticks(var_value.index,var_value.index.values)
             plt.ylabel("Frequency")
             plt.title(variable)
             plt.show()
             print("{}:\n{}".format(variable,var_value))
```

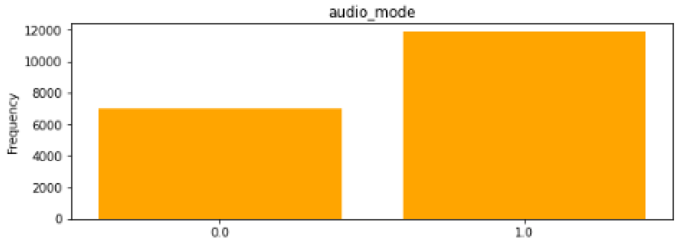
```
In [20]: category1 = ["popularity","key","audio_mode","time_signature"]
         for c in category1:
             bar_plot(c)
```



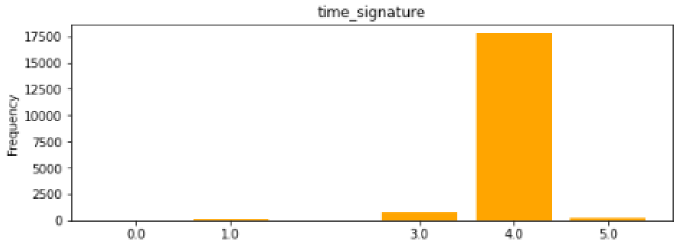

```
popularity:
0    13386
1     5449
Name: popularity, dtype: int64
```




```
key:
0     2182
1     2164
7     2032
2     1715
9     1698
11    1600
5     1574
6     1351
8     1349
10    1331
4     1327
3      512
Name: key, dtype: int64
```




```
audio_mode:
1.0    11831
0.0     7004
Name: audio_mode, dtype: int64
```




```
time_signature:
4.0    17754
3.0      772
5.0      233
1.0       73
0.0        3
Name: time_signature, dtype: int64
```

2. **Second data set(top50):**

Dimensions of the data set

In [207]:
```
data.head()
```

Out[207]:

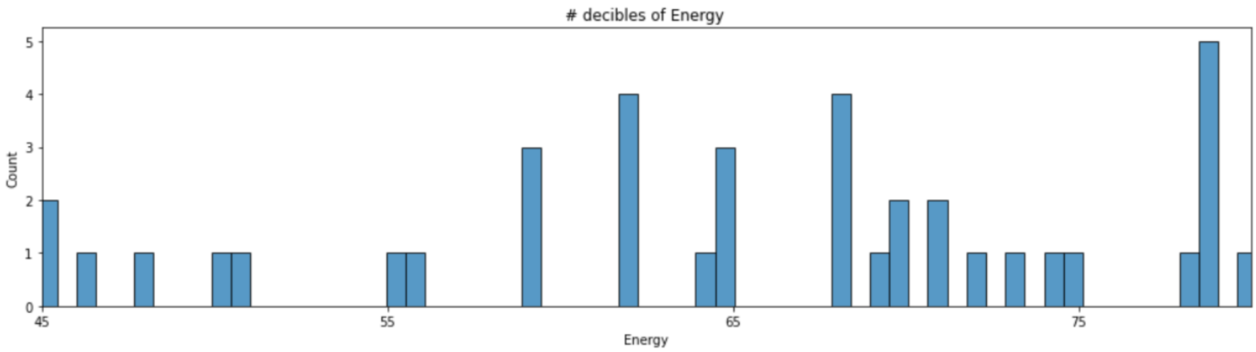
	Unnamed: 0	Track.Name	Artist.Name	Genre	Beats.Per.Minute	Energy	Danceability	Loudness..dB..	Liveness	Valence.	Length.	Acousticness..	Speechin
0	1	Señorita	Shawn Mendes	canadian pop	117	55	76	-6	8	75	191	4	
1	2	China	Anuel AA	reggaeton flow	105	81	79	-4	8	61	302	8	
2	3	boyfriend (with Social House)	Ariana Grande	dance pop	190	80	40	-4	16	70	186	12	
3	4	Beautiful People (feat. Khalid)	Ed Sheeran	pop	93	65	64	-8	8	55	198	12	
4	5	Goodbyes (Feat. Young Thug)	Post Malone	dfw rap	150	65	58	-4	11	18	175	45	

In [206]:
```
data.shape
```

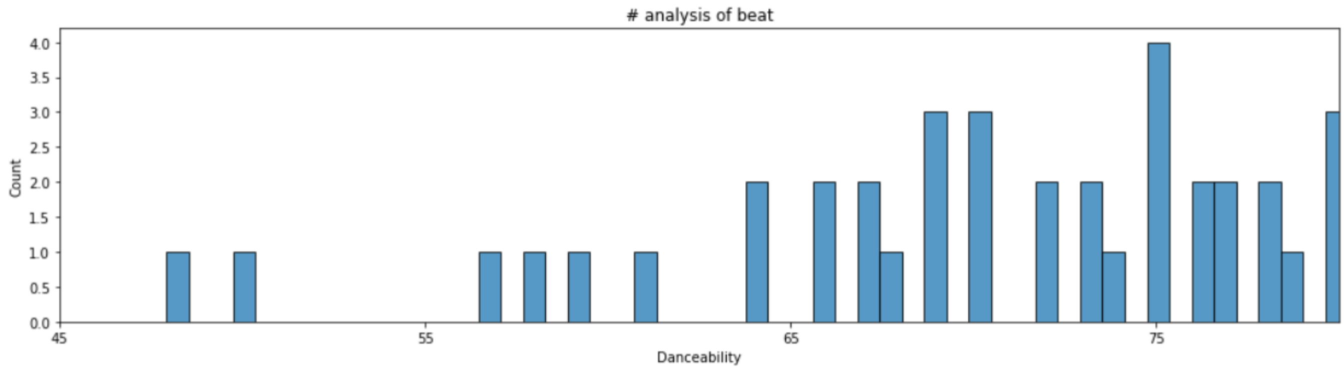
Out[206]: (50, 14)

Statistical summary of all attributes:

In [240]:
```
fig, ax = plt.subplots(figsize=(17, 4))
ax = sns.histplot(data['Energy'], bins = 100, kde = False)
ax.set_xlim(45,80)
ax.set_xticks(range(45, 80, 10))
ax.set_title('# decibles of Energy ')
plt.show()
```



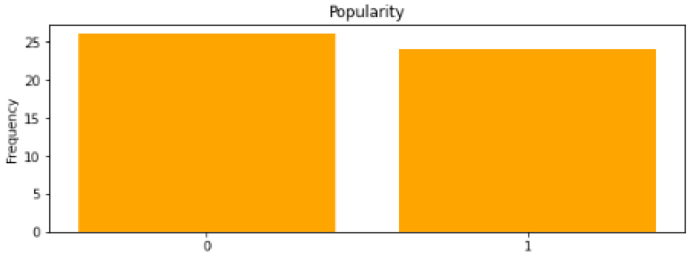
]:
```
fig, ax = plt.subplots(figsize=(17, 4))
ax = sns.histplot(data['Danceability'], bins = 100, kde = False)
ax.set_xlim(45,80)
ax.set_xticks(range(45, 80, 10))
ax.set_title('# analysis of beat')
plt.show()
```



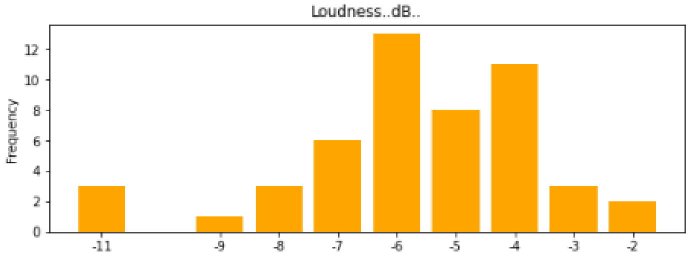
Breakdown of the data by the class variable:

In [219]:
```
def bar_plot(variable):

    var=data[variable]
    var_value= var.value_counts()

    #visualize
    plt.figure(figsize=(9,3))
    plt.bar(var_value.index,var_value,color="orange")
    plt.xticks(var_value.index,var_value.index.values)
    plt.ylabel("Frequency")
    plt.title(variable)
    plt.show()
    print("{}:\n{}".format(variable,var_value))
```

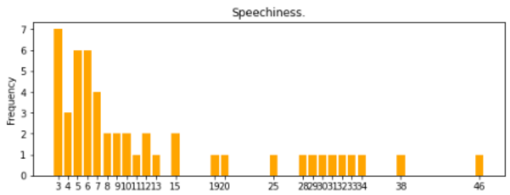
In [220]:
```
category1 = ["Popularity","Loudness..dB..","Speechiness."]
for c in category1:
    bar_plot(c)
```




```
Popularity:
0    26
1    24
Name: Popularity, dtype: int64
```




```
Loudness..dB..:
-6     13
-4     11
-5      8
-7      6
-8      3
-11     3
-3      3
-2      2
-9      1
Name: Loudness..dB.., dtype: int64
```




```
Speechiness.:
3     7
5     6
6     6
7     4
4     3
8     2
9     2
15    2
10    2
12    2
38    1
31    1
20    1
29    1
32    1
19    1
11    1
30    1
46    1
25    1
33    1
28    1
34    1
13    1
Name: Speechiness., dtype: int64
```

3. Third data set:(song_data, song_info):

Dimensions of the data set:

```
song_data.head()
```

	song_duration_ms	acousticness	danceability	energy	instrumentalness	liveness	loudness	speechiness	tempo	audio_valence	...	key_9	key_10	key_1
0	262333.0	0.005520	0.496	0.682	0.000029	0.0589	-4.095	0.0294	167.060	0.474	...	0	0	(
1	216933.0	0.010300	0.542	0.853	0.000000	0.1080	-6.407	0.0498	105.256	0.370	...	0	0	(
2	231733.0	0.008170	0.737	0.463	0.447000	0.2550	-7.828	0.0792	123.881	0.324	...	0	0	(
3	216933.0	0.026400	0.451	0.970	0.003550	0.1020	-4.938	0.1070	122.444	0.198	...	0	0	(
4	223826.0	0.000954	0.447	0.766	0.000000	0.1130	-5.065	0.0313	172.011	0.574	...	0	1	(

5 rows × 30 columns

```
song_data.shape
```

```
(18510, 30)
```

In [213]:
```
song_info.head()
```

Out[213]:

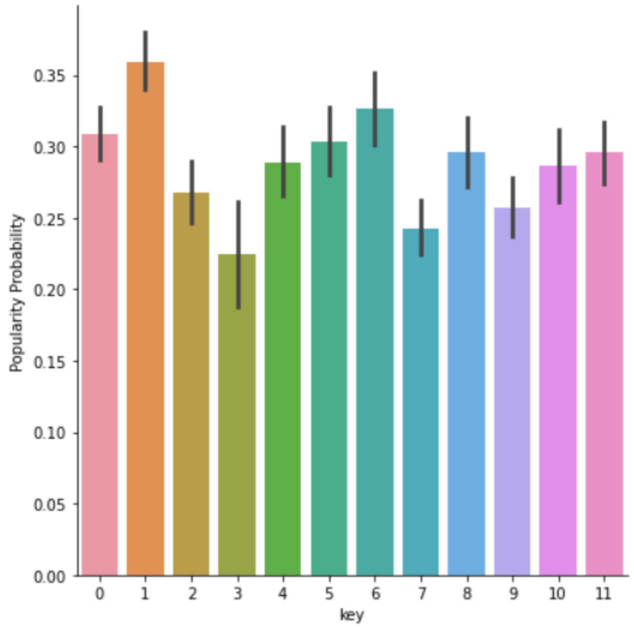
	song_name	artist_name	album_names	playlist
0	Boulevard of Broken Dreams	Green Day	Greatest Hits: God's Favorite Band	00s Rock Anthems
1	In The End	Linkin Park	Hybrid Theory	00s Rock Anthems
2	Seven Nation Army	The White Stripes	Elephant	00s Rock Anthems
3	By The Way	Red Hot Chili Peppers	By The Way (Deluxe Version)	00s Rock Anthems
4	How You Remind Me	Nickelback	Silver Side Up	00s Rock Anthems

In [214]:
```
song_info.shape
```

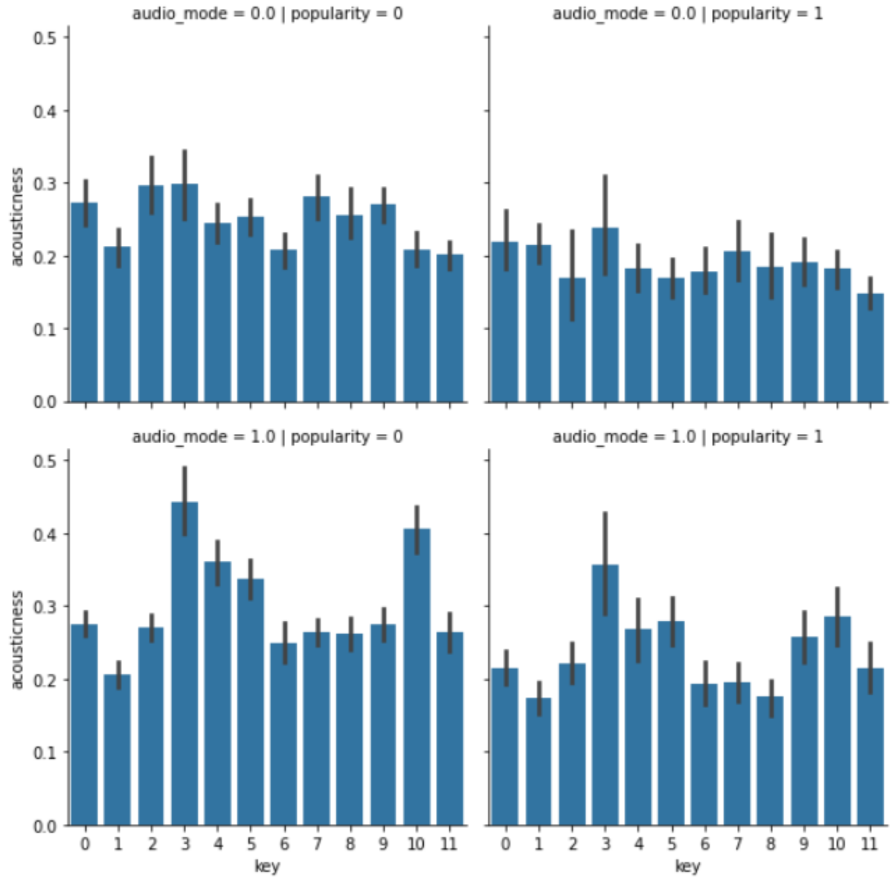
Out[214]:
```
(18835, 4)
```

Statistical summary of all attributes:

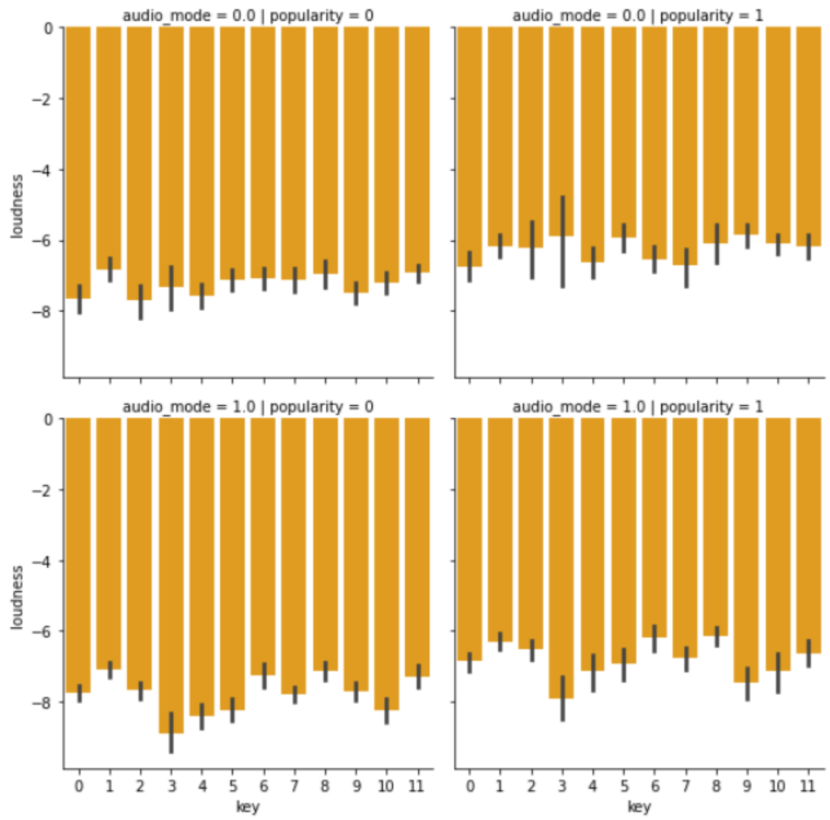
In [96]:
```
g = sns.factorplot(x = "key", y = "popularity", data = song_data, kind = "bar", size = 6)
g.set_ylabels("Popularity Probability")
plt.show()
```



