Spotify Popularity Predictive Model

Author: Carlos Garza

Overview

This notebook details the building, tuning, and deployment of a model that predicts a song's popularity on spotify.

Multiple machine learning models were trained using data from the Spotify API and were auditioned using a subset of testing data to determine which model best predicts a song's popularity.

The model was tuned using a grid search and used in a python script that predicts a song's popularity on a scale of 0-100.

Business Problem

In the music industry, an important metric that has surfaced in the last few years is an artist's Spotify numbers. Many entities in the industry, from venues to record labels, will check an artist's spotify numbers before choosing to work with or book said artist and, for better or worse, will base their decision in part on the artist's performance on the platform.

This increase in Spotify metric importance has opened opportunities for optimization in the pop and Nashville country music workflow. Typically, a producer or songwriter will rent studio time and hire studio musicians to produce singles that can then be pitched to artists. Artists buy these songs and rerecord them with their own studio teams to be released as singles or as part of a record.

If producers had a model that they could use to evaluate their music while in the production and could deliver a model's predictions while pitching music, they would have a new edge in the industry.

Data

The data for this project originates from the Spotify API. The data used for this model training was organized and uploaded to Kaggle by user Yamac Eren Ay, and can be found here.

The data describes 174,389 songs. Specifics features of the data are explored below.

In [1]:
```
# import standard libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
# import data
df = pd.read_csv('data/data.csv')
df.head()
```

Out[2]:

	acousticness	artists	danceability	duration_ms	energy	explicit	ic
0	0.991000	['Mamie Smith']	0.598	168333	0.224	0	0cS0A1fUEUd1EW3FcF8AE
1	0.643000	["Screamin' Jay Hawkins"]	0.852	150200	0.517	0	0hbkKFIJm7Z05H8ZI9w30
2	0.993000	['Mamie Smith']	0.647	163827	0.186	0	11m7laMUgmOKqI3oYzuhn
3	0.000173	['Oscar Velazquez']	0.730	422087	0.798	0	19Lc5SfJJ5O1oaxY0fpwfl
4	0.295000	['Mixe']	0.704	165224	0.707	1	2hJjbsLCytGsnAHfdsLej

EDA

Before processing any data, the data will be visualized so any missing or outlier data can be managed.

The overview of the dataframe shows that there are no missing values. However, the `id` column can be dropped as it points to specific songs rather than quantifying song details. Also, the `release_date` column only holds month and day data for some rows. The rest of the rows only show release year, which is redundant to the info in the `year` column. Because of this overlap, the `release_date` column will be dropped.

In [3]:
```
# view dataframe info
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 174389 entries, 0 to 174388
Data columns (total 19 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   acousticness      174389 non-null  float64
 1   artists           174389 non-null  object 
 2   danceability      174389 non-null  float64
 3   duration_ms       174389 non-null  int64  
 4   energy            174389 non-null  float64
 5   explicit          174389 non-null  int64  
 6   id                174389 non-null  object 
 7   instrumentalness  174389 non-null  float64
 8   key               174389 non-null  int64  
 9   liveness          174389 non-null  float64
 10  loudness          174389 non-null  float64
 11  mode              174389 non-null  int64  
 12  name              174389 non-null  object 
 13  popularity        174389 non-null  int64  
```

```
 14  release_date      174389 non-null  object 
 15  speechiness       174389 non-null  float64
 16  tempo             174389 non-null  float64
 17  valence           174389 non-null  float64
 18  year              174389 non-null  int64  
dtypes: float64(9), int64(6), object(4)
memory usage: 25.3+ MB
```

In [4]:
```
# drop unnecessary columns
to_drop = ['id', 'release_date']
df.drop(to_drop, axis=1, inplace=True)
```

Moving forward, it is useful to separate the dataframe into continuous and categorical columns for further analysis.

In [5]:
```
# split df columns into categorical and continuous columns
cat_columns = ['artists', 'explicit', 'key', 'mode', 'name']
cont_columns = ['acousticness', 'danceability', 'duration_ms',
                'energy', 'instrumentalness', 'liveness',
                'loudness', 'popularity', 'speechiness',
                'tempo', 'valence', 'year']
```

In [6]:
```
# create continuous df and categorical df
cont_df = df[cont_columns]
cat_df = df[cont_columns]

# preview continuous dataframe
cont_df.head()
```

Out[6]:

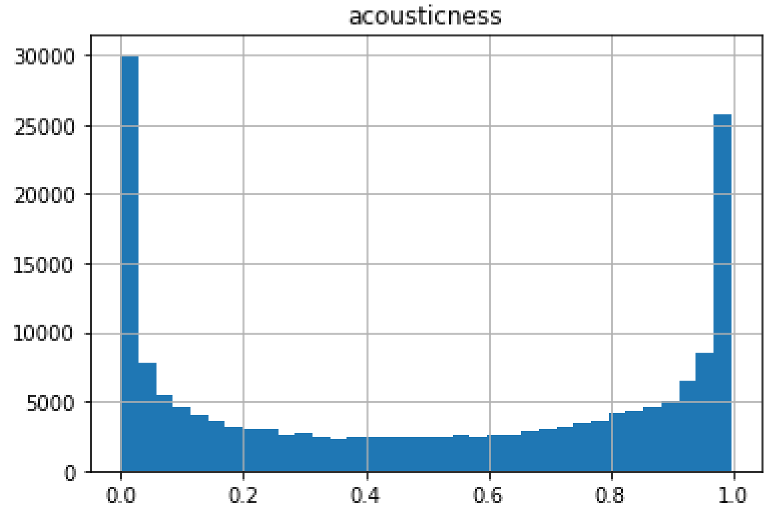
	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	popula
0	0.991000	0.598	168333	0.224	0.000522	0.3790	-12.628	
1	0.643000	0.852	150200	0.517	0.026400	0.0809	-7.261	
2	0.993000	0.647	163827	0.186	0.000018	0.5190	-12.098	
3	0.000173	0.730	422087	0.798	0.801000	0.1280	-7.311	
4	0.295000	0.704	165224	0.707	0.000246	0.4020	-6.036	

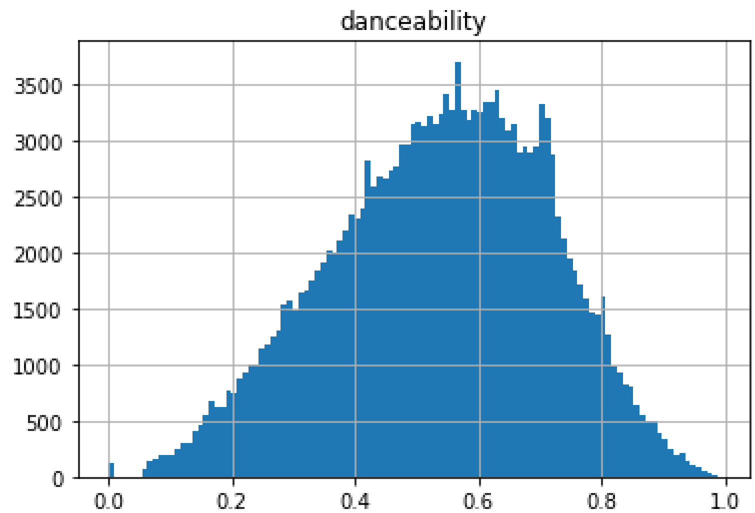
In [7]:
```
# preview categorical dataframe
cat_df.head()
```

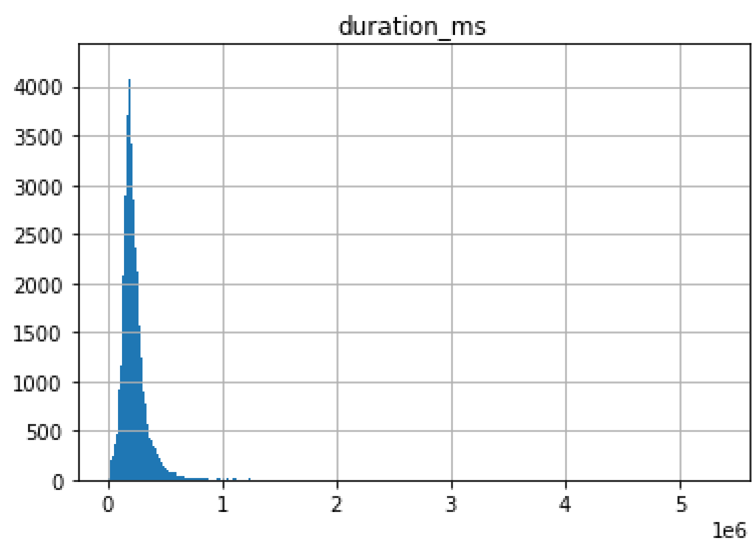
Out[7]:

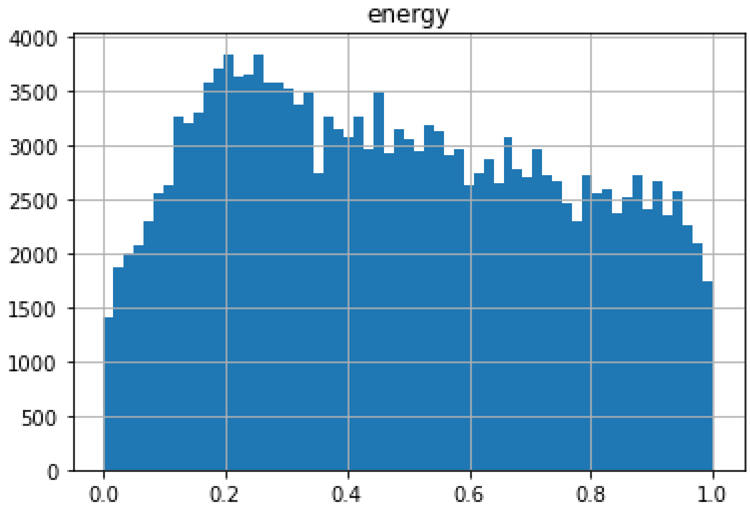
	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness	popula
0	0.991000	0.598	168333	0.224	0.000522	0.3790	-12.628	
1	0.643000	0.852	150200	0.517	0.026400	0.0809	-7.261	
2	0.993000	0.647	163827	0.186	0.000018	0.5190	-12.098	
3	0.000173	0.730	422087	0.798	0.801000	0.1280	-7.311	
4	0.295000	0.704	165224	0.707	0.000246	0.4020	-6.036	

