Importing Essential Libraries

In [1]:
```
%%time
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.figure_factory as ff
import time
import warnings
warnings.filterwarnings('ignore')
import sklearn
```

```
Wall time: 2.56 s
```

In [2]:
```
%%time
data = pd.read_csv('data.csv')
data_by_artist = pd.read_csv('data_by_artist.csv')
data_by_genres = pd.read_csv('data_by_genres.csv')
data_by_year = pd.read_csv('data_by_year.csv')
data_w_genres = pd.read_csv('data_w_genres.csv')
```

```
Wall time: 883 ms
```

In [3]:
```
data.head(2)
```

Out[3]:

	valence	year	acousticness	artists	danceability	duration_ms	energy	explicit	id	instrumentalness	key	liveness	loudness	mode	name	popularity	release_date	speechine
0	0.0594	1921	0.982	['Sergei Rachmaninoff', 'James Levine', 'Berli...	0.279	831667	0.211	0	4BJqT0PrAfrxzMOxytFOIz	0.878	10	0.665	-20.096	1	Piano Concerto No. 3 in D Minor, Op. 30: III. ...	4	1921	0.03
1	0.9630	1921	0.732	['Dennis Day']	0.819	180533	0.341	0	7xPhfUan2yNtyFG0cUWkt8	0.000	7	0.160	-12.441	1	Clancy Lowered the Boom	5	1921	0.41

In [4]:
```
data.shape, data_by_artist.shape, data_by_genres.shape, data_by_year.shape, data_w_genres.shape
```

Out[4]:
```
((170653, 19), (28680, 15), (2973, 14), (100, 14), (28680, 16))
```

In [5]:
```
data['year'].unique()
```

Out[5]:
```
array([1921, 1922, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 1930, 1931,
       1932, 1933, 1934, 1935, 1936, 1937, 1938, 1939, 1940, 1941, 1942,
       1943, 1944, 1945, 1946, 1947, 1948, 1949, 1950, 1951, 1952, 1953,
       1954, 1955, 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964,
       1965, 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975,
       1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986,
       1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997,
       1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008,
       2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019,
       2020], dtype=int64)
```

In [6]:
```
dd = pd.DataFrame(data.year.unique())
dd.count()
```

Out[6]:
```
0    100
dtype: int64
```

In [7]:
```
data.isnull().sum()
```

Out[7]:
```
valence             0
year                0
acousticness        0
artists             0
danceability        0
duration_ms         0
energy              0
explicit            0
id                  0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
name                0
popularity          0
release_date        0
speechiness         0
tempo               0
dtype: int64
```