```
g = sns.FacetGrid(song_data, row = "audio_mode", col = "popularity", size = 4)
g.map(sns.barplot, "key", "acousticness")
g.add_legend()
plt.show()
```



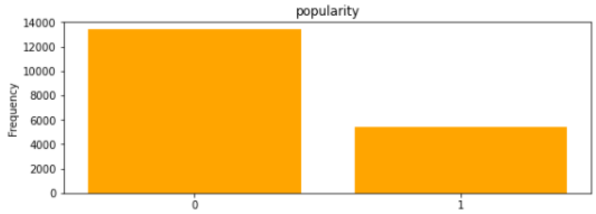

```
g = sns.FacetGrid(song_data, row = "audio_mode", col = "popularity", size = 4)
g.map(sns.barplot, "key", "loudness",color="orange")
g.add_legend()
plt.show()
```



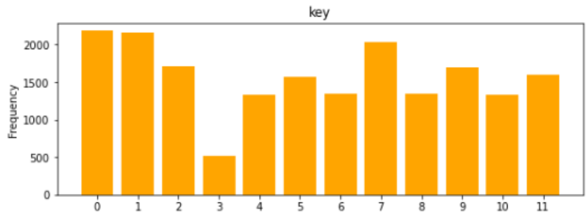
Breakdown of the data by the class variable:

```
In [89]: def bar_plot(variable):

             var=song_data[variable]
             var_value= var.value_counts()

             #visualize
             plt.figure(figsize=(9,3))
             plt.bar(var_value.index,var_value,color="orange")
             plt.xticks(var_value.index,var_value.index.values)
             plt.ylabel("Frequency")
             plt.title(variable)
             plt.show()
             print("{}:\n{}".format(variable,var_value))
```

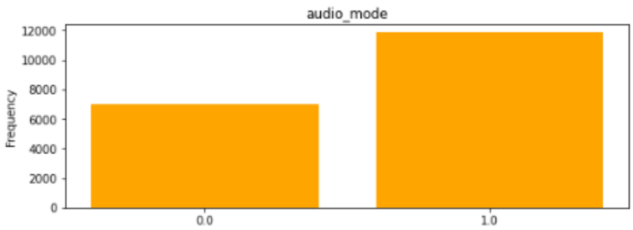
```
In [20]: category1 = ["popularity","key","audio_mode","time_signature"]
         for c in category1:
             bar_plot(c)
```



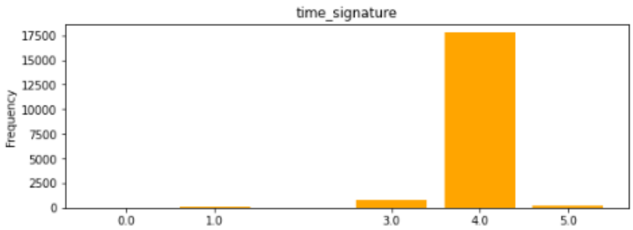

```
popularity:
0    13386
1     5449
Name: popularity, dtype: int64
```




```
key:
0     2182
1     2164
7     2032
2     1715
9     1698
11    1600
5     1574
6     1351
8     1349
10    1331
4     1327
3      512
Name: key, dtype: int64
```




```
audio_mode:
1.0    11831
0.0     7004
Name: audio_mode, dtype: int64
```



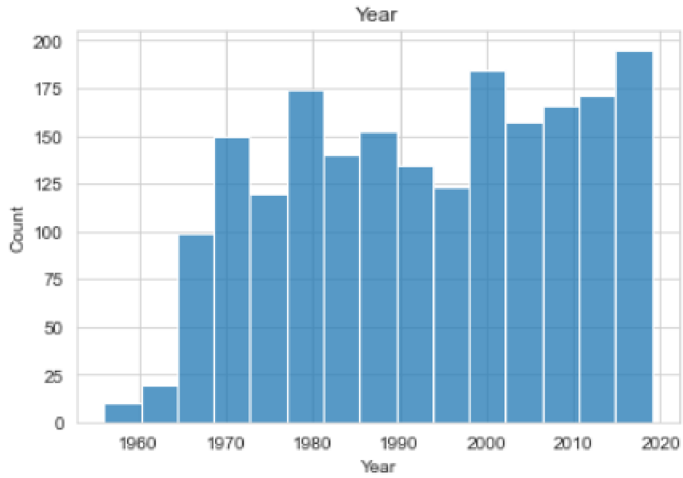

```
time_signature:
4.0    17754
3.0      772
5.0      233
1.0       73
0.0        3
Name: time_signature, dtype: int64
```

→ Data Visualization:

1. For the first data set(spotify-2000):

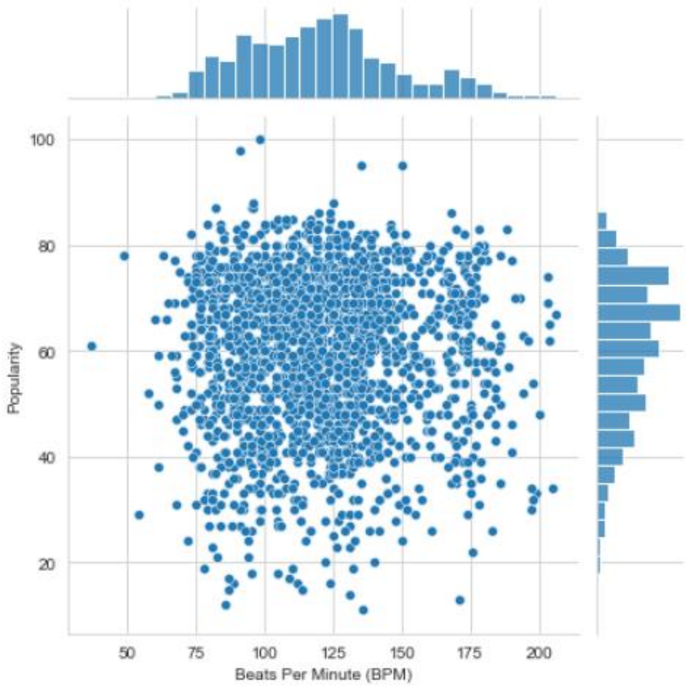
```
In [83]: sns.histplot(df['Year']).set_title('Year')
```

```
Out[83]: Text(0.5, 1.0, 'Year')
```



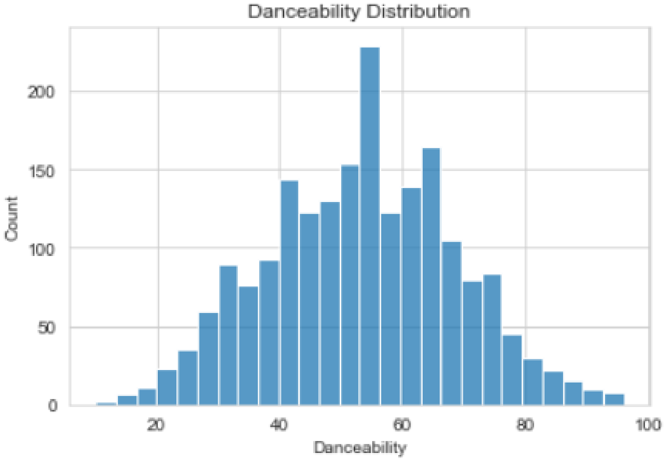

```
In [88]: #Nothing to see here
         sns.jointplot(x = 'Beats Per Minute (BPM)', y = 'Popularity', data = df)
```

```
Out[88]: <seaborn.axisgrid.JointGrid at 0x1fef3df57c0>
```

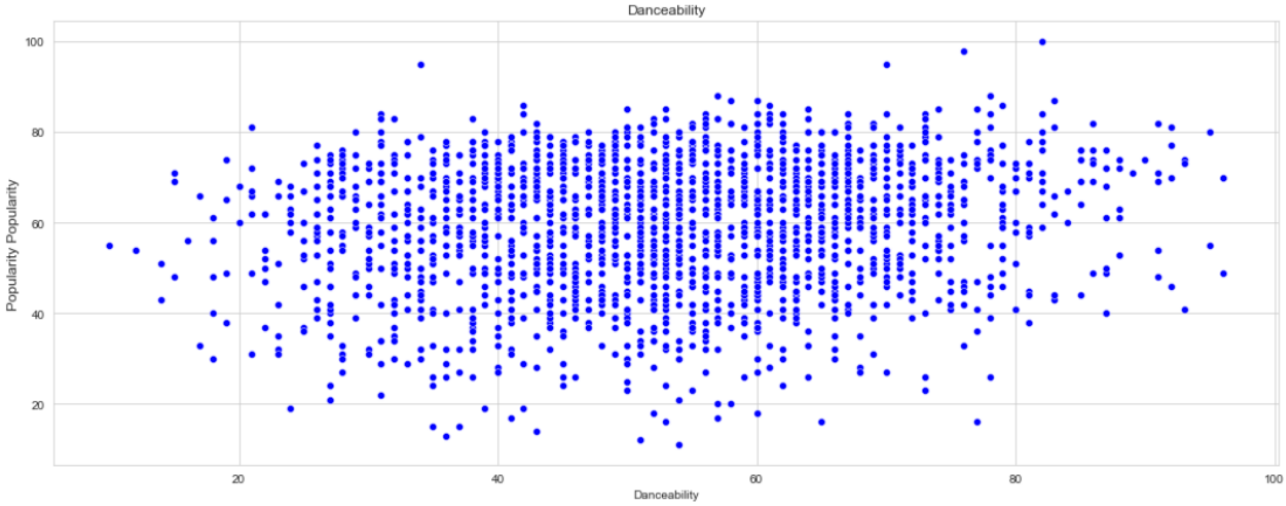