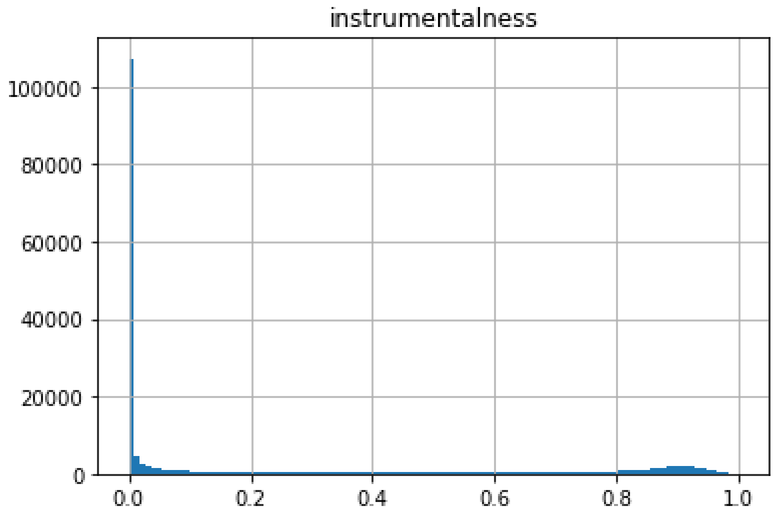
In [8]:
```
# view histograms of continuous variables
for column in cont_df.columns:
    plt.figure()
    cont_df[column].hist(bins='auto')
    plt.title(column)
```