In [8]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170653 entries, 0 to 170652
Data columns (total 19 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   valence           170653 non-null  float64
 1   year              170653 non-null  int64  
 2   acousticness      170653 non-null  float64
 3   artists           170653 non-null  object 
 4   danceability      170653 non-null  float64
 5   duration_ms       170653 non-null  int64  
 6   energy            170653 non-null  float64
 7   explicit          170653 non-null  int64  
 8   id                170653 non-null  object 
 9   instrumentalness  170653 non-null  float64
 10  key               170653 non-null  int64  
 11  liveness          170653 non-null  float64
 12  loudness          170653 non-null  float64
 13  mode              170653 non-null  int64  
 14  name              170653 non-null  object 
 15  popularity        170653 non-null  int64  
 16  release_date      170653 non-null  object 
 17  speechiness       170653 non-null  float64
 18  tempo             170653 non-null  float64
dtypes: float64(9), int64(6), object(4)
memory usage: 24.7+ MB
```

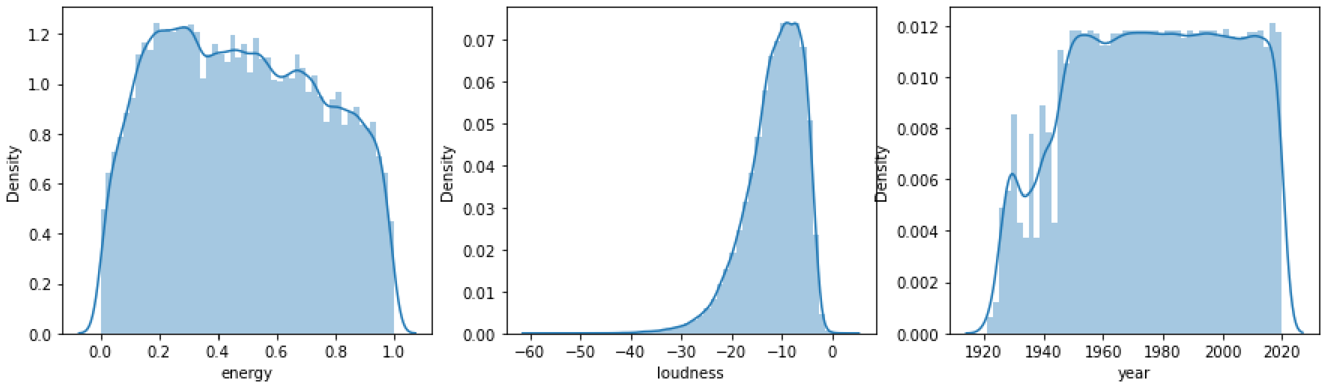
In [9]:
```
total = data.shape[0]
popularity_more_than_40 = data[data['popularity'] > 40].shape[0]

probability = (popularity_more_than_40/total)*100
print("Probability of song getting more than 40 in popularity is", probability)
```

```
Probability of song getting more than 40 in popularity is 38.141140208493255
```

In [10]:
```
%%time
plt.rcParams['figure.figsize'] = (15,4)

plt.subplot(1, 3, 1)
sns.distplot(data['energy'])

plt.subplot(1, 3, 2)
sns.distplot(data['loudness'])

plt.subplot(1, 3, 3)
sns.distplot(data['year']);
```

```
Wall time: 2.53 s
```

Out[10]: <AxesSubplot:xlabel='year', ylabel='Density'>



In [11]:
```
%%time
#sns.pairplot(data.select_dtypes(exclude = ['object']));
```