```
In [91]: sns.histplot(df['Danceability']).set_title('Danceability Distribution')
```

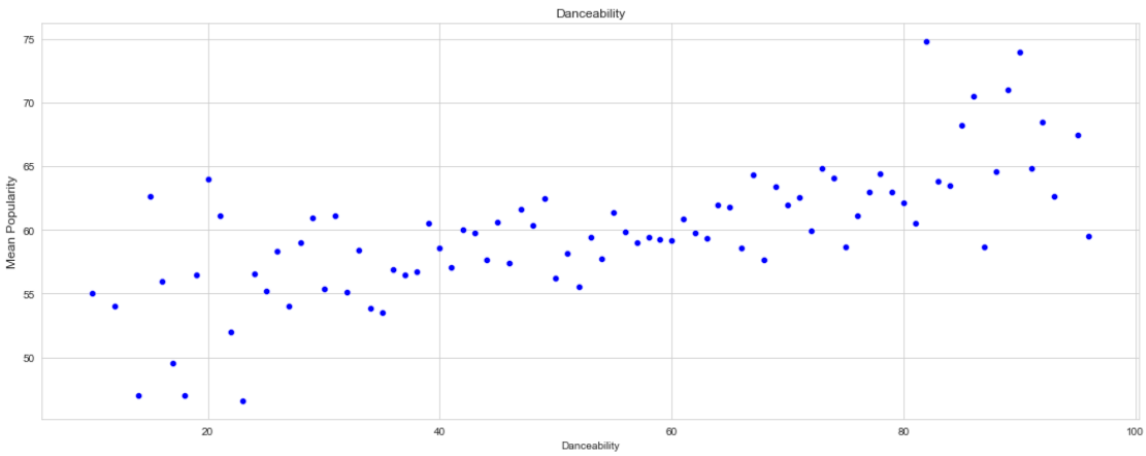
```
Out[91]: Text(0.5, 1.0, 'Danceability Distribution')
```



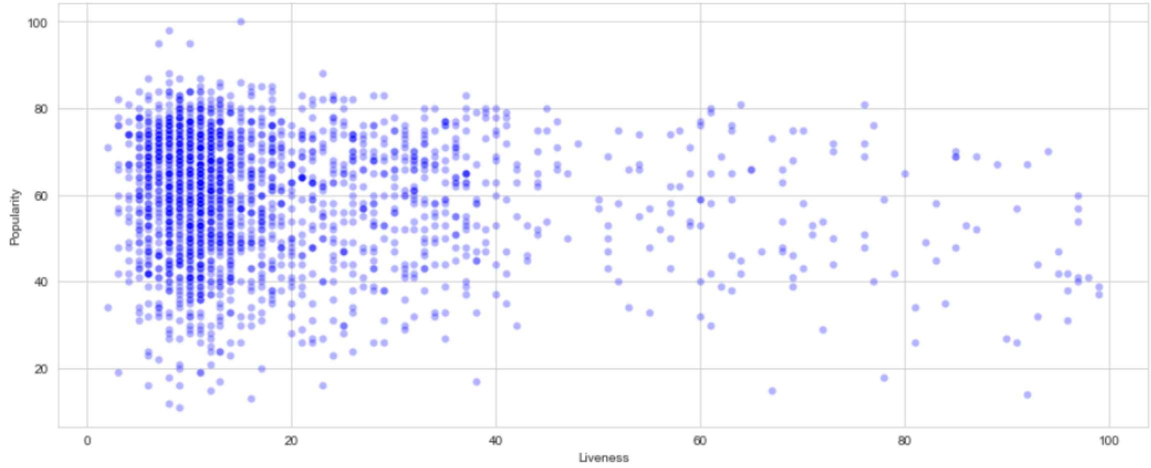

```
In [93]: #One of the better predictors
         fig, ax = plt.subplots(1, figsize=(15, 6), sharey=True, sharex = True)
         ax = sns.scatterplot(x='Danceability', y='Popularity', data=df, color='blue', ax=ax)
         ax.set_title('Danceability')
         ax.set_ylabel('Popularity Popularity', fontsize=12)
         plt.tight_layout()
         plt.show()
```




```
In [94]: fig, ax = plt.subplots(1, figsize=(15, 6), sharey=True, sharex = True)
         ax_data =  df.groupby('Danceability')['Popularity'].mean().to_frame().reset_index()
         ax = sns.scatterplot(x='Danceability', y='Popularity', data=ax_data, color='blue', ax=ax)
         ax.set_title('Danceability')
         ax.set_ylabel('Mean Popularity', fontsize=12)
         plt.tight_layout()
         plt.show()
```

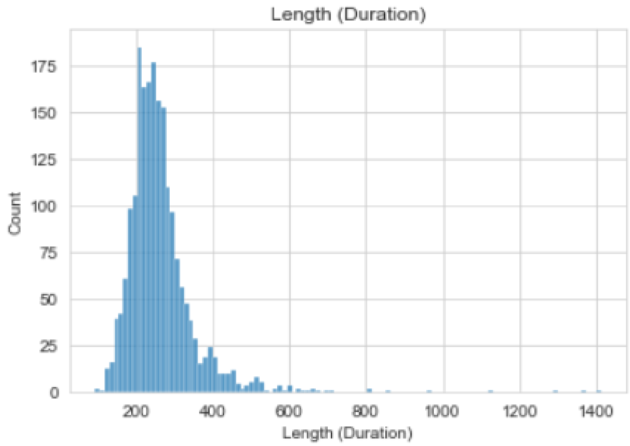


In [100]:
```
fig, ax = plt.subplots(figsize = (15, 6))
sns.scatterplot(x='Liveness', y='Popularity', data=df, color='blue', alpha=0.3)
plt.show()
```



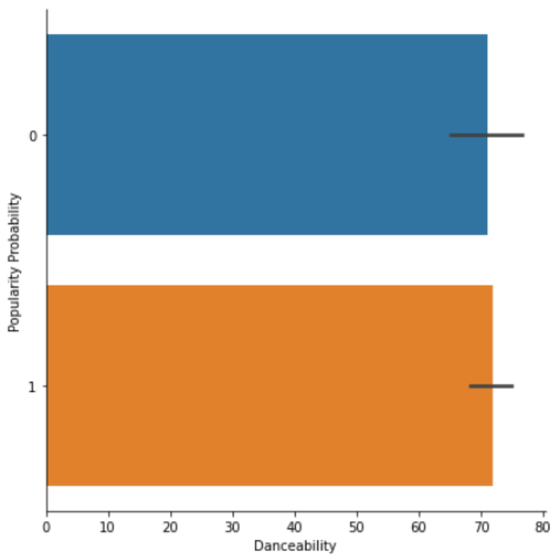
In [103]:
```
sns.histplot(df['Length (Duration)']).set_title('Length (Duration)')
```

Out[103]: Text(0.5, 1.0, 'Length (Duration)')

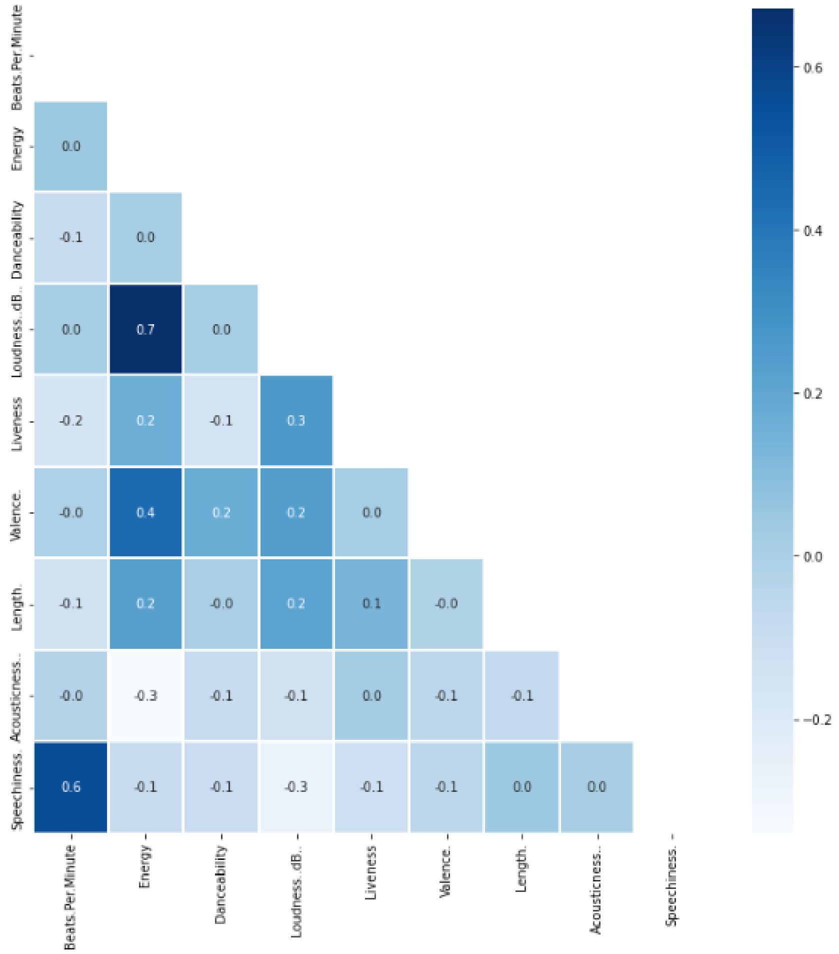


2. Second data set(top50):

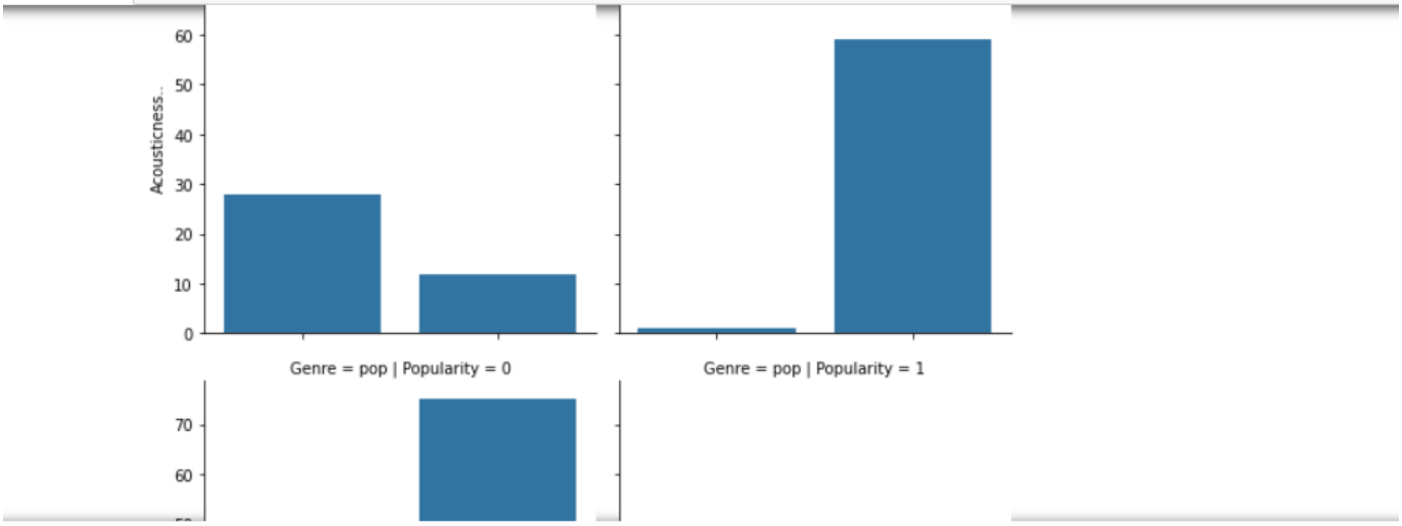
In [217]:
```
g = sns.catplot(x = "Danceability", y = "Popularity", data = data, kind = "bar", height = 6)
g.set_ylabels("Popularity Probability")
plt.show()
```



```
In [218]: f,ax = plt.subplots(figsize=(12, 12))
mask = np.zeros_like(data.corr())
mask[np.triu_indices_from(mask)] = True
sns.heatmap(data.corr(), annot=True, linewidths=0.4,linecolor="white", fmt= '.1f',ax=ax,cmap="Blues",mask=mask)
plt.show()
```

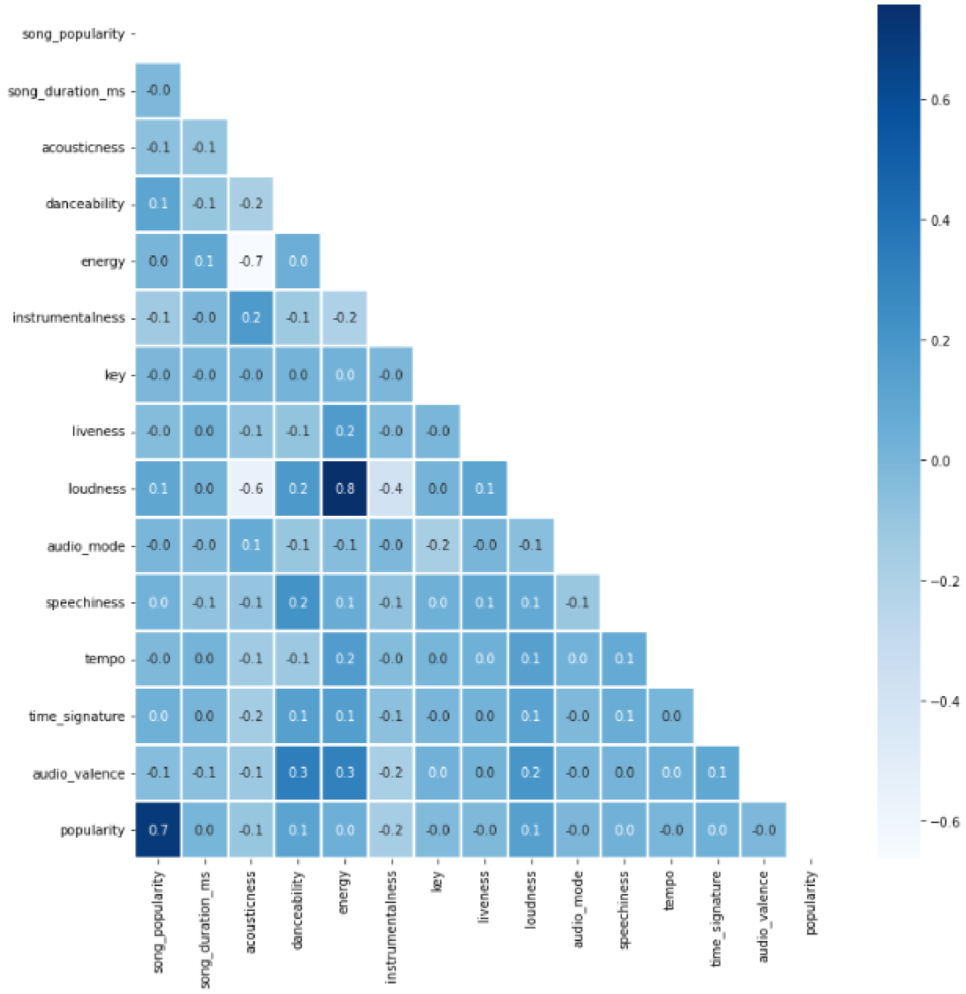