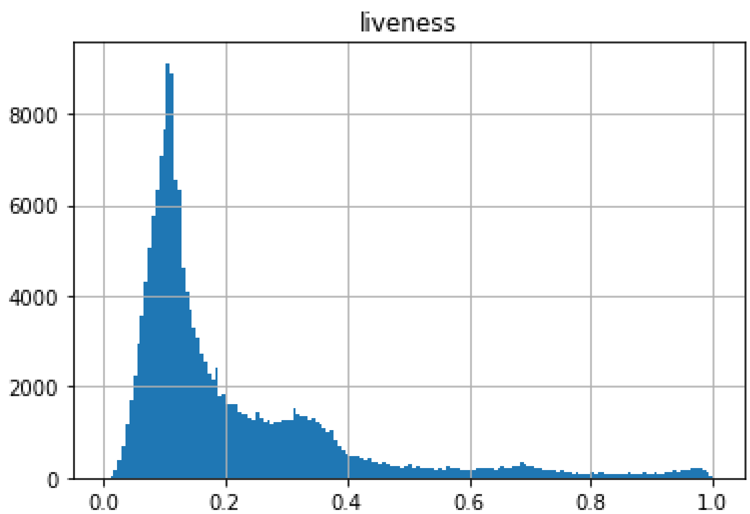

acousticness
30000
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

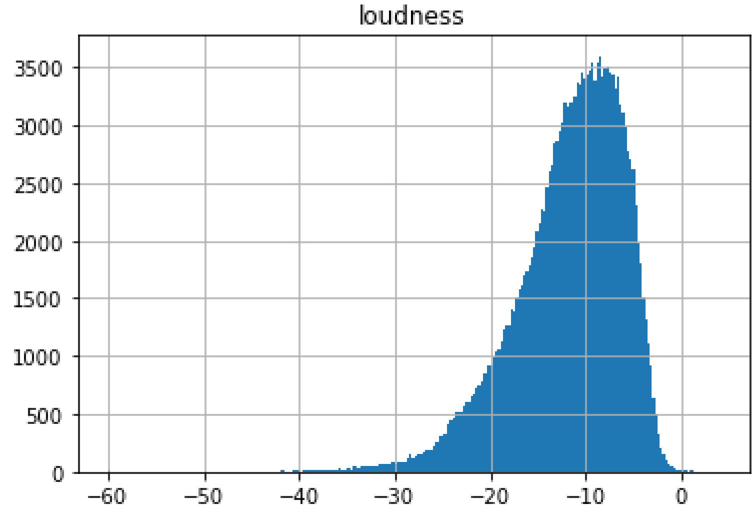

danceability
3500
3000
2500
2000
1500
1000
500
0
0.0
0.2
0.4
0.6
0.8
1.0

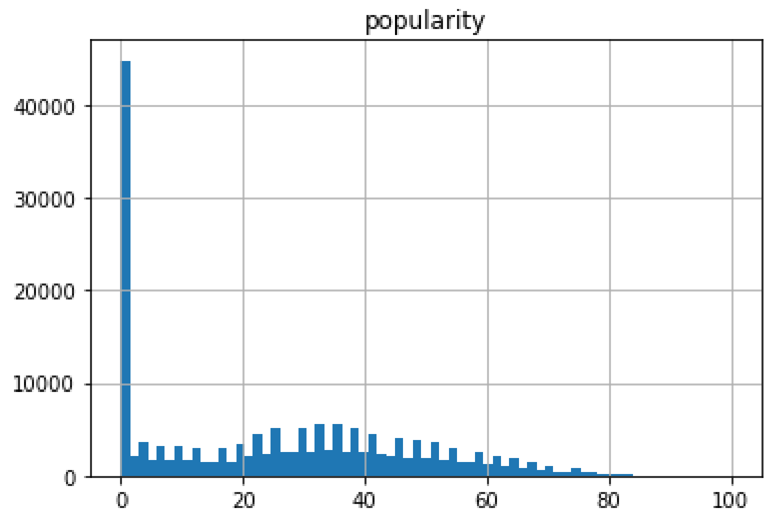

duration_ms
4000
3500
3000
2500
2000
1500
1000
500
0
0
1
2
3
4
5
1e6

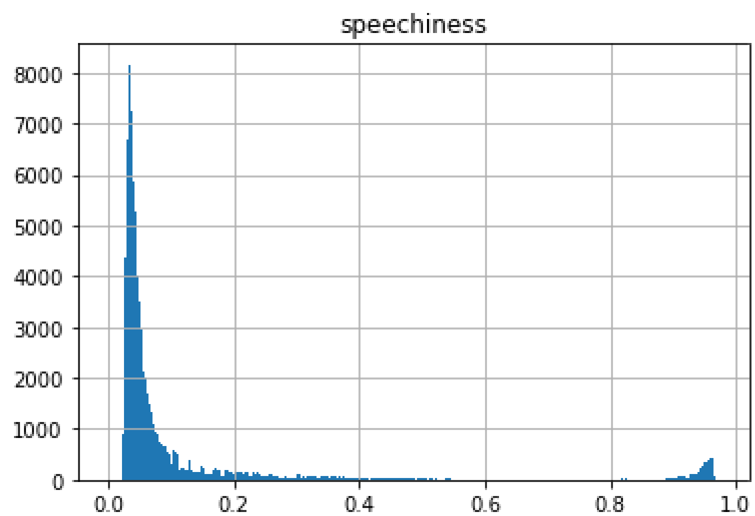

energy
4000
3500
3000
2500
2000
1500
1000
500
0
0.0
0.2
0.4
0.6
0.8
1.0

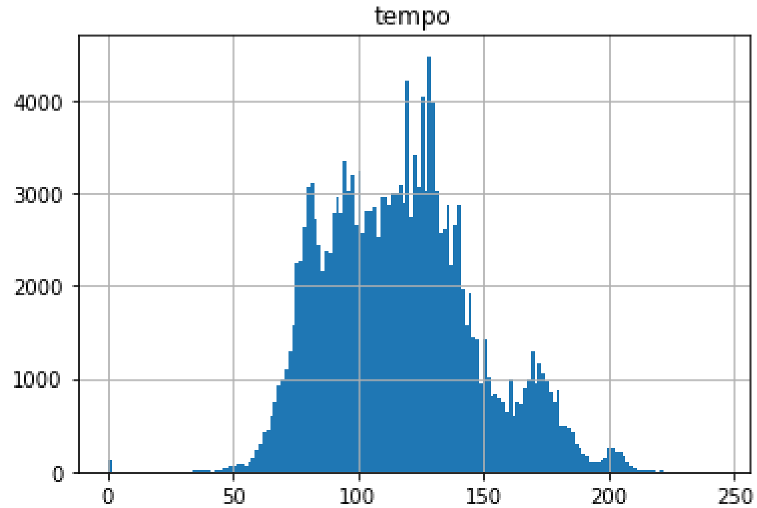

instrumentalness
100000
80000
60000
40000
20000
0
0.0
0.2
0.4
0.6
0.8
1.0

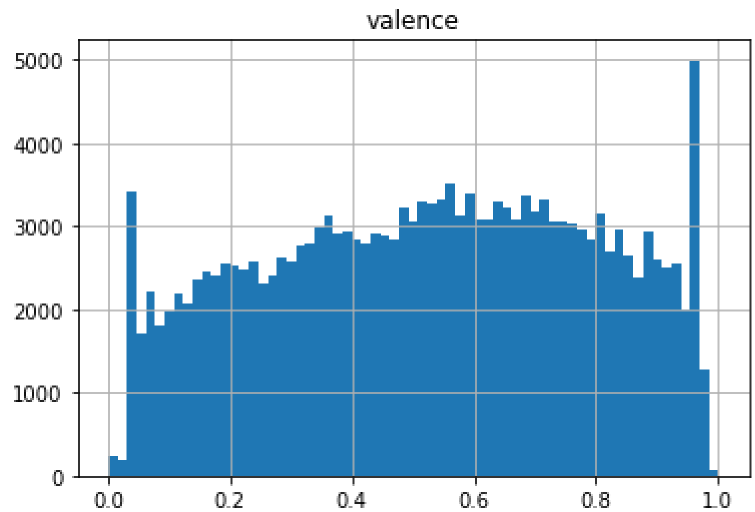

liveness
8000
6000
4000
2000
0
0.0
0.2
0.4
0.6
0.8
1.0

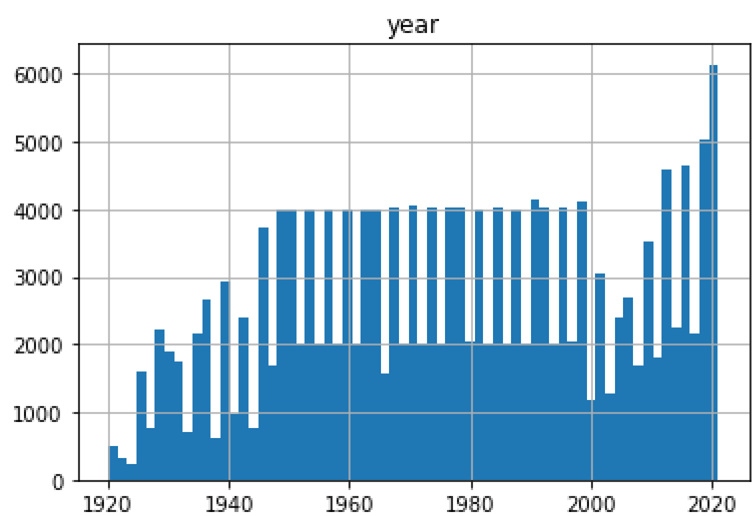

loudness
3500
3000
2500
2000
1500
1000
500
0
−60
−50
−40
−30
−20
−10
0


popularity
40000
30000
20000
10000
0
0
20
40
60
80
100


speechiness
8000
7000
6000
5000
4000
3000
2000
1000
0
0.0
0.2
0.4
0.6
0.8
1.0

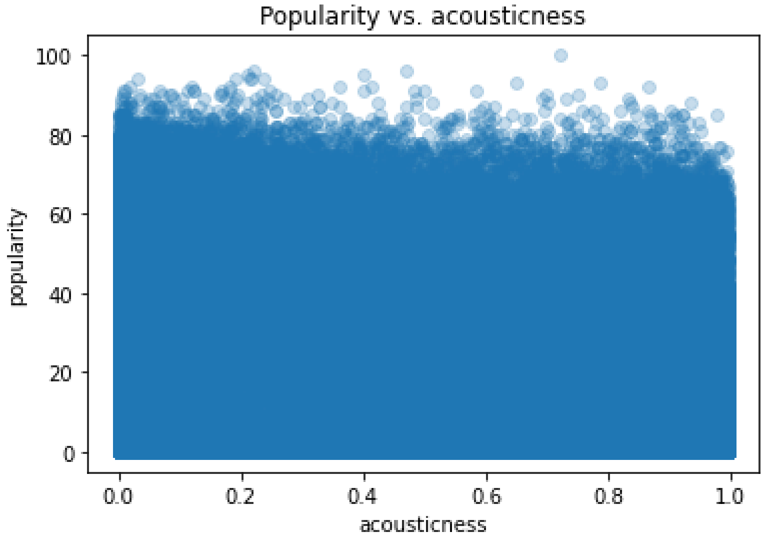


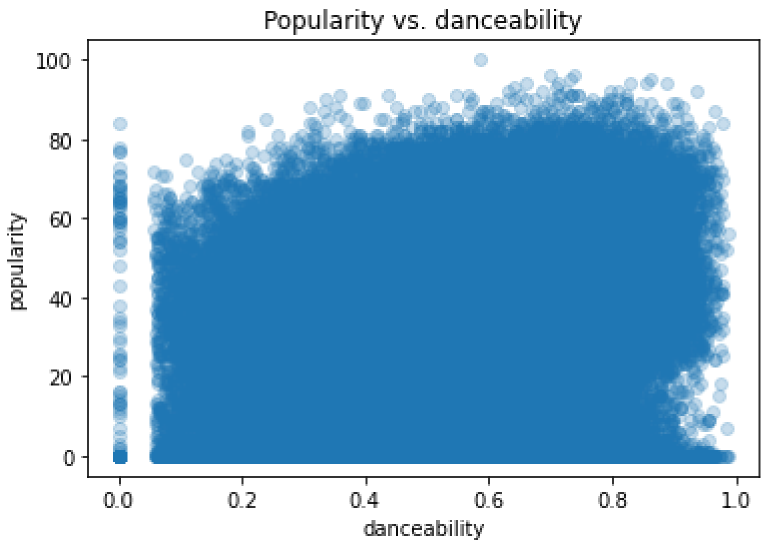


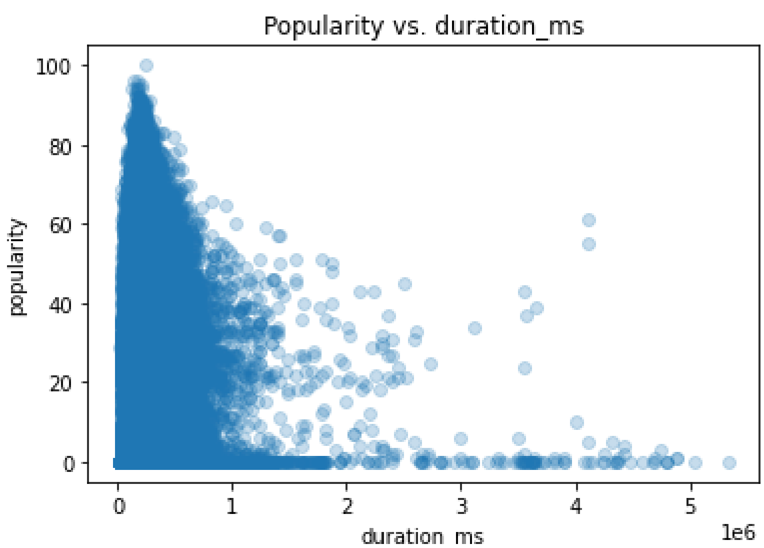


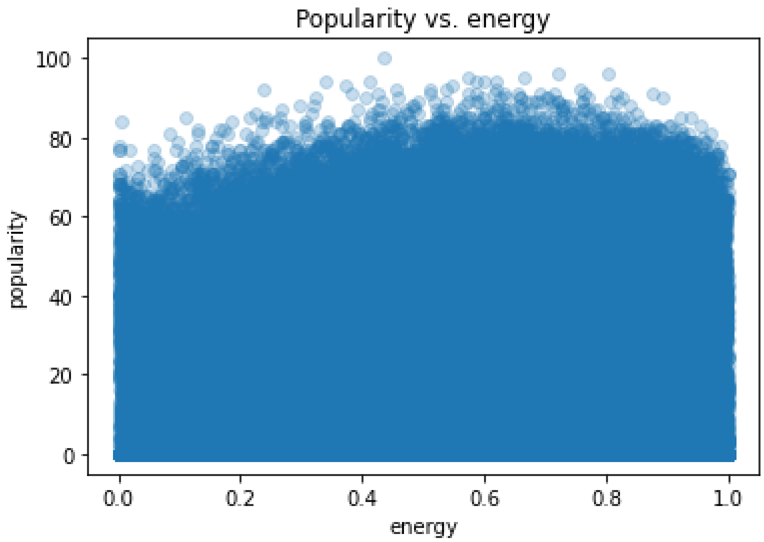
In [9]:

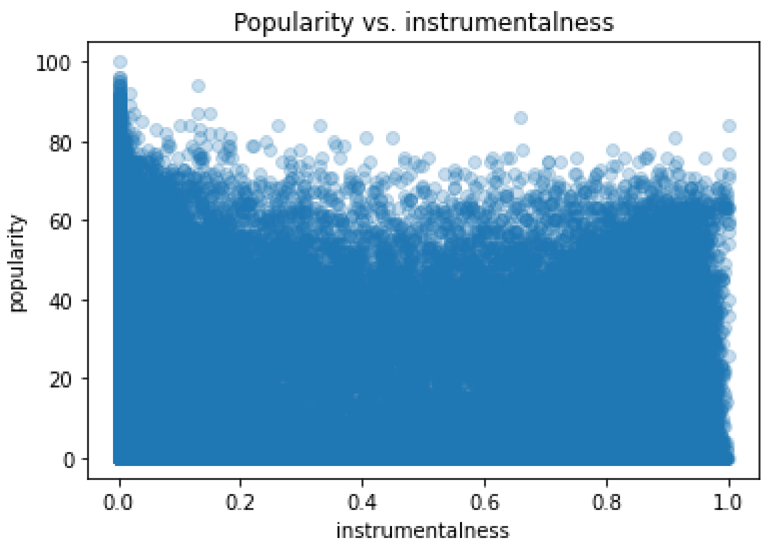
```
# view scatterplots of continuous variables vs. popularity
for column in cont_df.columns:
    plt.figure()
    plt.scatter(df[column], df.popularity, alpha=0.25)
    plt.title(f'Popularity vs. {column}')
    plt.xlabel(column)
    plt.ylabel('popularity')
```

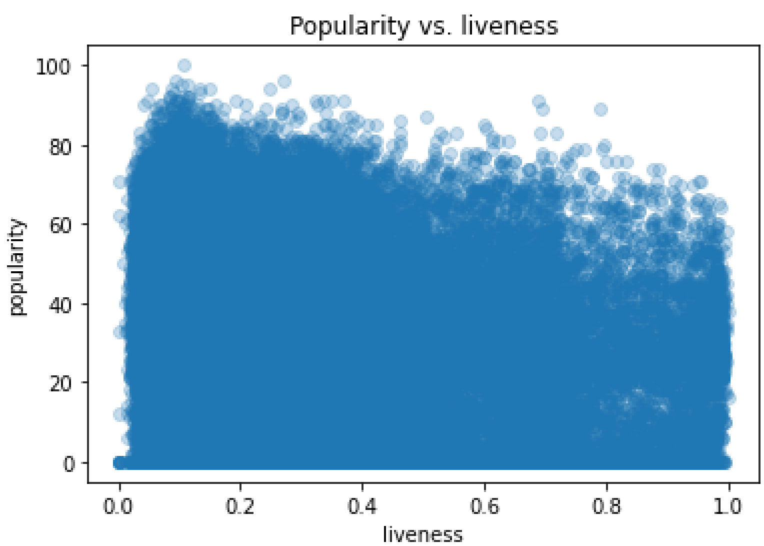

Popularity vs. acousticness
popularity
acousticness
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

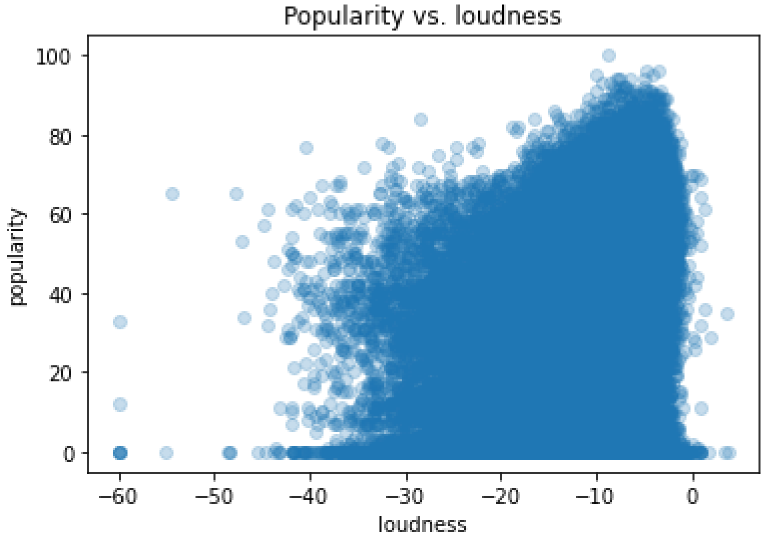

Popularity vs. danceability
popularity
danceability
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