```
Wall time: 0 ns
```

In [12]:
```
data2 = data.copy()
data2.groupby(['year', 'popularity']).mean().reset_index().head(10)
```

Out[12]:

	year	popularity	valence	acousticness	danceability	duration_ms	energy	explicit	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	1921	0	0.455633	0.884240	0.440758	210494.283019	0.247720	0.066038	0.293568	5.009434	0.213373	-16.782019	0.613208	0.079406	101.945953
1	1921	1	0.246472	0.930778	0.347406	327790.277778	0.199078	0.000000	0.477304	6.222222	0.199056	-16.962389	0.944444	0.050172	105.945556
2	1921	2	0.102564	0.891929	0.356714	431415.357143	0.163079	0.000000	0.490511	5.000000	0.163643	-17.933071	0.642857	0.056929	99.284214
3	1921	3	0.097900	0.977000	0.283500	297833.500000	0.192950	0.000000	0.436514	3.500000	0.224000	-17.157500	1.000000	0.037350	84.855000
4	1921	4	0.180060	0.971000	0.379600	384821.200000	0.274200	0.000000	0.639000	4.400000	0.274620	-16.415000	0.800000	0.055140	90.283800
5	1921	5	0.359350	0.669000	0.415000	478323.000000	0.136142	0.000000	0.229723	5.000000	0.103475	-23.281000	0.750000	0.127950	96.375250
6	1921	6	0.196000	0.579000	0.697000	395076.000000	0.346000	0.000000	0.168000	2.000000	0.130000	-12.506000	1.000000	0.070000	119.824000
7	1922	0	0.543435	0.941623	0.480580	165806.507246	0.234085	0.000000	0.435994	5.318841	0.242668	-19.437710	0.623188	0.112484	100.266304
8	1922	4	0.400000	0.994000	0.420000	180800.000000	0.288000	0.000000	0.000216	7.000000	0.196000	-14.005000	1.000000	0.070100	139.575000
9	1922	6	0.127000	0.674000	0.645000	126903.000000	0.445000	0.000000	0.744000	0.000000	0.151000	-13.338000	1.000000	0.451000	104.851000

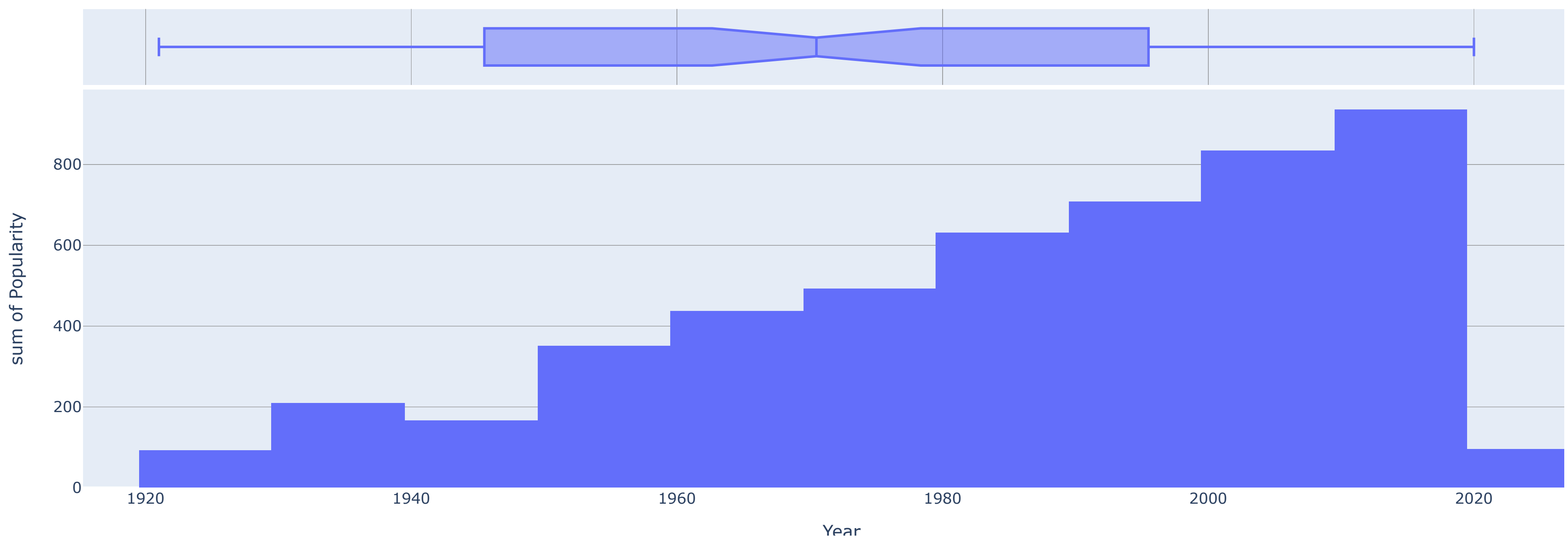
In [13]:
```
%%time
features = data.drop(columns = ['year', 'duration_ms', 'artists', 'id', 'name', 'release_date'], axis = 1)
features.head()
```