```
In [179]: g = sns.FacetGrid(data, row = "Genre", col = "Popularity", height = 4)
g.map(sns.barplot, "Length.", "Acousticness..")
g.add_legend()
plt.show()
```

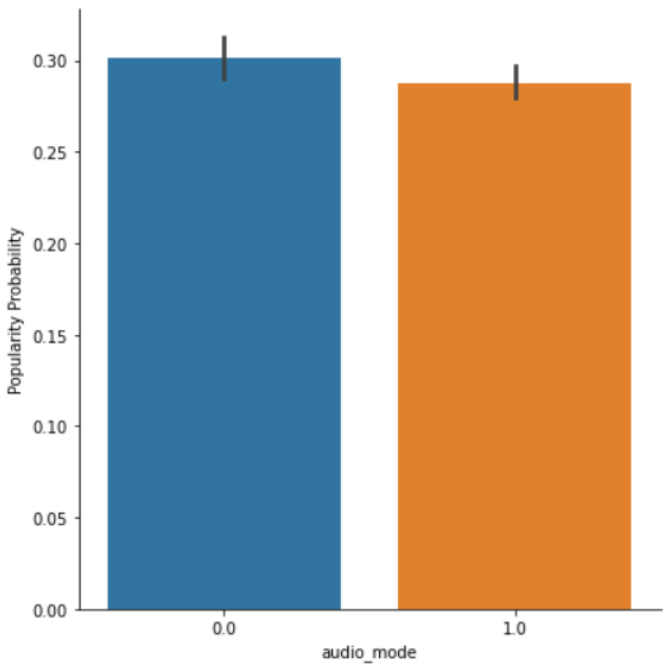


3. Third data set:(song_data, song_info):

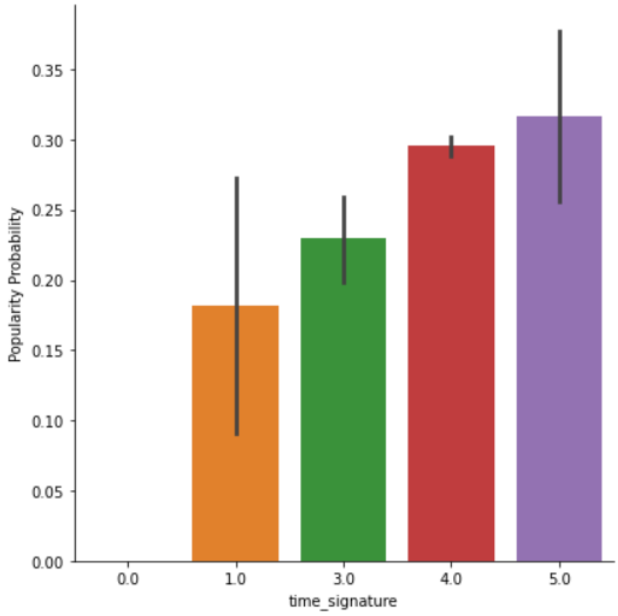
In [88]:
```
f,ax = plt.subplots(figsize=(12, 12))
mask = np.zeros_like(song_data.corr())
mask[np.triu_indices_from(mask)] = True
sns.heatmap(song_data.corr(), annot=True, linewidths=0.4,linecolor="white", fmt= '.1f',ax=ax,cmap="Blues",mask=mask)
plt.show()
```



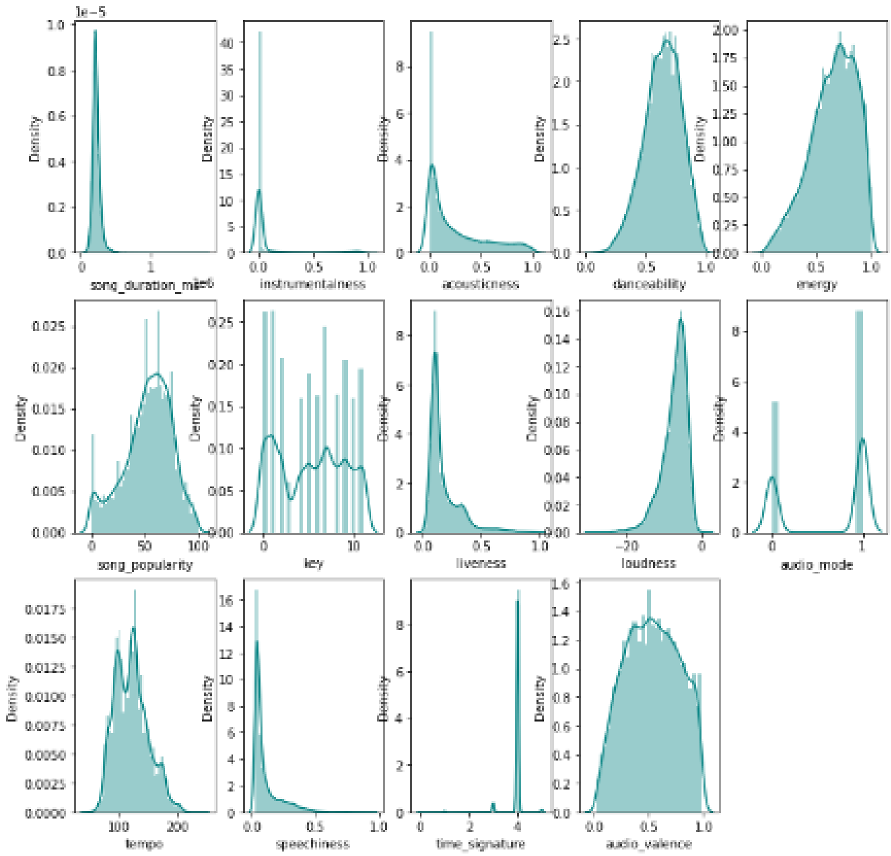
In [21]:
```
g = sns.factorplot(x = "audio_mode", y = "popularity", data = song_data, kind = "bar", size = 6)
g.set_ylabels("Popularity Probability")
plt.show()
```



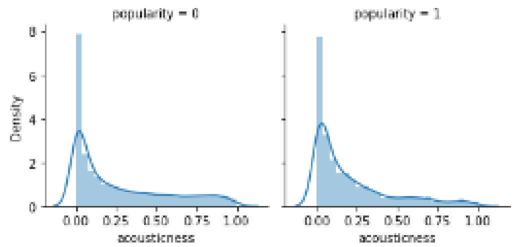
In [97]:
```
g = sns.factorplot(x = "time_signature", y = "popularity", data = song_data, kind = "bar", size = 6)
g.set_ylabels("Popularity Probability")
plt.show()
```



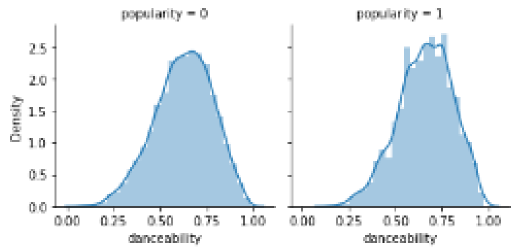
In [103]:
```
f, axes = plt.subplots(3, 5, figsize=(12, 12))
sns.distplot( song_data["song_duration_ms"] , color="teal", ax=axes[0, 0])
sns.distplot( song_data["instrumentalness"] , color="teal", ax=axes[0, 1])
sns.distplot( song_data["acousticness"] , color="teal", ax=axes[0, 2])
sns.distplot( song_data["danceability"] , color="teal", ax=axes[0, 3])
sns.distplot( song_data["energy"] , color="teal", ax=axes[0, 4])
sns.distplot( song_data["song_popularity"] , color="teal", ax=axes[1, 0])
sns.distplot( song_data["key"] , color="teal", ax=axes[1, 1])
sns.distplot( song_data["liveness"] , color="teal", ax=axes[1, 2])
sns.distplot( song_data["loudness"] , color="teal", ax=axes[1, 3])
sns.distplot( song_data["audio_mode"] , color="teal", ax=axes[1, 4])
sns.distplot( song_data["tempo"] , color="teal", ax=axes[2, 0])
sns.distplot( song_data["speechiness"] , color="teal", ax=axes[2, 1])
sns.distplot( song_data["time_signature"] , color="teal", ax=axes[2, 2])
sns.distplot( song_data["audio_valence"] , color="teal", ax=axes[2, 3])
f.delaxes(axes[2][4])
plt.show()
```



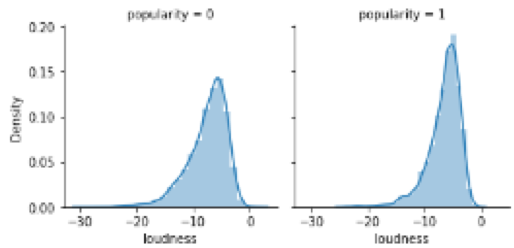
```
In [104]: g = sns.FacetGrid(song_data, col = "popularity")
          g.map(sns.distplot, "acousticness", bins = 25)
          plt.show()
```



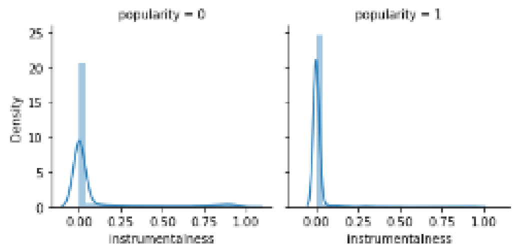

```
In [105]: g = sns.FacetGrid(song_data, col = "popularity")
          g.map(sns.distplot, "danceability", bins = 25)
          plt.show()
```




```
In [106]: g = sns.FacetGrid(song_data, col = "popularity")
          g.map(sns.distplot, "loudness", bins = 25)
          plt.show()
```




```
In [107]: g = sns.FacetGrid(song_data, col = "popularity")
          g.map(sns.distplot, "instrumentalness", bins = 25)
          plt.show()
```



4.PYTHON PACKAGES:

Numpy: NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices. NumPy is a Python library used for working with arrays.

Pandas: The Pandas module mainly works with the tabular data, whereas the NumPy module works with the numerical data. pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Plotly: The plotly Python library is an interactive, open-source plotting library that supports over 40 unique chart types covering a wide range of statistical, financial, geographic, scientific, and 3-dimensional use-cases. Plotly has several advantages over matplotlib. One of the main advantages is that only a few lines of codes are necessary to create aesthetically pleasing, interactive plots. The interactivity also offers a number of advantages over static matplotlib plots: Saves time when initially exploring your dataset.

Matplolib: Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Matplotlib makes easy things easy and hard things possible. Create publication quality plots. Make interactive figures that can zoom, pan, update.

Seaborn: Seaborn is a Python data visualization library based on matplotlib . It provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn is built on top of Python’s core visualization library Matplotlib. It is meant to serve as a complement, and not a replacement. However, Seaborn comes with some very important features. Let us see a few of them here. The features help in −

- Built in themes for styling matplotlib graphics
- Visualizing univariate and bivariate data
- Fitting in and visualizing linear regression models
- Plotting statistical time series data
- Seaborn works well with NumPy and Pandas data structures
- It comes with built in themes for styling Matplotlib graphics

Standard scaler: Python sklearn library offers us with StandardScaler() function to standardize the data values into a standard format. According to the above syntax, we initially create an object of the StandardScaler() function. Further, we use fit_transform() along with the assigned object to transform the data and standardize it.

Train_Test_Split: The train_test_split() method is used to split our data into train and test sets. First, we need to divide our data into features (X) and labels (y). The dataframe gets divided into X_train, X_test, y_train, and y_test. X_train and y_train sets are used for training and fitting the model.

5. LEARNING ALGORITHMS:

1. For the first data set(spotify-2000):

→ Split your dataset into training, validation and testing:

Splitting into Train and Test

```
In [123]: y = df.loc[:, 'Popularity']
          X = df.drop('Popularity', axis=1)
```

```
In [124]: scaler = StandardScaler()

          X = scaler.fit_transform(X)
```

```
In [125]: #random = 369
          X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=369)
```

```
In [126]: print("X_train: ",X_train.shape)
          print("X_test: ",X_test.shape)
          print("y_train: ",y_train.shape)
          print("y_test: ",y_test.shape)

          X_train:  (1395, 741)
          X_test:  (599, 741)
          y_train:  (1395,)
          y_test:  (599,)
```

→ Create models and estimate their accuracy on unseen data using the specified ML algorithms:

1. Logistic Regression:

```
In [133]: logreg = LogisticRegression()
          _ = logreg.fit(X_train, y_train)
          # print('Intercept:', logreg.intercept_)
          # print('Coefficients:\n', logreg.coef_)

          # We can predict the type of new organisms given measurements
          print('\nPredicted type of first five organisms from test split:', logreg.predict(X_test)[:10])
          print('Actual type of first five organisms from test split:', y_test[:10])

          Predicted type of first five organisms from test split: [1 0 1 1 0 0 0 1 0 0]
          Actual type of first five organisms from test split: 853     1
          131     0
          1666    0
          1938    0
          112     1
          1467    0
          480     0
          1555    1
          1027    0
          1329    0
          Name: Popularity, dtype: category
          Categories (2, int64): [0 < 1]
```

```
In [134]: from sklearn.model_selection import GridSearchCV
          import warnings
          warnings.filterwarnings('ignore')

          param_grid = [
              {'C': [1, 10, 100, 1000, 1e4, 1e5, 1e6, 1e7], 'penalty': ['l1', 'l2']}
          ]
          logreg = GridSearchCV(LogisticRegression(), param_grid)
          logreg.fit(X_train, y_train)