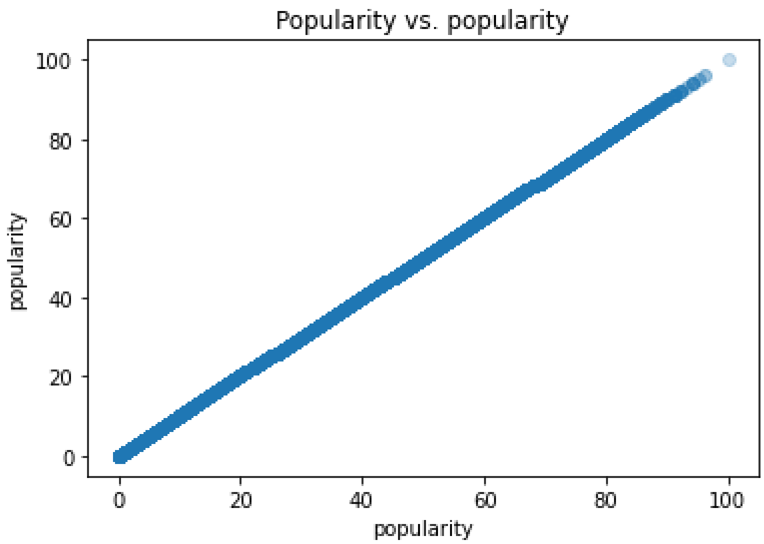

Popularity vs. duration_ms
popularity
duration_ms
1e6
0
20
40
60
80
100
0
1
2
3
4
5

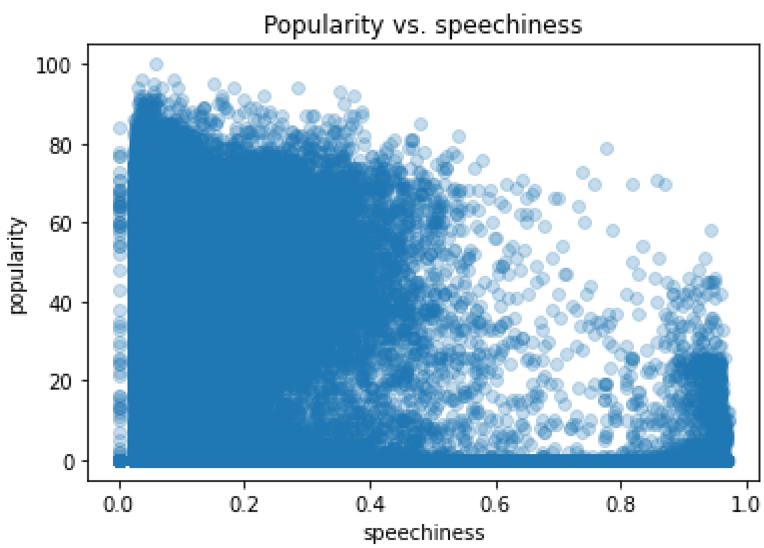

Popularity vs. energy
popularity
energy
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

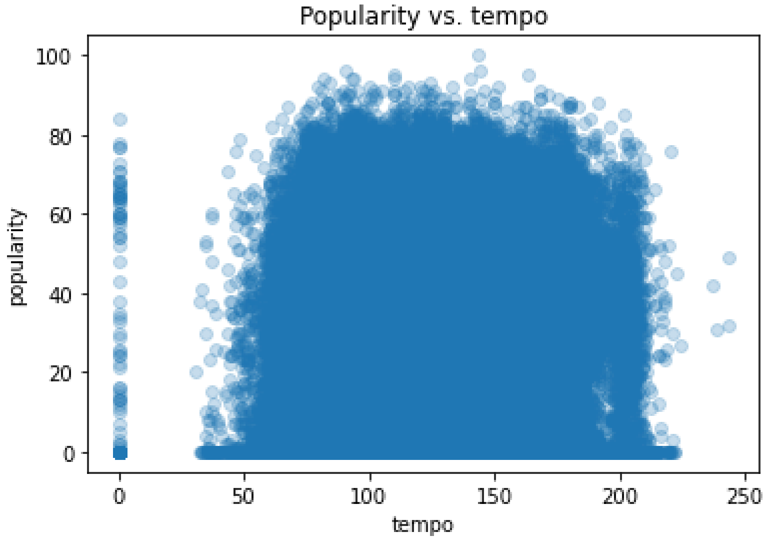

Popularity vs. instrumentalness
popularity
instrumentalness
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

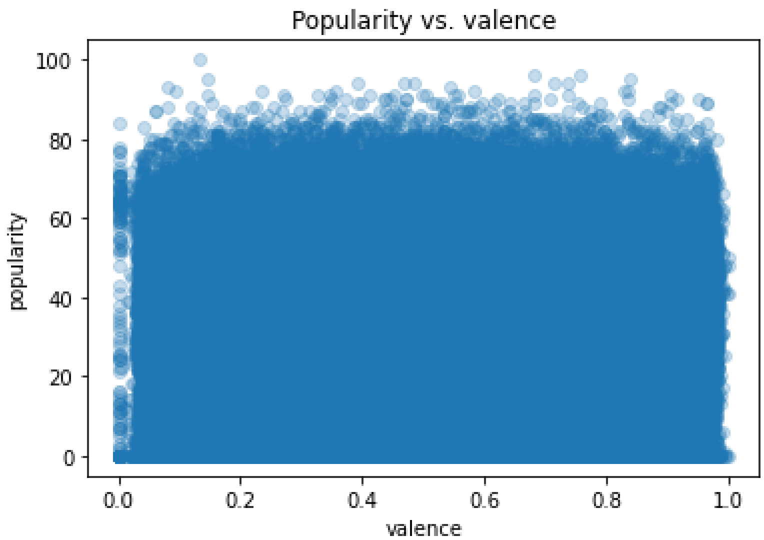

Popularity vs. liveness
popularity
liveness
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

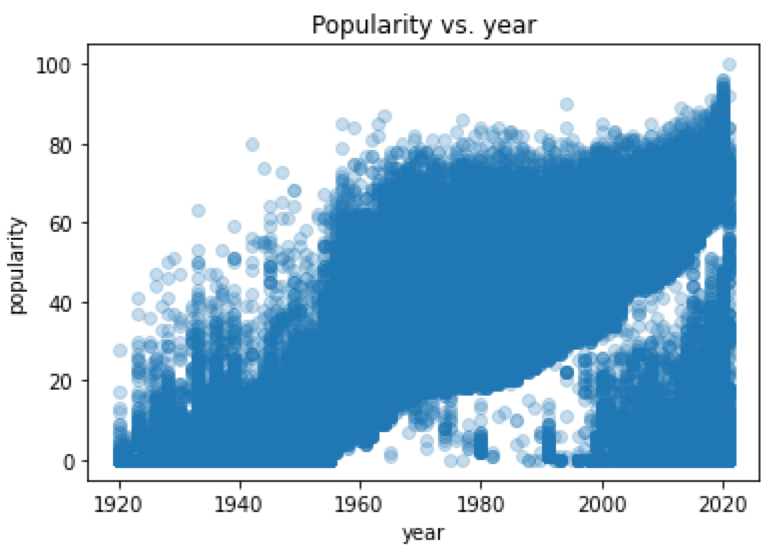

Popularity vs. loudness
popularity
loudness
100
80
60
40
20
0
−60
−50
−40
−30
−20
−10
0


Popularity vs. popularity
popularity
popularity
100
80
60
40
20
0
0
20
40
60
80
100


Popularity vs. speechiness
popularity
speechiness
100
80
60
40
20
0
0.0
0.2
0.4
0.6
0.8
1.0







In [10]:

```
# view value counts for categorical columns
for column in cat_columns:
    print(column, '\n')
    print(df[column].value_counts().head())
    print('------')
```

```
artists
```

```
['Tadeusz Dolega Mostowicz']    1281
['Эрнест Хемингуэй']           1175
['Эрих Мария Ремарк']          1062
['Francisco Canaro']            951
['Ignacio Corsini']             624
Name: artists, dtype: int64
------
explicit

0    162507
1     11882
Name: explicit, dtype: int64
------
key

0    21967
7    21363
2    18916
9    18109
5    16546
Name: key, dtype: int64
------
mode

1    122488
0     51901
Name: mode, dtype: int64
------
name

White Christmas      103
Winter Wonderland     88
Silent Night          81
Jingle Bells          71
2000 Years            70
Name: name, dtype: int64
------
```

After reviewing the continuous and categorical data, it can be seen that `instrumentalness` is defaulted to `0` for the vast majority of the data rows. The column does little to add detail and is therefore dropped.

Column `artists` is dropped to minimize the influence of "repeat hit artists" in the model. Similarly, `name` is dropped because the the most popular names of songs are all holiday oriented and that correlation may affect the data in unexpected ways.

In [11]:
```
# drop unnecessary columns
to_drop_2 = ['name', 'artists', 'instrumentalness']
df.drop(to_drop_2, axis=1, inplace=True)
```

`popularity` has a disproportionate ammount of `0` values, which hints at `0` being a default number for missing data. Those rows are removed.

There is no such thing as a song with a tempo of `0` , so the small handful of rows with that value are dropped.

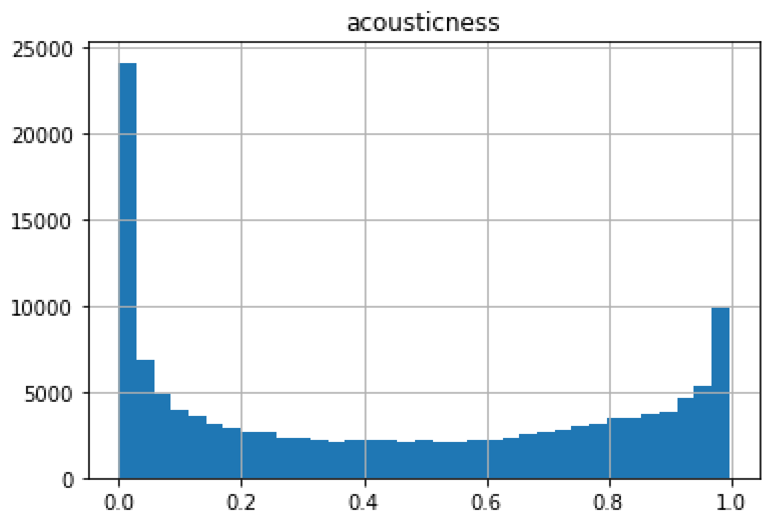
Lastly, the `duration_ms` outliers with songs longer than ~16 minutes are dropped.