```
Wall time: 7.99 ms
```

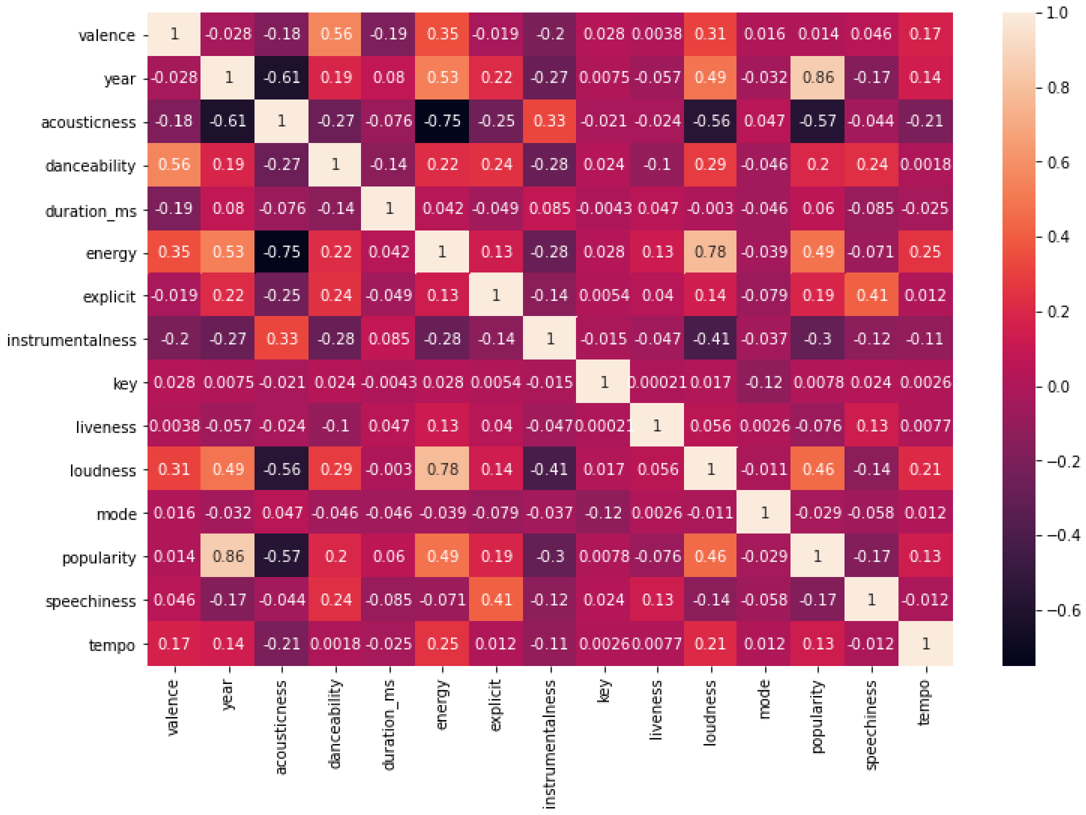
Out[13]:

	valence	acousticness	danceability	energy	explicit	instrumentalness	key	liveness	loudness	mode	popularity	speechiness	tempo
0	0.0594	0.982	0.279	0.211	0	0.878000	10	0.665	-20.096	1	4	0.0366	80.954
1	0.9630	0.732	0.819	0.341	0	0.000000	7	0.160	-12.441	1	5	0.4150	60.936
2	0.0394	0.961	0.328	0.166	0	0.913000	3	0.101	-14.850	1	5	0.0339	110.339
3	0.1650	0.967	0.275	0.309	0	0.000028	5	0.381	-9.316	1	3	0.0354	100.109
4	0.2530	0.957	0.418	0.193	0	0.000002	3	0.229	-10.096	1	2	0.0380	101.665

In [14]:
```
yy = data2.year.unique()
pp = data2.popularity.unique()
px.histogram(data2, yy, pp, marginal = 'box', labels = {'x' : 'Year', 'y' : 'Popularity'})
```



In [15]:
```
plt.figure(figsize = (12, 8))
sns.heatmap(data = data2.corr(), annot = True);
```



In [16]:
```
data2.isnull().sum()
```

Out[16]:
```
valence             0
year                0
acousticness        0
artists             0
danceability        0
duration_ms         0
energy              0
explicit            0
id                  0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
name                0
popularity          0
release_date        0
speechiness         0
tempo               0
dtype: int64
```

Train Test Split

In [17]:
```
X = data2.drop(columns = ['popularity', 'artists', 'id', 'name', 'release_date'])
y = data2['popularity']
```

In [18]:
```
X.head()
```

Out[18]:

	valence	year	acousticness	danceability	duration_ms	energy	explicit	instrumentalness	key	liveness	loudness	mode	speechiness	tempo
0	0.0594	1921	0.982	0.279	831667	0.211	0	0.878000	10	0.665	-20.096	1	0.0366	80.954
1	0.9630	1921	0.732	0.819	180533	0.341	0	0.000000	7	0.160	-12.441	1	0.4150	60.936
2	0.0394	1921	0.961	0.328	500062	0.166	0	0.913000	3	0.101	-14.850	1	0.0339	110.339
3	0.1650	1921	0.967	0.275	210000	0.309	0	0.000028	5	0.381	-9.316	1	0.0354	100.109
4	0.2530	1921	0.957	0.418	166693	0.193	0	0.000002	3	0.229	-10.096	1	0.0380	101.665

In [19]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

In [20]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from xgboost import XGBRegressor
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
```