          #print(logreg.cv_results_[{'mean_test_score','std_test_score','params'}])
          scoring = logreg.cv_results_
          for mean_score, std, params in zip(scoring['mean_test_score'],scoring['std_test_score'],scoring['params']):
              print("{:0.3f} (+/-{:0.03f}) for {}".format(
                    mean_score, std * 2, params))

          nan (+/-nan) for {'C': 1, 'penalty': 'l1'}
          0.708 (+/-0.030) for {'C': 1, 'penalty': 'l2'}
          nan (+/-nan) for {'C': 10, 'penalty': 'l1'}
          0.705 (+/-0.028) for {'C': 10, 'penalty': 'l2'}
          nan (+/-nan) for {'C': 100, 'penalty': 'l1'}
          0.706 (+/-0.029) for {'C': 100, 'penalty': 'l2'}
          nan (+/-nan) for {'C': 1000, 'penalty': 'l1'}
          0.705 (+/-0.028) for {'C': 1000, 'penalty': 'l2'}
          nan (+/-nan) for {'C': 10000.0, 'penalty': 'l1'}
          0.705 (+/-0.028) for {'C': 10000.0, 'penalty': 'l2'}
          nan (+/-nan) for {'C': 100000.0, 'penalty': 'l1'}
          0.705 (+/-0.028) for {'C': 100000.0, 'penalty': 'l2'}
          nan (+/-nan) for {'C': 1000000.0, 'penalty': 'l1'}
          0.705 (+/-0.028) for {'C': 1000000.0, 'penalty': 'l2'}
          nan (+/-nan) for {'C': 10000000.0, 'penalty': 'l1'}
          0.705 (+/-0.028) for {'C': 10000000.0, 'penalty': 'l2'}
```

In [135]:
```
print('\nBest parameters:', logreg.best_params_)
```

```
Best parameters: {'C': 1, 'penalty': 'l2'}
```

In [136]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, logreg.predict(X_test)))
```

```
              precision    recall  f1-score   support

           0       0.71      0.72      0.72       321
           1       0.67      0.67      0.67       278

    accuracy                           0.69       599
   macro avg       0.69      0.69      0.69       599
weighted avg       0.69      0.69      0.69       599
```

In [137]:
```
logreg_opt = LogisticRegression(penalty='l2',C= 10000000)
_ = logreg_opt.fit(X_train, y_train)
# print('Intercept:', logreg.intercept_)
# print('Coefficients:\n', logreg.coef_)

# We can predict the type of new organisms given measurements
print('\nPredicted type of first five songs from test split:', logreg_opt.predict(X_test)[:10])
print('Actual type of first five songs from test split:', y_test[:10])
```

```
Predicted type of first five songs from test split: [1 0 1 1 1 0 0 1 0 0]
Actual type of first five songs from test split: 853     1
131     0
1666    0
1938    0
112     1
1467    0
480     0
1555    1
1027    0
1329    0
Name: Popularity, dtype: category
Categories (2, int64): [0 < 1]
```

In [138]:
```
print(classification_report(y_test, logreg_opt.predict(X_test)))
```

```
              precision    recall  f1-score   support

           0       0.73      0.70      0.71       321
           1       0.67      0.69      0.68       278

    accuracy                           0.70       599
   macro avg       0.70      0.70      0.70       599
weighted avg       0.70      0.70      0.70       599
```

```
In [139]: y_train_pred = logreg_opt.predict(X_train).clip(0, 1)

          # RMSE Train
          LR_rmse_train_lr = np.sqrt(mse(y_train, y_train_pred))
          print(f"RMSE Train = {LR_rmse_train_lr:.6f}")

          #Predicting with the model
          y_test_pred = logreg_opt.predict(X_test).clip(0, 1)

          # RMSE Test
          LR_rmse_test_lr = np.sqrt(mse(y_test, y_test_pred))
          print(f"RMSE Test = {LR_rmse_test_lr:.6f}")

          RMSE Train = 0.344959
          RMSE Test = 0.549700
```

```
In [ ]:   y_train_pred = logreg.predict(X_train).clip(0, 1)

          # RMSE Train
          LR_rmse_train_lr = np.sqrt(mse(y_train, y_train_pred))
          print(f"RMSE Train = {LR_rmse_train_lr:.6f}")

          #Predicting with the model
          y_test_pred = logreg.predict(X_test).clip(0, 1)

          # RMSE Test
          LR_rmse_test_lr = np.sqrt(mse(y_test, y_test_pred))
          print(f"RMSE Test = {LR_rmse_test_lr:.6f}")
```

1a. Logistic Regression Model Reliability Test with Chi-Sq

Test whether the logistic regression model is a random guesser based on "H0: LogReg Model is not a random guesser"

```
In [140]: X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=369)
```

```
In [141]: from scipy.stats import chi2_contingency

          y_pred_lr = logreg_opt.predict(X)
          y_pred__lr1 = pd.Series(y_pred_lr, name='Predicted')
          y_actu = pd.Series(y, name='Actual')
          confusion_lr = pd.crosstab(y_actu, y_pred__lr1)
          print(confusion_lr)

          Predicted    0    1
          Actual
          0          855  184
          1          163  792
```

```
In [142]: data=[[855,184],[163,792]] #Model M1 table
          #Chi square statistic,pvalue,DOF,expected table
          stat, p, dof, expected = chi2_contingency(data)
          print('Chi-square statistic=',stat)
          print('Pvalue=',p)
          alpha=0.01
          if p < alpha:
              print('Not a random guesser')
          else:
              print('Model is a random guesser')

          Chi-square statistic= 844.5076059455982
          Pvalue= 1.1366557178889804e-185
          Not a random guesser
```

2. KNN:

2. KNN

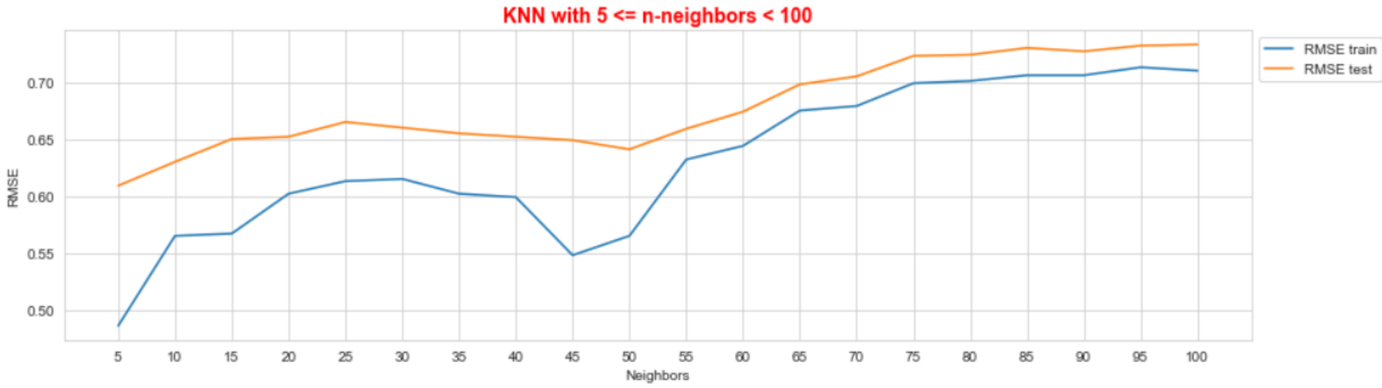
Estimate K as sqrt(n) ~ 45, so we can try K with 5-100 Neighbours

```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)

# compute accuracy of the model
knn.score(X_test, y_test)

# compute the classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.63      0.75      0.68       321
           1       0.63      0.49      0.55       278

    accuracy                           0.63       599
   macro avg       0.63      0.62      0.62       599
weighted avg       0.63      0.63      0.62       599
```

```
RMSE1_train, RMSE1_test = [], []

for i in range(5,105,5):
  knn = KNeighborsClassifier(n_neighbors=i)
  knn.fit(X_train,y_train)
  y_train_pred = knn.predict(X_train)
  knn_train_rmse = np.sqrt(mse(y_train, y_train_pred))
  RMSE1_train.append(knn_train_rmse.round(3))
  y_test_pred = knn.predict(X_test)
  knn_test_rmse = np.sqrt(mse(y_test, y_test_pred))
  RMSE1_test.append(knn_test_rmse.round(3))
```

```
fig, ax = plt.subplots(figsize=(15,4))
x = np.arange(5, 105, 5)
ax = sns.lineplot(x=x, y=RMSE1_train)
sns.lineplot(x=x, y=RMSE1_test, ax=ax)
ax.legend(labels=['RMSE train', 'RMSE test'], bbox_to_anchor=(1, 1))
ax.set_xlabel('Neighbors')
ax.set_ylabel('RMSE')
ax.set_xticks(np.arange(5,101,5))
ax.set_title('KNN with 5 <= n-neighbors < 100', c='r', fontdict={'c':'r', 'fontsize':14, 'weight':'bold'})
plt.show()
```



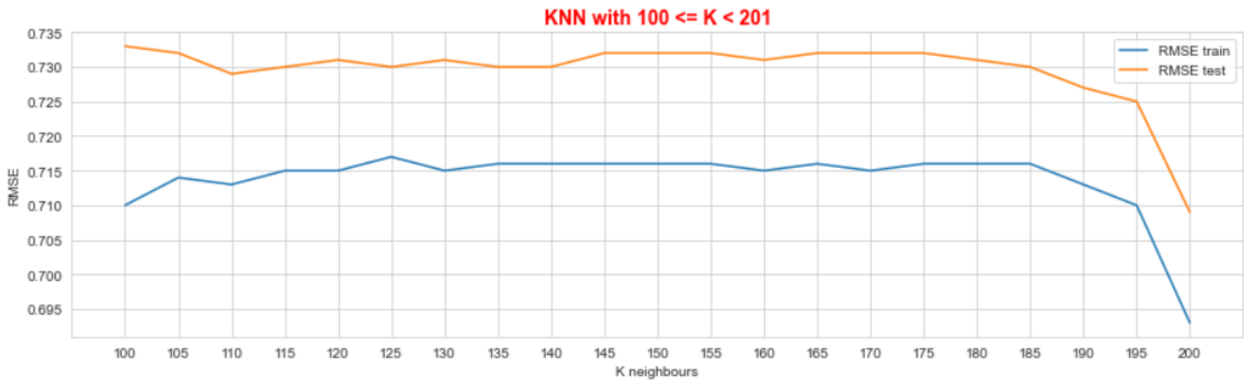

```
gap1 = [RMSE1_test[num]-RMSE1_train[num] for num, i in enumerate(RMSE1_train)]
print(f'RMSE Train: {RMSE1_train[gap1.index(min(gap1))]}, RMSE_test: {RMSE1_test[gap1.index(min(gap1))]}')
```

```
RMSE Train: 0.713, RMSE_test: 0.732
```

Increase K to see if we can lower the RMSE:

```
RMSE_train, RMSE_test = [], []

for i in range(100,201,5):
  knn = KNeighborsClassifier(n_neighbors=i)
  knn.fit(X_train,y_train)
  y_train_pred = knn.predict(X_train)
  knn_train_rmse = np.sqrt(mse(y_train, y_train_pred))
  RMSE_train.append(knn_train_rmse.round(3))
  y_test_pred = knn.predict(X_test)
  knn_test_rmse = np.sqrt(mse(y_test, y_test_pred))
  RMSE_test.append(knn_test_rmse.round(3))
```

```
fig, ax = plt.subplots(figsize=(15,4))
x = np.arange(100, 201, 5)
ax = sns.lineplot(x=x, y=RMSE_train)
sns.lineplot(x=x, y=RMSE_test, ax=ax)
# ax.axvline(x=180, ymin=0, ymax=0.3, color='blue')
# ax.axvline(x=139, ymin=0.5, ymax=0.8, color='orange')
ax.legend(labels=['RMSE train', 'RMSE test'], bbox_to_anchor=(1, 1))
ax.set_xticks(np.arange(100,201,5))
ax.set_xlabel('K neighbours')
ax.set_ylabel('RMSE')
ax.set_title('KNN with 100 <= K < 201', c='r', fontdict={'c':'r', 'fontsize':14, 'weight':'bold'})
plt.show()
```




```
gap2 = [RMSE_test[num]-RMSE_train[num] for num, i in enumerate(RMSE_train)]
print(f'RMSE Train: {RMSE_train[gap2.index(min(gap2))]}, RMSE_test: {RMSE_test[gap2.index(min(gap2))]}')
```

```
RMSE Train: 0.717, RMSE_test: 0.73
```

That worked a bit

```
knn_opt = KNeighborsClassifier(n_neighbors = 30)
knn_opt.fit(X_train, y_train)
y_pred = knn_opt.predict(X_test)

# compute accuracy of the model
knn.score(X_test, y_test)

# compute the classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.57      0.78      0.66       321
           1       0.55      0.31      0.40       278

    accuracy                           0.56       599
   macro avg       0.56      0.55      0.53       599
weighted avg       0.56      0.56      0.54       599
```

3. Decision Tree:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=369)
```

```
tree = DecisionTreeClassifier()
_ = tree.fit(X_train, y_train)

# Evaluate
print('Classification report Decision Tree, 0 = Upper Popular, 1 = Lower Popular\n')
print(classification_report(y_test, tree.predict(X_test)))
```

```
Classification report Decision Tree, 0 = Upper Popular, 1 = Lower Popular

              precision    recall  f1-score   support

           0       0.61      0.59      0.60       321
           1       0.54      0.57      0.56       278

    accuracy                           0.58       599
   macro avg       0.58      0.58      0.58       599
weighted avg       0.58      0.58      0.58       599
```