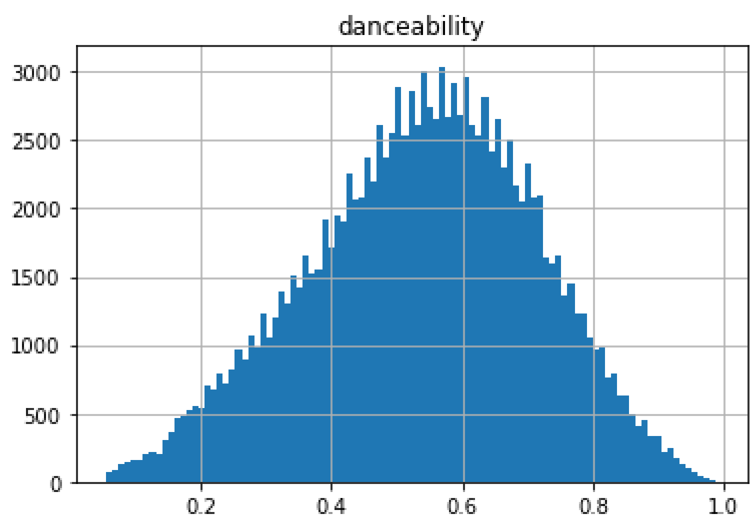
In [12]:
```
# drop outliers and default values
```

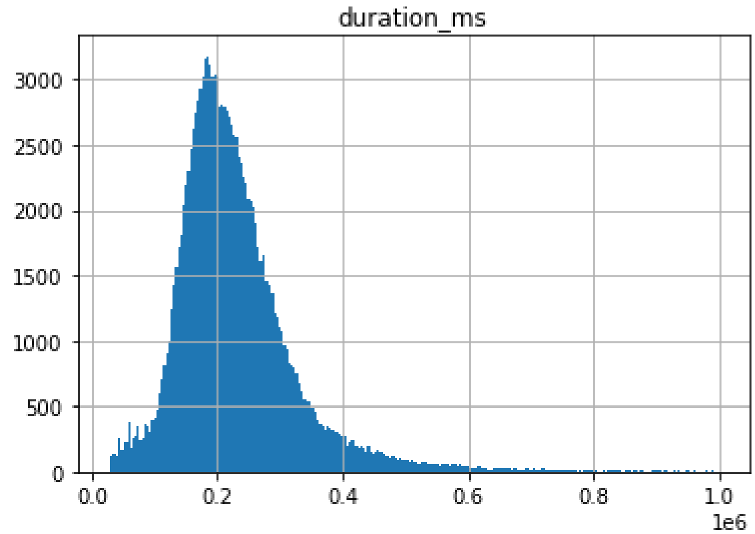
```
df.drop(df[df['popularity']==0].index, inplace=True)
df.drop(df[df['tempo']==0].index, inplace=True)
df.drop(df[df['duration_ms']>1000000].index, inplace=True)
```

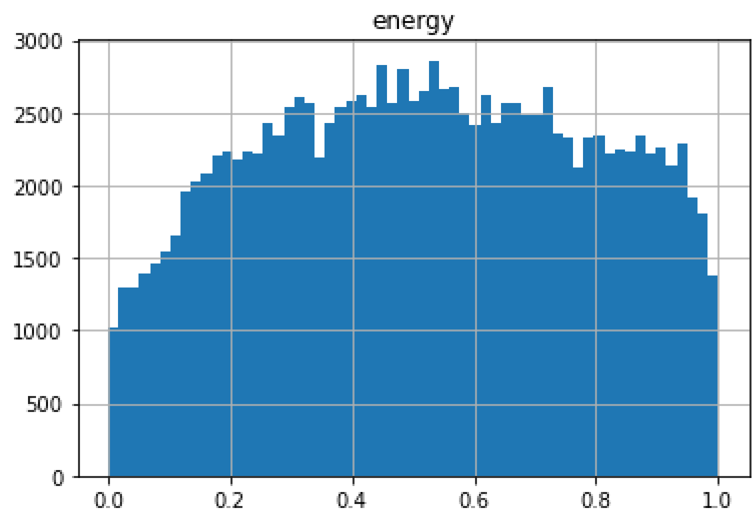
Histograms and scatterplots are rechecked.

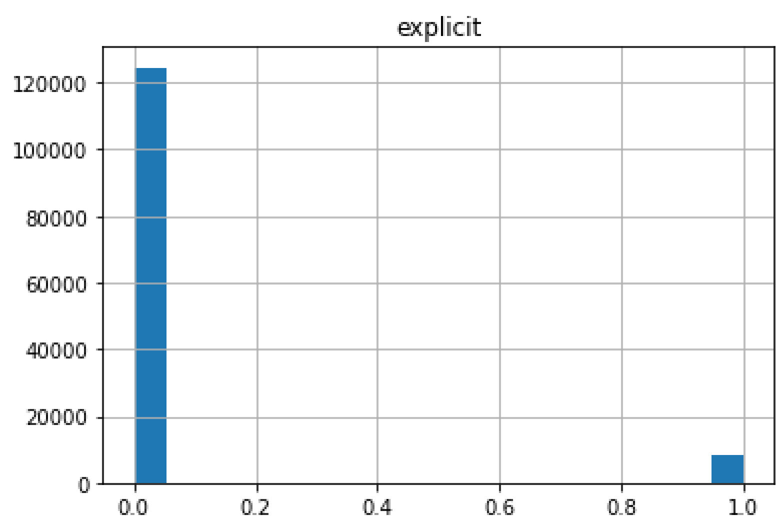
In [13]:
```
# view histograms of all columns
for column in df.columns:
    plt.figure()
    df[column].hist(bins='auto')
    plt.title(column)
```

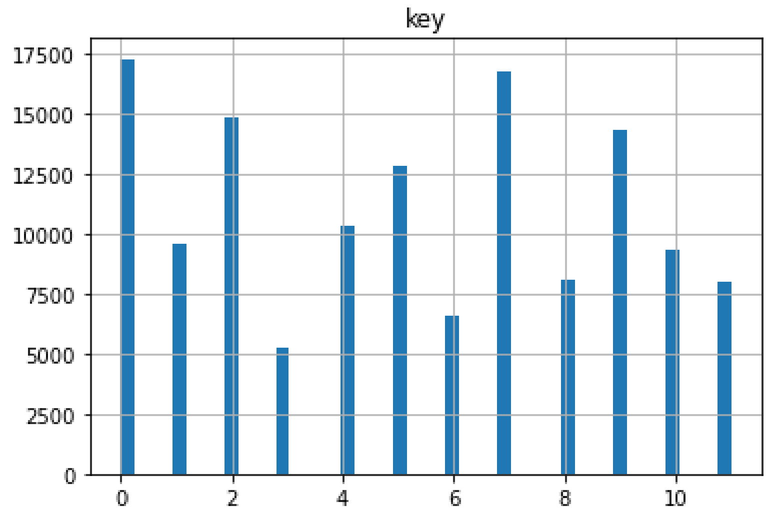


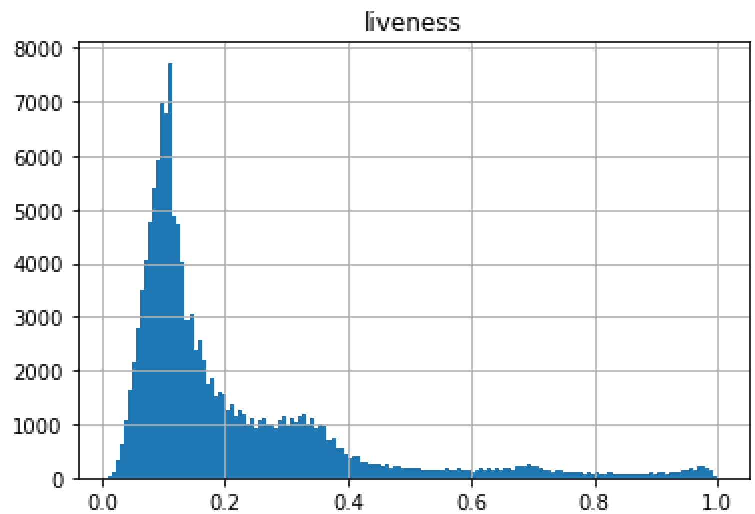


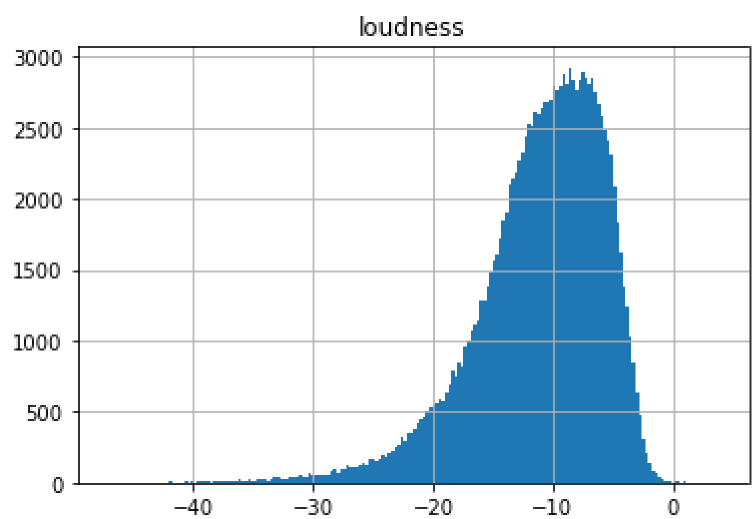

duration_ms
3000
2500
2000
1500
1000
500
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6