In [21]:
```
sc = StandardScaler()
sc.fit_transform(X_train[['duration_ms', 'loudness', 'tempo']])
sc.fit(X_test[['duration_ms', 'loudness', 'tempo']])
```

Out[21]:
```
StandardScaler()
```

In [22]:
```
pipeline_rf = Pipeline([('rf_regressor', RandomForestRegressor())], verbose = 10)

pipeline_dt = Pipeline([('dt_regressor', DecisionTreeRegressor())], verbose = 10)

pipeline_xgb = Pipeline([('xgb_regressor', XGBRegressor())], verbose = 10)
```

In [23]:
```
pipelines = [pipeline_rf, pipeline_dt, pipeline_xgb]
```

In [24]:
```
best_accuracy = 0.0
best_classifier = 0
best_pipeline = ""
```

In [25]:
```
pipe_dict = {0: 'Random Forest', 1: 'Decision Tree', 2: 'XGBoost'}

for pipe in pipelines:
    pipe.fit(X_train, y_train)
```

```
[Pipeline] ...... (step 1 of 1) Processing rf_regressor, total= 2.3min
[Pipeline] ...... (step 1 of 1) Processing dt_regressor, total=   2.5s
[Pipeline] ..... (step 1 of 1) Processing xgb_regressor, total=   7.5s
```

In [26]:
```
for i, model in enumerate(pipelines):
    print("{} Test Accuracy: {}".format(pipe_dict[i], round(model.score(X_test, y_test) * 100, 2)))
```

```
Random Forest Test Accuracy: 80.76
Decision Tree Test Accuracy: 59.95
XGBoost Test Accuracy: 80.79
```

In [27]:
```
y_pred = pipeline_xgb.predict(X_test)
```

In [28]:
```
# REGRESSION METRICS
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

print('R2 Score is :', r2_score(y_test, y_pred))
print('Mean Squared Error is :', mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error is :', np.sqrt(mean_squared_error(y_test, y_pred)))
print('Mean Absolute Error is :', mean_absolute_error(y_test, y_pred))
```

```
R2 Score is : 0.8079437582298994
Mean Squared Error is : 91.58713188596205
Root Mean Squared Error is : 9.570116607751551
Mean Absolute Error is : 6.766035515770487
```

In [29]:
```
# RandomizedSearchCV for XGBoost
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV

params={
 "xgb_regressor__n_estimators"     : [100, 200, 300],
 "xgb_regressor__learning_rate"    : [0.05, 0.10, 0.15, 0.20, 0.25, 0.30 ] ,
 "xgb_regressor__max_depth"        : [ 3, 4, 5, 6, 8, 10, 12, 15],
 "xgb_regressor__min_child_weight" : [ 1, 3, 5, 7 ],
 "xgb_regressor__gamma"            : [ 0.0, 0.1, 0.2 , 0.3, 0.4 ],
 "xgb_regressor__colsample_bytree" : [ 0.3, 0.4, 0.5 , 0.7 ]

}

pipeline_xgb_random = RandomizedSearchCV(estimator = pipeline_xgb, param_distributions = params, n_iter = 50,
                                         scoring = 'r2' , n_jobs = -1, cv = 10, verbose = 3)
```