```
# Perform grid search
param_grid = [
    {'max_depth': [1, 2, 3, 4, 5, 6],
     'criterion': ['entropy', 'gini'],
     'splitter': ['best', 'random']}
]
tree = GridSearchCV(DecisionTreeClassifier(), param_grid)
tree.fit(X_train, y_train)

# Print grid search results
print('Grid search mean and stdev:\n')
scoring = tree.cv_results_
for mean_score, std, params in zip(scoring['mean_test_score'],scoring['std_test_score'],scoring['params']):
    print("{:0.3f} (+/-{:0.03f}) for {}".format(
            mean_score, std * 2, params))


# Evaluate on held-out test
print('\n Original Classification report Decision Tree, 0 = Upper Popular, 1 = Lower Popular\n')
print(classification_report(y_test, tree.predict(X_test)))

# Print best params
print('\nBest parameters:', tree.best_params_)
best_max_depth = tree.best_params_['max_depth']

#Use updated with best params
tree_optimized = DecisionTreeClassifier(criterion = 'entropy', max_depth=best_max_depth, splitter = 'best')
_opt = tree_optimized.fit(X_train, y_train)

print('\n Optimized Classification report Decision Tree, 0 = Upper Popular, 1 = Lower Popular\n')
print(classification_report(y_test, tree_optimized.predict(X_test)))
```

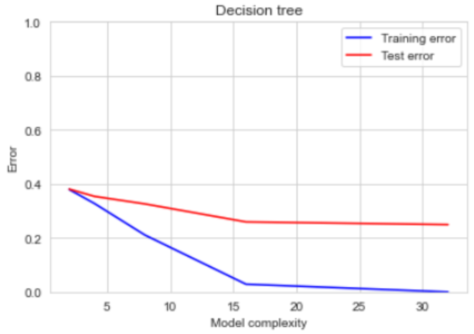
```
Grid search mean and stdev:

0.562 (+/-0.063) for {'criterion': 'entropy', 'max_depth': 1, 'splitter': 'best'}
0.523 (+/-0.041) for {'criterion': 'entropy', 'max_depth': 1, 'splitter': 'random'}
0.586 (+/-0.094) for {'criterion': 'entropy', 'max_depth': 2, 'splitter': 'best'}
0.526 (+/-0.047) for {'criterion': 'entropy', 'max_depth': 2, 'splitter': 'random'}
0.569 (+/-0.087) for {'criterion': 'entropy', 'max_depth': 3, 'splitter': 'best'}
0.531 (+/-0.022) for {'criterion': 'entropy', 'max_depth': 3, 'splitter': 'random'}
0.588 (+/-0.077) for {'criterion': 'entropy', 'max_depth': 4, 'splitter': 'best'}
0.558 (+/-0.065) for {'criterion': 'entropy', 'max_depth': 4, 'splitter': 'random'}
0.587 (+/-0.069) for {'criterion': 'entropy', 'max_depth': 5, 'splitter': 'best'}
0.559 (+/-0.082) for {'criterion': 'entropy', 'max_depth': 5, 'splitter': 'random'}
0.584 (+/-0.110) for {'criterion': 'entropy', 'max_depth': 6, 'splitter': 'best'}
0.591 (+/-0.108) for {'criterion': 'entropy', 'max_depth': 6, 'splitter': 'random'}
0.562 (+/-0.063) for {'criterion': 'gini', 'max_depth': 1, 'splitter': 'best'}
0.549 (+/-0.085) for {'criterion': 'gini', 'max_depth': 1, 'splitter': 'random'}
0.606 (+/-0.082) for {'criterion': 'gini', 'max_depth': 2, 'splitter': 'best'}
0.561 (+/-0.068) for {'criterion': 'gini', 'max_depth': 2, 'splitter': 'random'}
0.584 (+/-0.078) for {'criterion': 'gini', 'max_depth': 3, 'splitter': 'best'}
0.587 (+/-0.102) for {'criterion': 'gini', 'max_depth': 3, 'splitter': 'random'}
0.586 (+/-0.075) for {'criterion': 'gini', 'max_depth': 4, 'splitter': 'best'}
0.591 (+/-0.093) for {'criterion': 'gini', 'max_depth': 4, 'splitter': 'random'}
0.586 (+/-0.063) for {'criterion': 'gini', 'max_depth': 5, 'splitter': 'best'}
0.567 (+/-0.072) for {'criterion': 'gini', 'max_depth': 5, 'splitter': 'random'}
0.587 (+/-0.060) for {'criterion': 'gini', 'max_depth': 6, 'splitter': 'best'}
0.547 (+/-0.067) for {'criterion': 'gini', 'max_depth': 6, 'splitter': 'random'}

 Original Classification report Decision Tree, 0 = Upper Popular, 1 = Lower Popular

              precision    recall  f1-score   support

           0       0.63      0.70      0.66       321
           1       0.60      0.52      0.56       278

    accuracy                           0.62       599
   macro avg       0.62      0.61      0.61       599
weighted avg       0.62      0.62      0.62       599
```

```python
X_td, X_test, y_td, y_test = train_test_split(X, y, train_size=0.7, random_state=5) # so we get the same results
X_train, X_dev, y_train, y_dev = train_test_split(X_td, y_td, test_size=0.33, random_state=5) # so we get the same results
```

```python
import random

NUM_SAMPLES = 10
NUM_TRAIN_SETS = 10

def subsample(X, y, sample_size):
    xy_tuples = list(zip(X, y))
    xy_sample = [random.choice(xy_tuples) for _ in range(sample_size)]
    X_sample, y_sample = zip(*xy_sample)
    return X_sample, y_sample

def error(clf, X, y):
    "Calculate error as 1-accuracy"
    return 1-clf.score(X,y)
```

```python
def bootstrap_error(clf, X_train, y_train, X_test, y_test, sample_size, num_samples=NUM_SAMPLES):
    train_errors = []
    test_errors = []
    for _ in range(num_samples):
        X_sample, y_sample = subsample(X_train, y_train, sample_size)
        clf.fit(X_sample, y_sample)
        train_errors.append(error(clf,X_sample,y_sample))
        test_errors.append(error(clf,X_test,y_test))
    train_error = sum(train_errors)/len(train_errors)
    test_error = sum(test_errors)/len(test_errors)
    return train_error, test_error
```

```python
complexities = []
train_errors = []
test_errors = []
for max_depth in [2,4,8,16,32,None]:
    clf = DecisionTreeClassifier(max_depth=max_depth)
    sample_size = len(y_train)
    train_error, test_error = bootstrap_error(clf, X_train, y_train, X_dev, y_dev, sample_size)
    complexities.append(max_depth)
    train_errors.append(train_error)
    test_errors.append(test_error)
plt.plot(complexities, train_errors, c='b', label='Training error')
plt.plot(complexities, test_errors, c='r', label='Test error')
plt.ylim(0,1)
plt.ylabel('Error')
plt.xlabel('Model complexity')
plt.title('Decision tree')
plt.legend()
plt.show()
# Errors level out at same time
# This suggests that higher values of max_depth may lead to overfitting, as confirmed by best_params
```



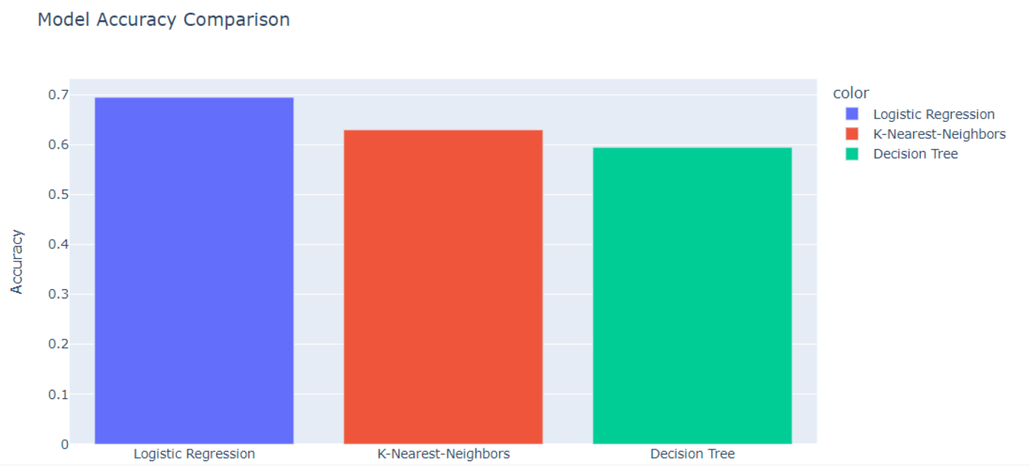

```python
tree_optimized = DecisionTreeClassifier(criterion = 'gini', max_depth=8, splitter = 'best')
_opt = tree_optimized.fit(X_train, y_train)

print('\n Optimized Classification via RMSE graph report Decision Tree, 0 = Upper Popular, 1 = Lower Popular\n')
print(classification_report(y_test, tree_optimized.predict(X_test)))
```

Optimized Classification via RMSE graph report Decision Tree, 0 = Upper Popular, 1 = Lower Popular

	precision	recall	f1-score	support
0	0.57	0.66	0.61	310
1	0.56	0.46	0.50	289
accuracy			0.56	599
macro avg	0.56	0.56	0.56	599
weighted avg	0.56	0.56	0.56	599

→ Plot a comparison graph showing the accuracy comparison of various algorithms on each of your datasets.

```python
fig = px.bar(
    x=["Logistic Regression", "K-Nearest-Neighbors", "Decision Tree"],
    y=[log_acc, knn_acc, dec_acc],
    color=["Logistic Regression", "K-Nearest-Neighbors", "Decision Tree"],
    labels={'x': "Model", 'y': "Accuracy"},
    title="Model Accuracy Comparison"
)

fig.show()
```



→ Use k-fold cross-validation to evaluate your ML algorithm

```
Given the inability to further optimize the Decision Tree models, it is subsequently dropped from the final comparison. While
the RMSE on Decision Tree model did eventually decline to ~0.1, the accuracy results did not increase with further model
complexity.
As such, the only two models for consideration became the KNN against the Logistic Regression. Before these comparisons were
carried out, the two models were tested for reliability using the chi-squared test to see if the model's performance could be
random.
```

```
]: X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=369)
```

```
]: y_pred_lr = logreg_opt.predict(X_test)
   y_pred__lr1 = pd.Series(y_pred_lr, name='Predicted')
   y_pred__lr1
```

```
]: 0      1
   1      0
   2      1
   3      1
   4      1
         ..
   594    1
   595    0
   596    1
   597    1
   598    0
   Name: Predicted, Length: 599, dtype: int64
```

```
: y_pred_frst = knn.predict(X_test)
  y_pred__frst2 = pd.Series(y_pred_frst, name='Predicted')
  y_pred__frst2
```

```
: 0      1
  1      1
  2      1
  3      1
  4      1
        ..
  594    1
  595    1
  596    1
  597    1
  598    1
  Name: Predicted, Length: 599, dtype: int64
```

```
: y_actu = pd.Series(y_test, name='Actual')
  y_actu
```

```
: 853     1
  131     0
  1666    0
  1938    0
  112     1
         ..
  653     1
  1475    0
  553     1
  590     1
  698     0
  Name: Actual, Length: 599, dtype: category
  Categories (2, int64): [0 < 1]
```

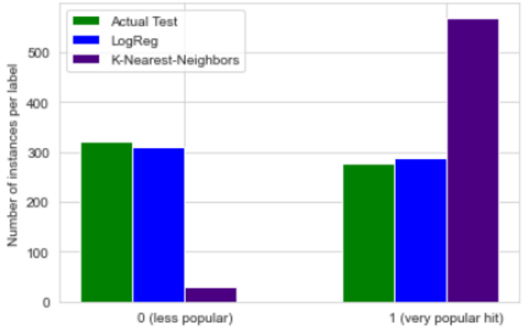
```
y_actu = pd.Series(y_test, name='Actual')
y_actu
```

```
853     1
131     0
1666    0
1938    0
112     1
       ..
653     1
1475    0
553     1
590     1
698     0
Name: Actual, Length: 599, dtype: category
Categories (2, int64): [0 < 1]
```

```
confusion_s1 = pd.crosstab(y_actu, y_pred__lr1)
print(confusion_s1)
print()
confusion_s2 = pd.crosstab(y_actu, y_pred__frst2)
print(confusion_s2)
```

```
Predicted   0   1
Actual
0          64  46
1          41  35

Predicted  0    1
Actual
0          7  103
1          6   70
```

```
from collections import Counter

def class_distr(Y):
    return zip(*sorted(Counter(Y).items()))

actual_classes, actual_freqs = class_distr(y_actu)
sys1_classes, sys1_freqs = class_distr(y_pred__lr1)
sys2_classes, sys2_freqs = class_distr(y_pred__frst2)

bar_width = 0.2

_ = plt.bar([b-(1.5*bar_width) for b in actual_classes], actual_freqs, bar_width, color='green', label='Actual Test')
_ = plt.bar([b-(0.5*bar_width) for b in sys1_classes], sys1_freqs, bar_width, color='blue', label='LogReg')
_ = plt.bar([b+(0.5*bar_width) for b in sys2_classes], sys2_freqs, bar_width, color='indigo', label='K-Nearest-Neighbors')


plt.xticks([0,1], ['0 (less popular)', '1 (very popular hit)'])
plt.ylabel('Number of instances per label')

_ = plt.legend()
```



Results of K-fold validation

```
: import numpy as np
NUM_FOLDS = 10
Y_actu_folds = np.array_split(y_actu, NUM_FOLDS)
Y_sys1_folds = np.array_split(y_pred__lr1, NUM_FOLDS)
Y_sys2_folds = np.array_split(y_pred__frst2, NUM_FOLDS)

freq = Counter(Y_actu_folds[0])
print (freq)
```

```
Counter({0: 34, 1: 26})
```

```
: from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

PRF_KWARGS = {
    'pos_label': 1,     # 1 is the big hit label
    'average': 'binary' # evaluate p/r/f of the positive label
}