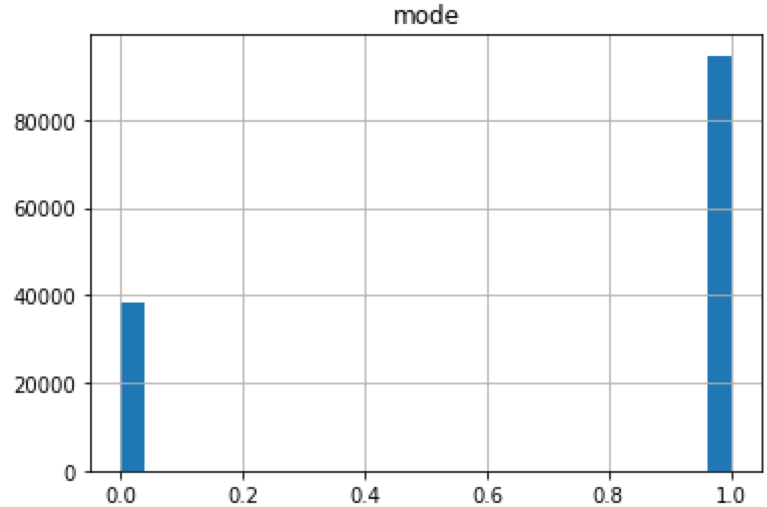

energy
3000
2500
2000
1500
1000
500
0
0.0
0.2
0.4
0.6
0.8
1.0

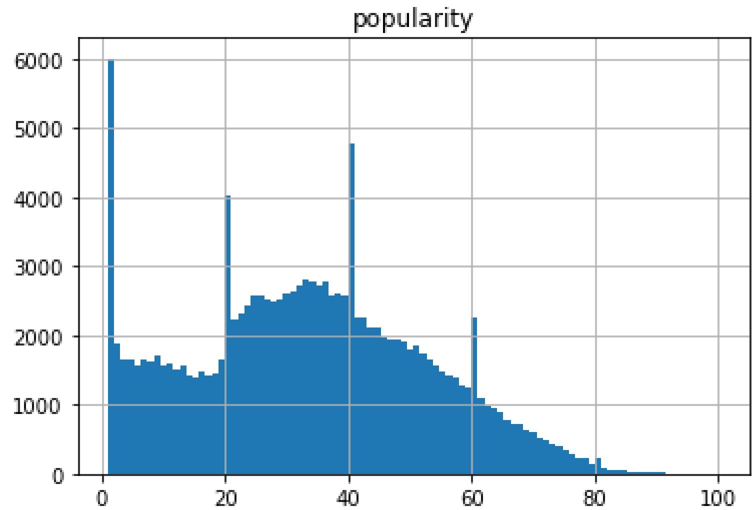

explicit
120000
100000
80000
60000
40000
20000
0
0.0
0.2
0.4
0.6
0.8
1.0

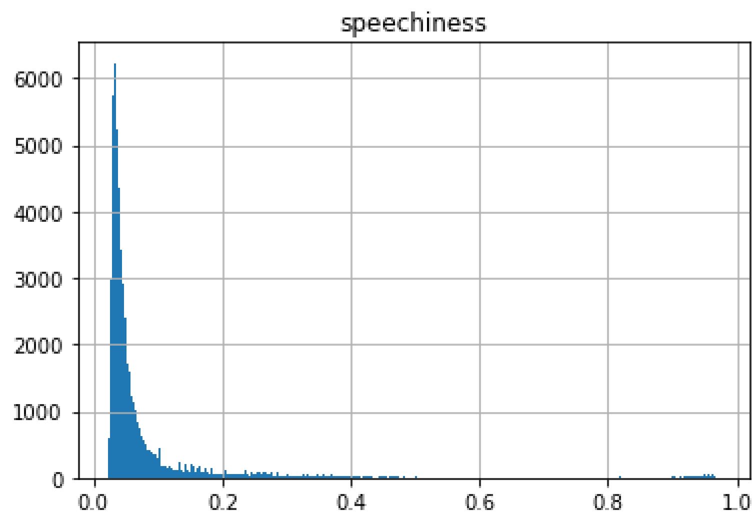

key
17500
15000
12500
10000
7500
5000
2500
0
0
2
4
6
8
10

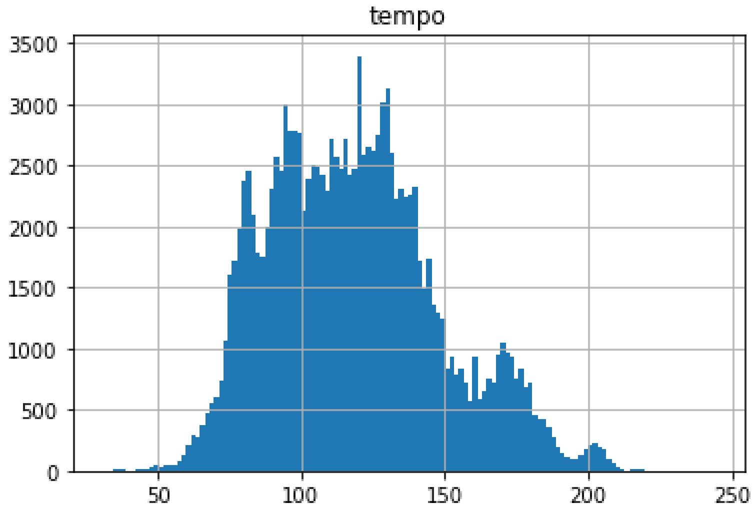

liveness
8000
7000
6000
5000
4000
3000
2000
1000
0
0.0
0.2
0.4
0.6
0.8
1.0

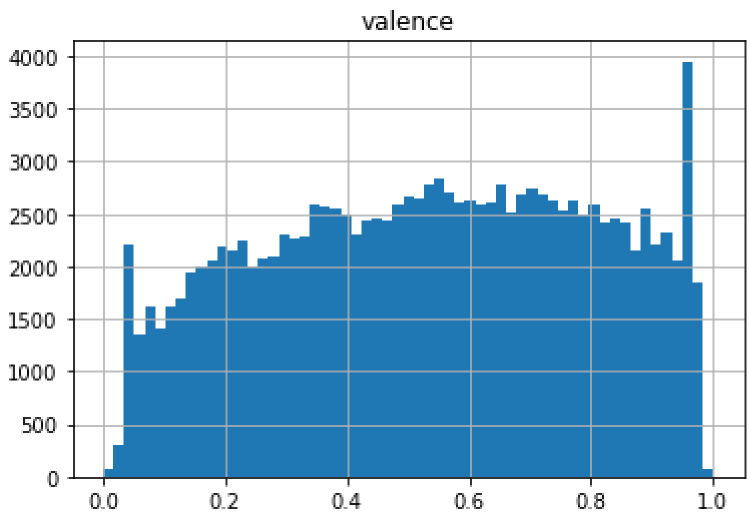

loudness
3000
2500
2000
1500
1000
500
0
−40
−30
−20
−10
0

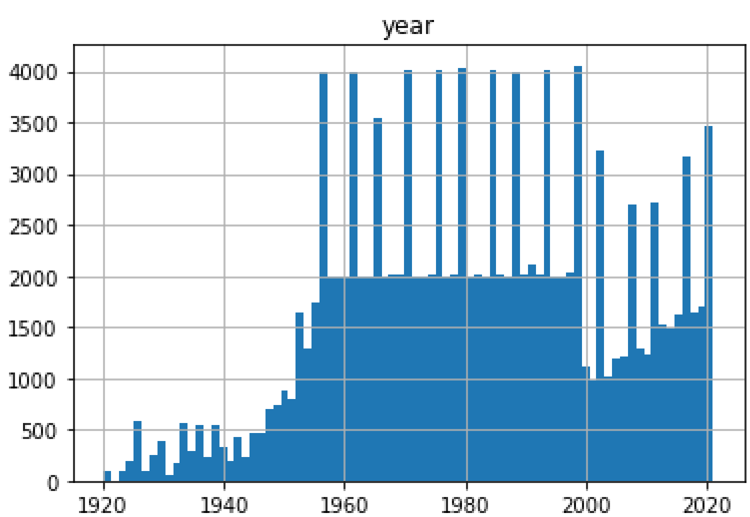

mode
0
20000
40000
60000
80000
0.0
0.2
0.4
0.6
0.8
1.0


popularity
0
1000
2000
3000
4000
5000
6000
0
20
40
60
80
100


speechiness
0
1000
2000
3000
4000
5000
6000
0.0
0.2
0.4
0.6
0.8
1.0

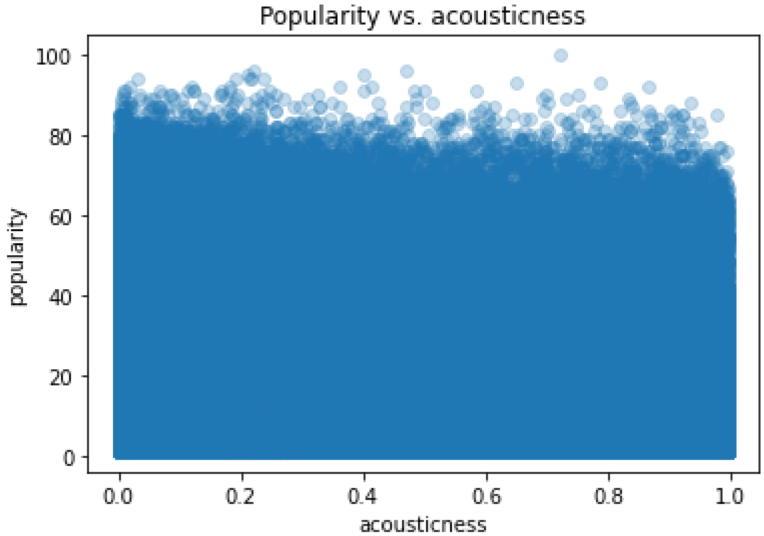


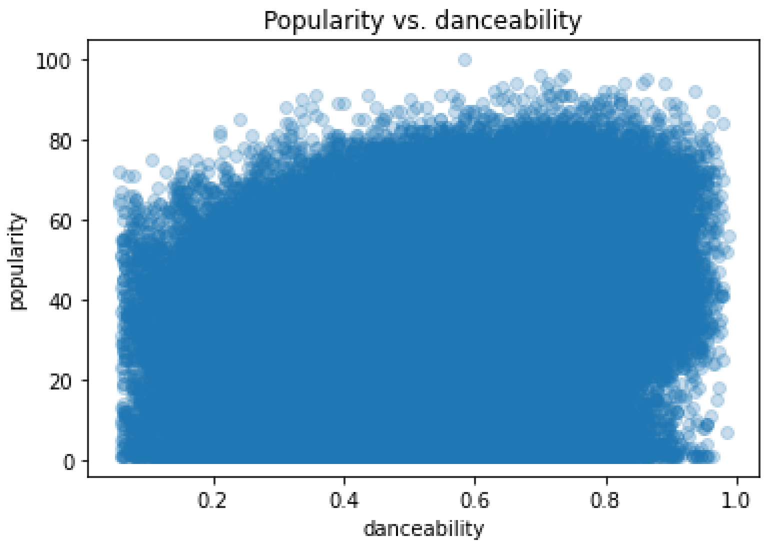


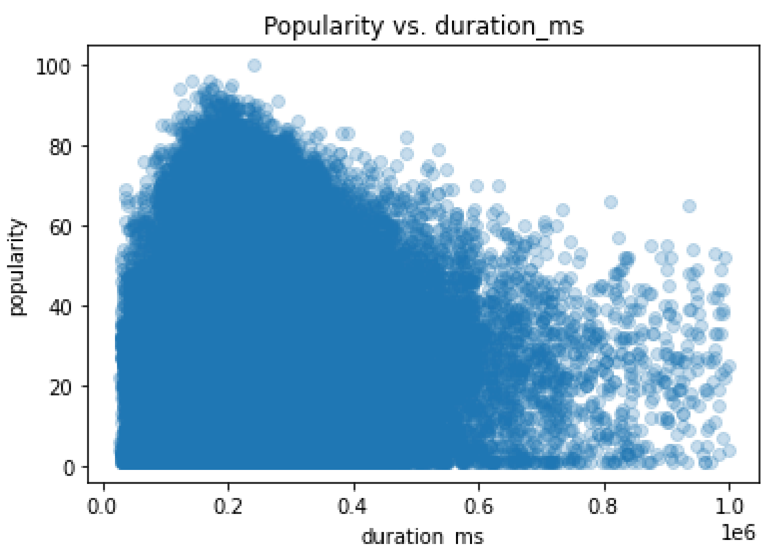


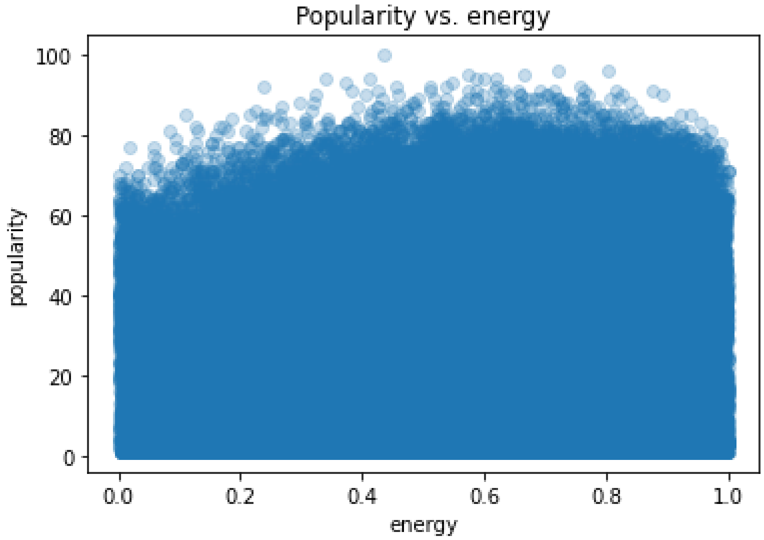
In [14]:

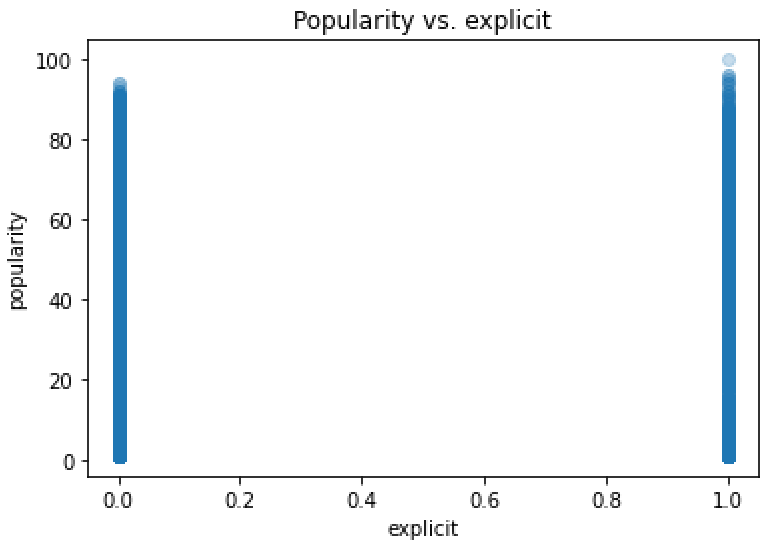