In [30]:
```
pipeline_xgb_random.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 50 candidates, totalling 500 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:  5.4min
[Parallel(n_jobs=-1)]: Done 112 tasks      | elapsed: 17.1min
[Parallel(n_jobs=-1)]: Done 272 tasks      | elapsed: 34.6min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed: 68.4min finished
[Pipeline] ..... (step 1 of 1) Processing xgb_regressor, total=  38.1s
```

Out[30]:
```
RandomizedSearchCV(cv=10,
                   estimator=Pipeline(steps=[('xgb_regressor',
                                              XGBRegressor(base_score=0.5,
                                                           booster='gbtree',
                                                           colsample_bylevel=1,
                                                           colsample_bynode=1,
                                                           colsample_bytree=1,
                                                           gamma=0, gpu_id=-1,
                                                           importance_type='gain',
                                                           interaction_constraints='',
                                                           learning_rate=0.300000012,
                                                           max_delta_step=0,
                                                           max_depth=6,
                                                           min_child_weight=1,
                                                           missing=nan,
                                                           monotone_constraints='()'...
                   n_iter=50, n_jobs=-1,
                   param_distributions={'xgb_regressor__colsample_bytree': [0.3,
                                                                            0.4,
                                                                            0.5,
                                                                            0.7],
                                        'xgb_regressor__gamma': [0.0, 0.1, 0.2,
                                                                 0.3, 0.4],
                                        'xgb_regressor__learning_rate': [0.05,
                                                                         0.1,
                                                                         0.15,
                                                                         0.2,
                                                                         0.25,
                                                                         0.3],
                                        'xgb_regressor__max_depth': [3, 4, 5, 6,
                                                                     8, 10, 12,
                                                                     15],
                                        'xgb_regressor__min_child_weight': [1,
                                                                            3,
                                                                            5,
                                                                            7],
                                        'xgb_regressor__n_estimators': [100,
                                                                        200,
                                                                        300]},
                   scoring='r2', verbose=3)
```

In [31]:
```
# XGBoost Accuracy
acc_xgb = round(pipeline_xgb_random.score(X_test, y_test) * 100, 2)
print(acc_xgb)
```