Ya0 = Y_actu_folds[0]
Y10 = Y_sys1_folds[0]
Y20 = Y_sys2_folds[0]

y_actu = pd.Series(Ya0, name='Actual')
y_pred_s1 = pd.Series(Y10, name='Predicted')
y_pred_s2 = pd.Series(Y20, name='Predicted')
confusion_s1 = pd.crosstab(y_actu, y_pred_s1)

print("System 1 accuracy:", accuracy_score(Ya0, Y10))
print("System 2 accuracy:", accuracy_score(Ya0, Y20))
print("System 1 f1 score:", f1_score(Ya0, Y10, **PRF_KWARGS))
print("System 2 f1 score:", f1_score(Ya0, Y20, **PRF_KWARGS))
```

```
System 1 accuracy: 0.6833333333333333
System 2 accuracy: 0.5
System 1 f1 score: 0.6779661016949152
System 2 f1 score: 0.625
```

Result and analysis:

Finally, the two models were compared against each other using a paired t-test to test our initial H0 that the Logistic Regression model is significantly better at predicting popularity than other models. Using K-fold cross validation, the final data of both models was split into 10 folds each and a t-test across folds for the mean F1 score yielded the following results:

H0: Logistic Regression mean F1 score > KNN mean F1 score

2. Second data set(top50):

→ Split your dataset into training, validation and testing:

Splitting and Scaling

```
]: y = data.loc[:, 'Popularity']
X = data.drop('Popularity', axis=1)
```

```
]: scaler = StandardScaler()

X = scaler.fit_transform(X)
```

```
]: X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=20)
```

```
]: print("X_train: ",X_train.shape)
print("X_test: ",X_test.shape)
print("y_train: ",y_train.shape)
print("y_test: ",y_test.shape)
```

```
X_train:  (35, 68)
X_test:  (15, 68)
y_train:  (35,)
y_test:  (15,)
```

→ **Create models and estimate their accuracy on unseen data using the specified ML algorithms:**

```
log_model = LogisticRegression()
knn_model = KNeighborsClassifier()
dec_model = DecisionTreeClassifier()
svm_model = SVC()
```

```
log_model.fit(X_train, y_train)
knn_model.fit(X_train, y_train)
dec_model.fit(X_train, y_train)
svm_model.fit(X_train, y_train)
```

```
SVC()
```

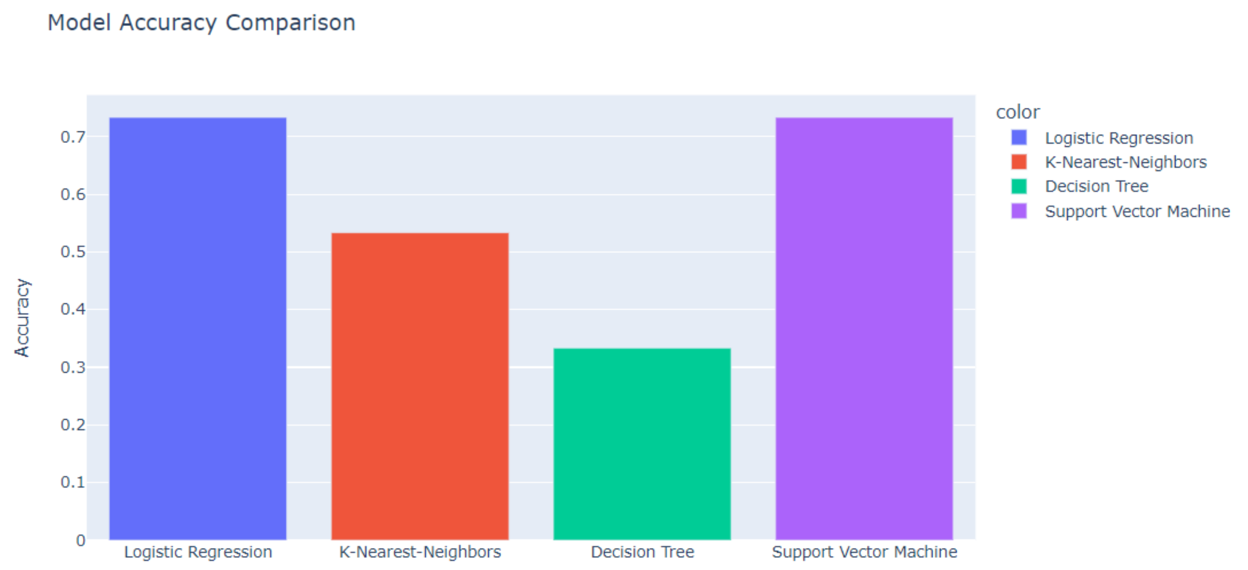
```
log_acc = log_model.score(X_test, y_test)
knn_acc = knn_model.score(X_test, y_test)
dec_acc = dec_model.score(X_test, y_test)
svm_acc = svm_model.score(X_test, y_test)
```

```
print("Logistic Regression Accuracy:", log_acc)
print("K-Nearest-Neighbors Accuracy:", knn_acc)
print("Decision Tree Accuracy:", dec_acc)
print("Support Vector Machine Accuracy:", svm_acc)
```

```
Logistic Regression Accuracy: 0.7333333333333333
K-Nearest-Neighbors Accuracy: 0.5333333333333333
Decision Tree Accuracy: 0.3333333333333333
Support Vector Machine Accuracy: 0.7333333333333333
```

→ Plot a comparison graph showing the accuracy comparison of various algorithms on each of your datasets.

```
fig = px.bar(
    x=["Logistic Regression", "K-Nearest-Neighbors", "Decision Tree", "Support Vector Machine"],
    y=[log_acc, knn_acc, dec_acc, svm_acc],
    color=["Logistic Regression", "K-Nearest-Neighbors", "Decision Tree", "Support Vector Machine"],
    labels={'x': "Model", 'y': "Accuracy"},
    title="Model Accuracy Comparison"
)

fig.show()
```



→ Use k-fold cross-validation to evaluate your ML algorithm

```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=369)
```

```
y_pred_lr = log_model.predict(X_test)
y_pred__lr1 = pd.Series(y_pred_lr, name='Predicted')
y_pred__lr1
```

```
0     1
1     0
2     1
3     0
4     1
5     0
6     0
7     0
8     0
9     0
10    0
11    0
12    0
13    1
14    1
Name: Predicted, dtype: int64
```

```
y_pred_frst = svm_model.predict(X_test)
y_pred__frst2 = pd.Series(y_pred_frst, name='Predicted')
y_pred__frst2
```

```
0     1
1     0
2     1
3     0
4     1
5     0
6     0
7     0
8     0
9     0
10    0
11    0
12    0
13    1
14    1
Name: Predicted, dtype: int64
```

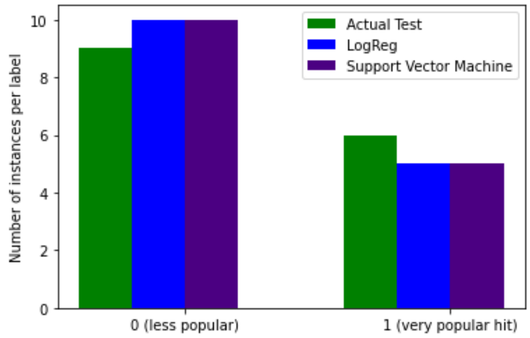
```
y_actu = pd.Series(y_test, name='Actual')
y_actu
```

```
32    1
38    0
17    1
3     0
21    1
15    0
46    0
48    0
37    0
1     1
2     0
5     0
0     0
9     1
16    1
Name: Actual, dtype: category
Categories (2, int64): [0 < 1]
```

```
confusion_s1 = pd.crosstab(y_actu, y_pred__lr1)
print(confusion_s1)
print()
confusion_s2 = pd.crosstab(y_actu, y_pred__frst2)
print(confusion_s2)
```

```
Predicted  0  1
Actual
0          2  2
1          2  0

Predicted  0  1
Actual
0          2  2
1          2  0
```

```
from collections import Counter

def class_distr(Y):
    return zip(*sorted(Counter(Y).items()))

actual_classes, actual_freqs = class_distr(y_actu)
sys1_classes, sys1_freqs = class_distr(y_pred__lr1)
sys2_classes, sys2_freqs = class_distr(y_pred__frst2)

bar_width = 0.2

_ = plt.bar([b-(1.5*bar_width) for b in actual_classes], actual_freqs, bar_width, color='green', label='Actual Test')
_ = plt.bar([b-(0.5*bar_width) for b in sys1_classes], sys1_freqs, bar_width, color='blue', label='LogReg')
_ = plt.bar([b+(0.5*bar_width) for b in sys2_classes], sys2_freqs, bar_width, color='indigo', label='Support Vector Machine')


plt.xticks([0,1], ['0 (less popular)', '1 (very popular hit)'])
plt.ylabel('Number of instances per label')

_ = plt.legend()
```



Results of k-fold validation:

```
import numpy as np
NUM_FOLDS = 10
Y_actu_folds = np.array_split(y_actu, NUM_FOLDS)
Y_sys1_folds = np.array_split(y_pred__lr1, NUM_FOLDS)
Y_sys2_folds = np.array_split(y_pred__frst2, NUM_FOLDS)

freq = Counter(Y_actu_folds[0])
print (freq)
```

```
Counter({1: 1, 0: 1})
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

PRF_KWARGS = {
    'pos_label': 1,     # 1 is the big hit label
    'average': 'binary' # evaluate p/r/f of the positive label
}

Ya0 = Y_actu_folds[0]
Y10 = Y_sys1_folds[0]
Y20 = Y_sys2_folds[0]

y_actu = pd.Series(Ya0, name='Actual')
y_pred_s1 = pd.Series(Y10, name='Predicted')
y_pred_s2 = pd.Series(Y20, name='Predicted')
confusion_s1 = pd.crosstab(y_actu, y_pred_s1)

print("System 1 accuracy:", accuracy_score(Ya0, Y10))
print("System 2 accuracy:", accuracy_score(Ya0, Y20))
print("System 1 f1 score:", f1_score(Ya0, Y10, **PRF_KWARGS))
print("System 2 f1 score:", f1_score(Ya0, Y20, **PRF_KWARGS))
```

```
System 1 accuracy: 1.0
System 2 accuracy: 1.0
System 1 f1 score: 1.0
System 2 f1 score: 1.0
```

→ Result and analysis:

Both models have same accuracy and f1 score after validating using 10 folds. So, here we can either consider logistic regression or svm for the best model.

3. Third data set:(song_data, song_info):

→ Create models and estimate their accuracy on unseen data using the specified ML algorithms:

1. KNN:

KNN Algorithm

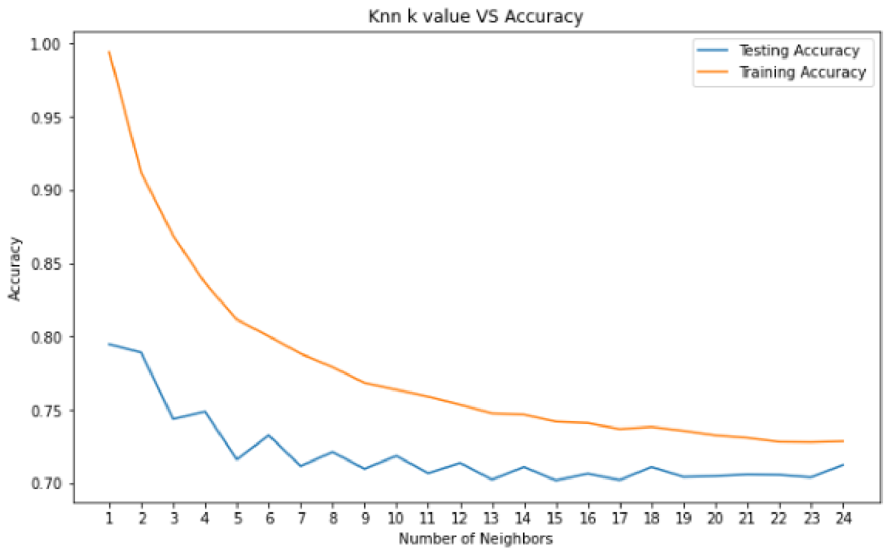
```
# KNN prediction
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 3)
x,y = song_data.loc[:,song_data.columns != 'popularity'], song_data.loc[:,'popularity']
y=y.astype(int)
knn.fit(x,y)
prediction = knn.predict(x)
print('Prediction: {}'.format(prediction))
```

```
Prediction: [1 0 0 ... 0 0 0]
```

```
#KNN Test
knn = KNeighborsClassifier(n_neighbors = 1)
knn.fit(x_train,y_train)
prediction = knn.predict(x_test)
print('With KNN (K=3) train accuracy is: ',knn.score(x_train,y_train))
print('With KNN (K=3) test accuracy is: ',knn.score(x_test,y_test))
```

```
With KNN (K=3) train accuracy is:  0.993989735278228
With KNN (K=3) test accuracy is:  0.7947055645596974
```

```
neig = np.arange(1, 25)
train_accuracy = []
test_accuracy = []

for i, k in enumerate(neig):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(x_train,y_train)
    train_accuracy.append(knn.score(x_train, y_train))
    test_accuracy.append(knn.score(x_test, y_test))

plt.figure(figsize=[10,6])
plt.plot(neig, test_accuracy, label = 'Testing Accuracy')
plt.plot(neig, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.title('Knn k value VS Accuracy')
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.xticks(neig)
plt.savefig('graph.png')
plt.show()
print("Best accuracy is {} with K = {}".format(np.max(test_accuracy),1+test_accuracy.index(np.max(test_accuracy))))
```




```
Best accuracy is 0.7947055645596974 with K = 1
```

```
from sklearn.model_selection import cross_val_score
k = 10
cv_result = cross_val_score(knn,x_train,y_train,cv=k)
cv_result_knn=np.sum(cv_result)/k
print('Cross_val Scores: ',cv_result)
print('Cross_val scores average: ',np.sum(cv_result)/k)
```