```
# view scatterplot of all columns vs. popularity
for column in df.columns:
    plt.figure()
    plt.scatter(df[column], df.popularity, alpha=0.25)
    plt.title(f'Popularity vs. {column}')
    plt.xlabel(column)
    plt.ylabel('popularity')
```

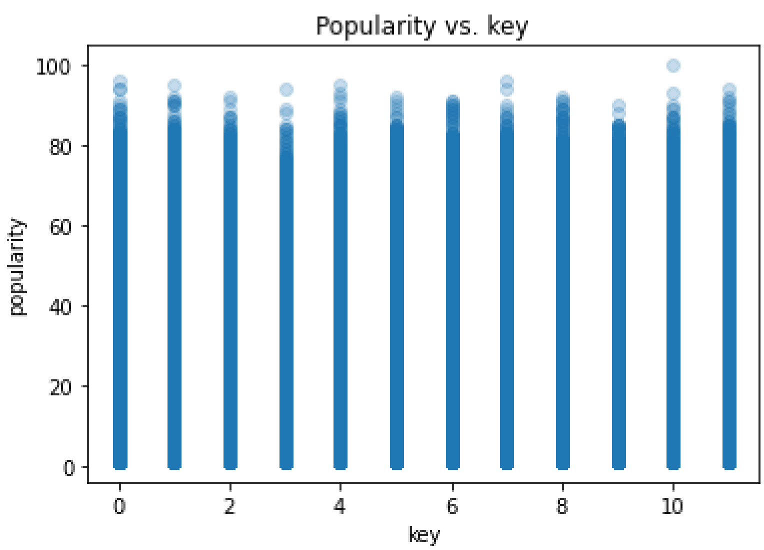

Popularity vs. acousticness
popularity
acousticness
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

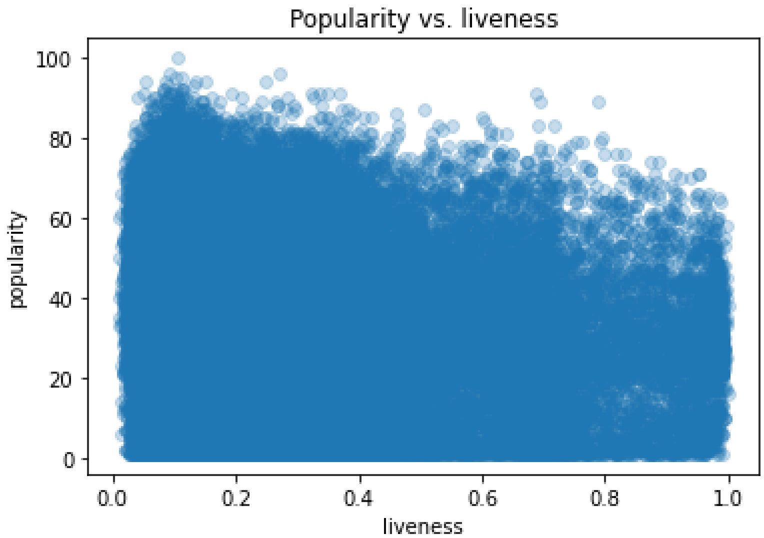

Popularity vs. danceability
popularity
danceability
0
20
40
60
80
100
0.2
0.4
0.6
0.8
1.0

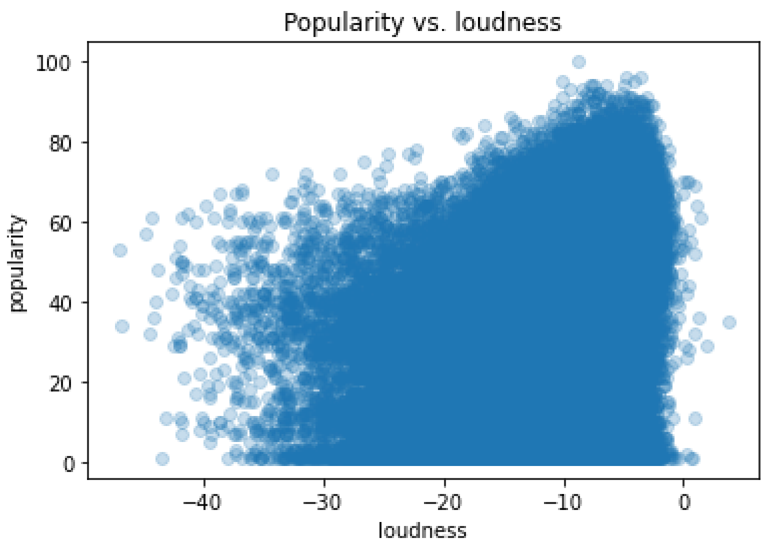

Popularity vs. duration_ms
popularity
duration_ms
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0
1e6

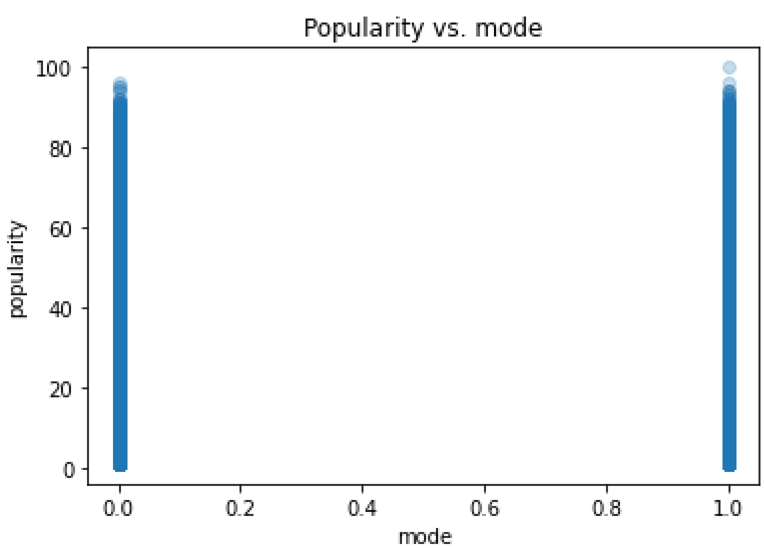

Popularity vs. energy
popularity
energy
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

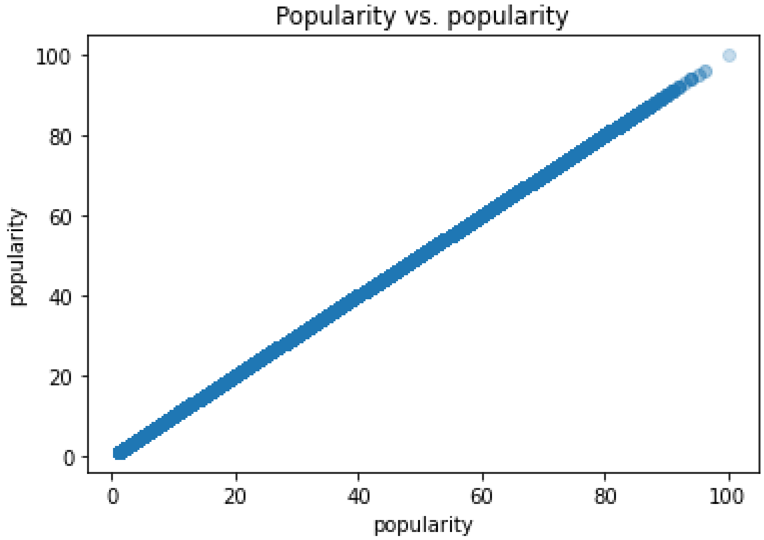

Popularity vs. explicit
popularity
explicit
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