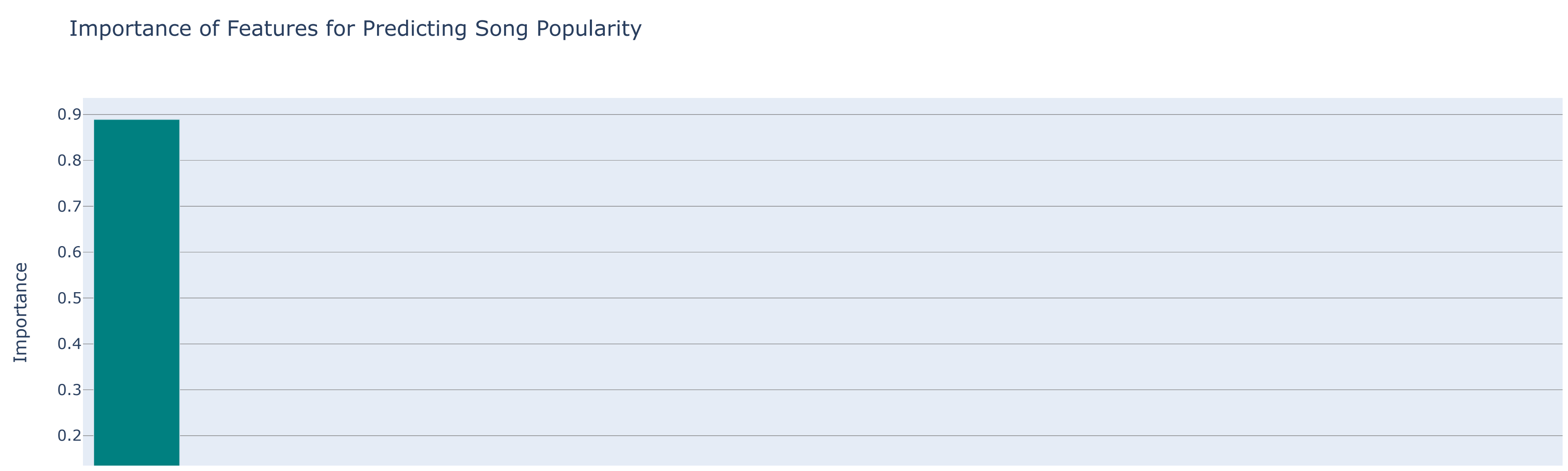
```
80.84
```

In [32]:
```
y_pred_tuned = pipeline_xgb_random.predict(X_test)
```

In [33]:
```
# REGRESSION METRICS
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

print('R2 Score is :', r2_score(y_test, y_pred_tuned))
print('Mean Squared Error is :', mean_squared_error(y_test, y_pred_tuned))
print('Root Mean Squared Error is :', np.sqrt(mean_squared_error(y_test, y_pred_tuned)))
print('Mean Absolute Error is :', mean_absolute_error(y_test, y_pred_tuned))
```

```
R2 Score is : 0.808446954003891
Mean Squared Error is : 91.34716958485556
Root Mean Squared Error is : 9.557571322509478
Mean Absolute Error is : 6.813602463109473
```

In [34]:
```
ab = pd.Series(pipeline_xgb.named_steps["xgb_regressor"].feature_importances_, X_train.columns).sort_values(ascending = False)
px.bar(ab, labels = {'index' : 'Features', 'value' : 'Importance'},
       title = 'Importance of Features for Predicting Song Popularity', color_discrete_sequence = ['teal'])
```



Results for root mean squared error as scoring method

Without Parameter Tuning

R2 Score is : 0.8079437582298994

Mean Squared Error is : 91.58713188596205

Root Mean Squared Error is : 9.570116607751551

Mean Absolute Error is : 6.766035515770487

With Parameter Tuning

R2 Score is : 0.8071561848207268

Mean Squared Error is : 91.96270723321955

Root Mean Squared Error is : 9.589718829726946

Mean Absolute Error is : 6.844903688846095

Results for R2 Score as scoring method

Without Parameter Tuning

R2 Score is : 0.8079437582298994

Mean Squared Error is : 91.58713188596205

Root Mean Squared Error is : 9.570116607751551

Mean Absolute Error is : 6.766035515770487

With Parameter Tuning

R2 Score is : 0.808434817891743

Mean Squared Error is : 91.35295701302573

Root Mean Squared Error is : 9.557874084388523

Mean Absolute Error is : 6.804451638059193

Results for R2 Score as scoring method with n_iters 50, cv 10

On IBM Watson Studio with n_iters 50, cv 10

R2 Score is : 0.8101521044031971

Mean Squared Error is : 90.53402322175225

Root Mean Squared Error is : 9.51493684801703

Mean Absolute Error is : 6.76631248706134

On Local Machine (This Notebook)

Without Tuning

R2 Score is : 0.8079437582298994

Mean Squared Error is : 91.58713188596205

Root Mean Squared Error is : 9.570116607751551

Mean Absolute Error is : 6.766035515770487

With Tuning

R2 Score is : 0.808446954003891

Mean Squared Error is : 91.34716958485556

Root Mean Squared Error is : 9.557571322509478

Mean Absolute Error is : 6.813602463109473

Further parameter optimizations should give us better accuracy