```
Cross_val Scores:  [0.70965564 0.7049291  0.71100608 0.7116813  0.70762998 0.69345037
 0.7116813  0.71573261 0.70540541 0.70878378]
Cross_val scores average:  0.7079955563260761
```

```
from sklearn.model_selection import GridSearchCV
grid = {'n_neighbors': np.arange(1,50)}
knn = KNeighborsClassifier()
knn_cv = GridSearchCV(knn, grid, cv=3)
knn_cv.fit(x,y)
print("Tuned hyperparameter k: {}".format(knn_cv.best_params_))
print("Best accuracy: {}".format(knn_cv.best_score_))
```

```
Tuned hyperparameter k: {'n_neighbors': 2}
Best accuracy: 0.773257698541329
```

```
KKN_Score= max(test_accuracy)
CrossVal_KKN_Score=cv_result_knn
```

2. SVM:

```
from sklearn.svm import SVC
svm= SVC(random_state=1)  #kernel='rbf'
svm.fit(x_train,y_train)
print("Train accuracy of svm algo:",svm.score(x_train,y_train))
print("Test accuracy of svm algo:",svm.score(x_test,y_test))
```

```
Train accuracy of svm algo: 0.7071853052404106
Test accuracy of svm algo: 0.7074554294975689
```

```
from sklearn.model_selection import cross_val_score
k = 10
cv_result = cross_val_score(svm,x_train,y_train,cv=k)
cv_result_svm= np.sum(cv_result)/k
print('Cross_val Scores: ',cv_result)
print('Cross_val scores average: ',np.sum(cv_result)/k)
```

```
Cross_val Scores:  [0.70695476 0.70695476 0.70695476 0.70695476 0.70695476 0.70695476
 0.70762998 0.70762998 0.70743243 0.70743243]
Cross_val scores average:  0.7071853386134278
```

```
SVM_score= svm.score(x_test,y_test)
CrossVal_SVM_score=cv_result_svm
```

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

steps = [('scalar', StandardScaler()),
         ('SVM', SVC())]
pipeline = Pipeline(steps)
parameters = {'SVM__C':[1, 10, 100],
              'SVM__gamma':[0.1, 0.01]}
cv = GridSearchCV(pipeline,param_grid=parameters,cv=10)
cv.fit(x_train,y_train)
y_pred = cv.predict(x_test)

print("Tuned Model Parameters: {}".format(cv.best_params_))
print("Test accuracy: {}".format(cv.score(x_test, y_test)))
```

```
Tuned Model Parameters: {'SVM__C': 100, 'SVM__gamma': 0.1}
Test accuracy: 0.7909238249594813
```

3. Decision Tree classifier:

```
from sklearn.metrics import accuracy_score,recall_score,precision_score,confusion_matrix,f1_score
from sklearn.tree import DecisionTreeClassifier

dt= DecisionTreeClassifier()
dt.fit(x_train,y_train)
y_pred=dt.predict(x_test)
DecisionTree_score=dt.score(x_test,y_test)
print("Train ccuracy of decision tree:",dt.score(x_train,y_train))
print("Test accuracy of decision tree:",dt.score(x_test,y_test))
```

```
Train ccuracy of decision tree: 0.9953403565640194
Test accuracy of decision tree: 0.8060507833603457
```

```
from sklearn.model_selection import cross_val_score
k =10
cv_result = cross_val_score(dt,x_train,y_train,cv=k) # uses R^2 as score
print('Cross_val Scores: ',cv_result)
print('Cross_val scores average: ',np.sum(cv_result)/k)
```

```
Cross_val Scores:  [0.80081026 0.78528022 0.80081026 0.79068197 0.79270763 0.7812289
 0.77852802 0.78257934 0.78716216 0.78783784]
Cross_val scores average:  0.7887626603646185
```

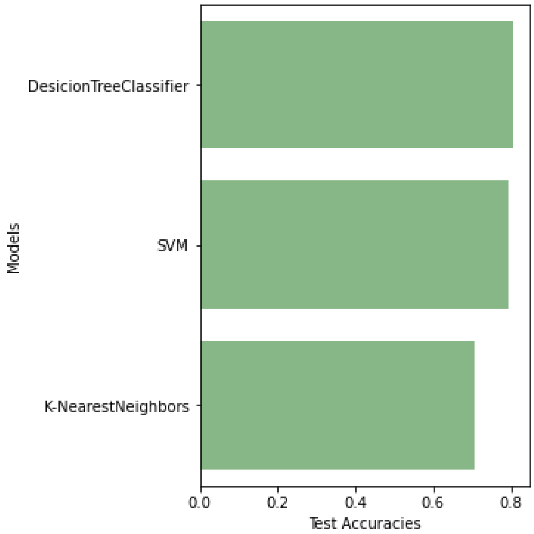
→ Plot a comparison graph showing the accuracy comparison of various algorithms on each of your datasets.

Comparison Of Performance

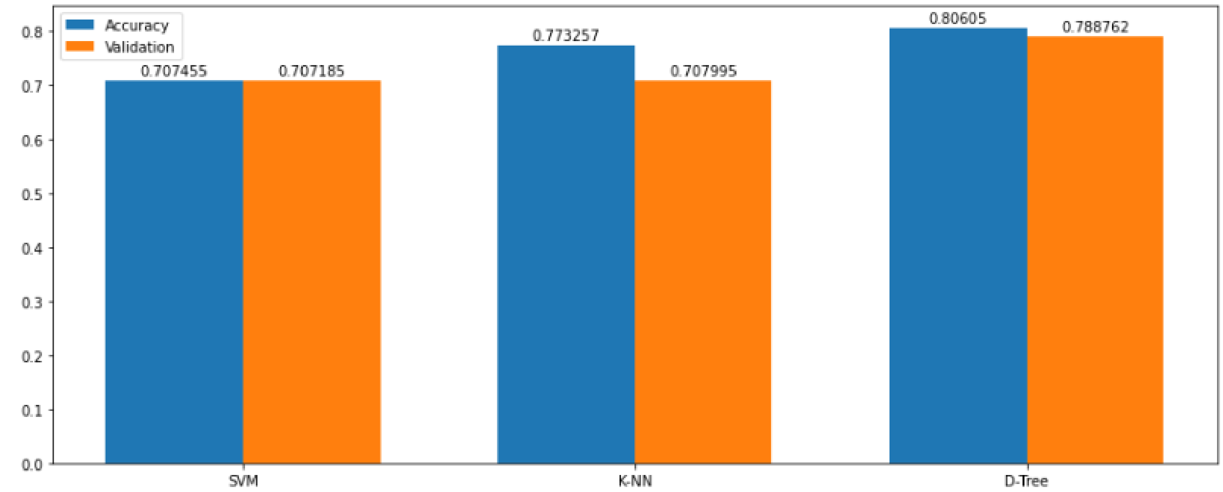
```
model_performances=pd.DataFrame({'Model':['K-NearestNeighbors','DesicionTreeClassifier','SVM'],
                                 'Accuracy':[SVM_score,DecisionTree_score,KKN_Score]})
model_performances.sort_values(by = "Accuracy",ascending=False)
```

	Model	Accuracy
1	DesicionTreeClassifier	0.806051
2	SVM	0.794706
0	K-NearestNeighbors	0.707455

```
model_list= list(model_performances['Model'].unique())
accuracy_list= list(model_performances['Accuracy'].sort_values(ascending=False))
f,ax = plt.subplots(figsize = (4,6))
sns.barplot(x=accuracy_list,y=model_list,color='green',alpha = 0.5)
ax.set(xlabel='Test Accuracies', ylabel='Models')
plt.show()
```



```python
labels = ['SVM', 'K-NN','D-Tree']
accuracy = [0.707455,0.773257,0.806050]
validation = [0.707185, 0.707995,0.788762]

x = np.arange(len(labels))
width = 0.35

fig, ax = plt.subplots(figsize=(12, 5))
rects1 = ax.bar(x - width/2,accuracy, width, label='Accuracy')
rects2 = ax.bar(x + width/2, validation, width, label='Validation')

ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()

def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 2),
                    textcoords="offset points",
                    ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)
fig.tight_layout()
plt.show()
```



→ Use k-fold cross-validation to evaluate your ML algorithm

In [ ]:
```
#k-fold
```

In [27]:
```
from sklearn.metrics import accuracy_score,recall_score,precision_score,confusion_matrix,f1_score
from sklearn.tree import DecisionTreeClassifier

dt= DecisionTreeClassifier()
```

In [40]:
```
from sklearn.svm import SVC
svm= SVC(random_state=1)  #kernel='rbf'
svm.fit(x_train,y_train)
```

Out[40]:
```
SVC(random_state=1)
```

In [41]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=0.7, random_state=369)
```

In [42]:
```
svm.fit(x_train, y_train)
```

Out[42]:
```
SVC(random_state=1)
```

In [43]:
```
dt.fit(x_train, y_train)
```

Out[43]:
```
DecisionTreeClassifier()
```

In [44]:
```
y_pred_lr = svm.predict(x_test)
y_pred__lr1 = pd.Series(y_pred_lr, name='Predicted')
y_pred__lr1
```

Out[44]:
```
0       0
1       0
2       0
3       0
4       0
       ..
5548    0
5549    0
5550    0
5551    0
5552    0
Name: Predicted, Length: 5553, dtype: int64
```

In [45]:
```
y_pred_frst = dt.predict(x_test)
y_pred__frst2 = pd.Series(y_pred_frst, name='Predicted')
y_pred__frst2
```

Out[45]:
```
0       0
1       1
2       0
3       1
4       0
       ..
5548    0
5549    0
5550    0
5551    1
5552    0
Name: Predicted, Length: 5553, dtype: int64
```

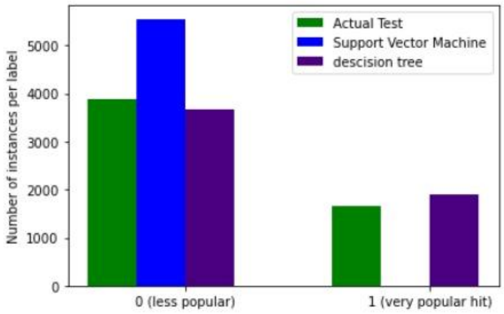
In [46]:
```
y_actu = pd.Series(y_test, name='Actual')
y_actu
```

Out[46]:
```
0       0
1       0
2       0
3       0
4       0
       ..
5548    0
5549    0
5550    0
5551    1
5552    1
Name: Actual, Length: 5553, dtype: int64
```

In [47]:
```
confusion_s1 = pd.crosstab(y_actu, y_pred__lr1)
print(confusion_s1)
print()
confusion_s2 = pd.crosstab(y_actu, y_pred__frst2)
print(confusion_s2)
```

```
Predicted     0
Actual
0          3888
1          1665

Predicted     0     1
Actual
0          3155   733
1           504  1161
```

In [48]:
```
from collections import Counter

def class_distr(Y):
    return zip(*sorted(Counter(Y).items()))

actual_classes, actual_freqs = class_distr(y_actu)
sys1_classes, sys1_freqs = class_distr(y_pred__lr1)
sys2_classes, sys2_freqs = class_distr(y_pred__frst2)

bar_width = 0.2

_ = plt.bar([b-(1.5*bar_width) for b in actual_classes], actual_freqs, bar_width, color='green', label='Actual Test')
_ = plt.bar([b-(0.5*bar_width) for b in sys1_classes], sys1_freqs, bar_width, color='blue', label='Support Vector Machine')
_ = plt.bar([b+(0.5*bar_width) for b in sys2_classes], sys2_freqs, bar_width, color='indigo', label='descision tree')


plt.xticks([0,1], ['0 (less popular)', '1 (very popular hit)'])
plt.ylabel('Number of instances per label')

_ = plt.legend()
```



Results of k-fold validation:

```
In [49]: import numpy as np
         NUM_FOLDS = 10
         Y_actu_folds = np.array_split(y_actu, NUM_FOLDS)
         Y_sys1_folds = np.array_split(y_pred__lr1, NUM_FOLDS)
         Y_sys2_folds = np.array_split(y_pred__frst2, NUM_FOLDS)

         freq = Counter(Y_actu_folds[0])
         print (freq)

         Counter({0: 394, 1: 162})
```

```
In [50]: from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

         PRF_KWARGS = {
             'pos_label': 1,     # 1 is the big hit label
             'average': 'binary' # evaluate p/r/f of the positive label
         }

         Ya0 = Y_actu_folds[0]
         Y10 = Y_sys1_folds[0]
         Y20 = Y_sys2_folds[0]

         y_actu = pd.Series(Ya0, name='Actual')
         y_pred_s1 = pd.Series(Y10, name='Predicted')
         y_pred_s2 = pd.Series(Y20, name='Predicted')
         confusion_s1 = pd.crosstab(y_actu, y_pred_s1)

         print("System 1 accuracy:", accuracy_score(Ya0, Y10))
         print("System 2 accuracy:", accuracy_score(Ya0, Y20))
         print("System 1 f1 score:", f1_score(Ya0, Y10, **PRF_KWARGS))
         print("System 2 f1 score:", f1_score(Ya0, Y20, **PRF_KWARGS))

         System 1 accuracy: 0.7086330935251799
         System 2 accuracy: 0.7841726618705036
         System 1 f1 score: 0.0
         System 2 f1 score: 0.6551724137931035
```

→ Result and analysis:

First we tried to predict popular songs using audio features then we added song name texts' polarity to it and tried to improve our model. In this dataset there is less corelation b/w data hence we cannot use lr as the model to classify it will not give accurate results

We had 18835 songs available. Decision Tree algorithms which mainly given better results when we don't have so much data. As in many popularity studies, we achieved the second best result with SVM. Adding Polarity to features value has almost not changed the result at all. So, our best model will be decision tree.