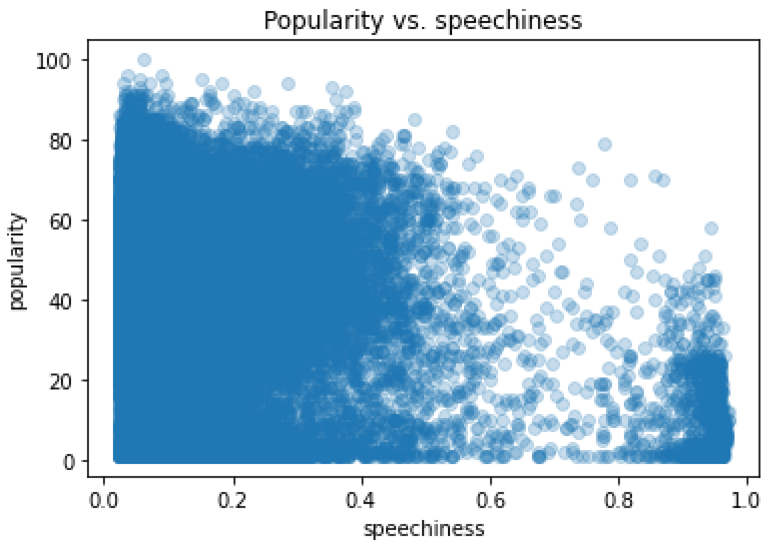

Popularity vs. key
popularity
key
0
20
40
60
80
100
0
2
4
6
8
10

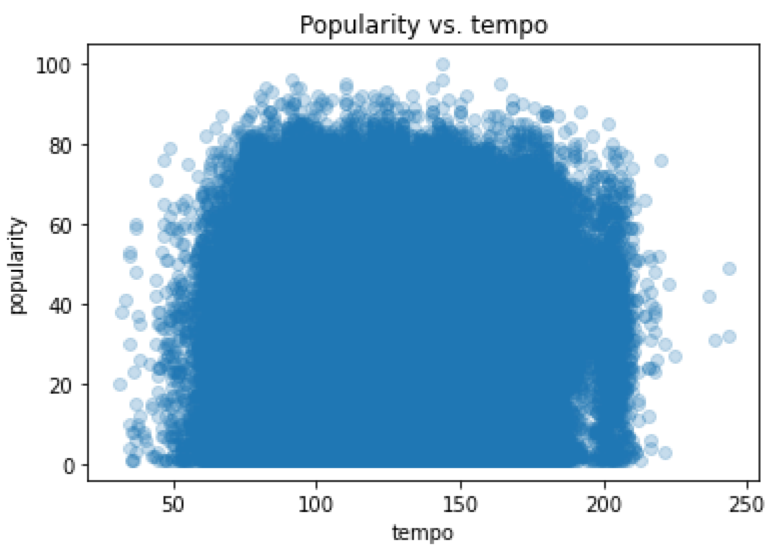

Popularity vs. liveness
popularity
liveness
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0

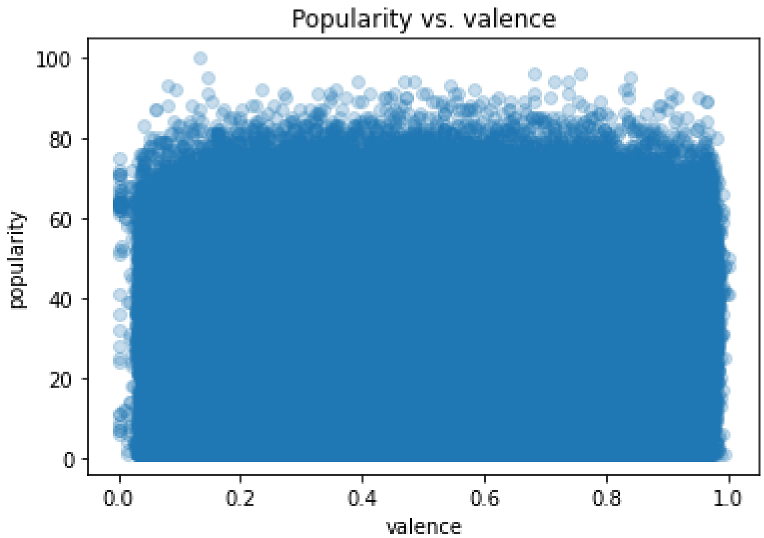

Popularity vs. loudness
popularity
loudness
0
20
40
60
80
100
−40
−30
−20
−10
0

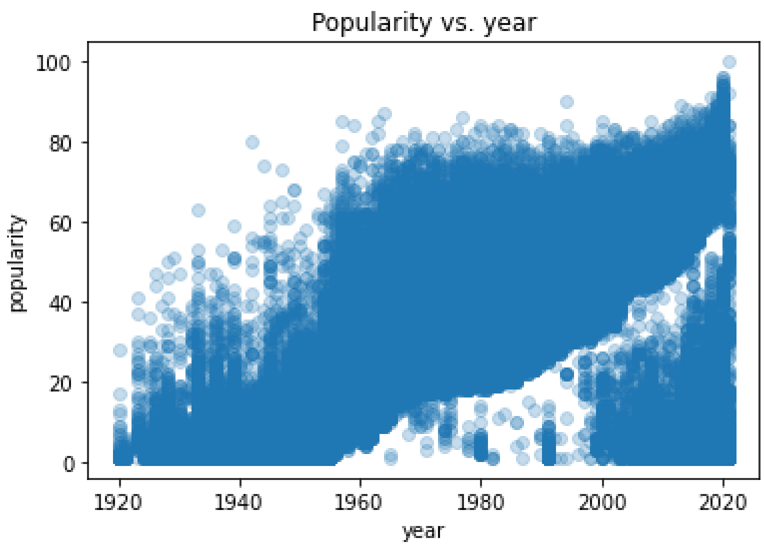

Popularity vs. mode
popularity
mode
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0


Popularity vs. popularity
popularity
popularity
0
20
40
60
80
100
0
20
40
60
80
100


Popularity vs. speechiness
popularity
speechiness
0
20
40
60
80
100
0.0
0.2
0.4
0.6
0.8
1.0


Popularity vs. tempo
popularity
tempo
0
20
40
60
80
100
50
100
150
200
250





In [15]:
```
# check final df shape
df.shape
```

Out[15]: (133091, 14)

In [16]:
```
# reset index and preview current dataframe
df.reset_index(inplace=True, drop=True)
df.head()
```

Out[16]:

	acousticness	danceability	duration_ms	energy	explicit	key	liveness	loudness	mode	pop
0	0.991000	0.598	168333	0.224	0	5	0.3790	-12.628	0	
1	0.643000	0.852	150200	0.517	0	5	0.0809	-7.261	0	
2	0.993000	0.647	163827	0.186	0	0	0.5190	-12.098	1	
3	0.000173	0.730	422087	0.798	0	2	0.1280	-7.311	1	
4	0.295000	0.704	165224	0.707	1	10	0.4020	-6.036	0	

Data Preprocessing

Before the data can be used to train and evaluate models, it must be split into training and test sets, the continuous variables must be normalized, and the the categorical variables must be one hot encoded.

one hot encoding

In [17]:
```
# rename key column and preview results
key_names = {0:'C', 1:'C#/Db', 2:'D', 3:'D#/Eb',
             4:'E', 5:'F', 6:'F#/Gb', 7:'G',
             8:'G#/Ab', 9:'A', 10:'A#/Bb', 11:'B'}
df['key'] = df['key'].map(lambda x: key_names[x])
df.head()
```

Out[17]:

	acousticness	danceability	duration_ms	energy	explicit	key	liveness	loudness	mode	p
0	0.991000	0.598	168333	0.224	0	F	0.3790	-12.628	0	
1	0.643000	0.852	150200	0.517	0	F	0.0809	-7.261	0	
2	0.993000	0.647	163827	0.186	0	C	0.5190	-12.098	1	
3	0.000173	0.730	422087	0.798	0	D	0.1280	-7.311	1	
4	0.295000	0.704	165224	0.707	1	A#/Bb	0.4020	-6.036	0	

In [18]:
```
# separate categorical columns for one hot encoding and create dummy variables
category_columns = ['explicit', 'key', 'mode']
category_df = pd.get_dummies(df[category_columns], drop_first=True)
```

In [19]:
```
# recombine one hot encoded variables with continuous variables
df.drop(category_columns, axis=1, inplace=True)
df = pd.concat([df, category_df], axis=1)
df.head()
```

Out[19]:

	acousticness	danceability	duration_ms	energy	liveness	loudness	popularity	speechiness
0	0.991000	0.598	168333	0.224	0.3790	-12.628	12	0.0936
1	0.643000	0.852	150200	0.517	0.0809	-7.261	7	0.0534
2	0.993000	0.647	163827	0.186	0.5190	-12.098	4	0.1740
3	0.000173	0.730	422087	0.798	0.1280	-7.311	17	0.0425
4	0.295000	0.704	165224	0.707	0.4020	-6.036	2	0.0768

5 rows × 24 columns

Train-Test Split

In [20]:
```
# import train test split to randomly separate data
from sklearn.model_selection import train_test_split
```

In [21]:
```
# designate independent variables X and target variable y
X = df.drop('popularity', axis=1)
y = df.popularity
```

In [22]:
```
# train test split: default test size of 0.25
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Standardization

In [23]:
```
# Import standard scaler
from sklearn.preprocessing import StandardScaler
```

In [24]:
```
# separate categorical and continuous columns so continuous variables can be sca
one_hot_columns = category_df.columns

# training data
X_train_cat = X_train[one_hot_columns].reset_index(drop=True)
X_train_cont = X_train.drop(one_hot_columns, axis=1).reset_index(drop=True)

# testing data
X_test_cat = X_test[one_hot_columns].reset_index(drop=True)
X_test_cont = X_test.drop(one_hot_columns, axis=1).reset_index(drop=True)
```

In [25]:
```
# fit-transform scaler to training data and transform testing data. convert to p
std = StandardScaler()
X_train_scaled = std.fit_transform(X_train_cont)
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X_train_cont.columns)

X_test_scaled = std.transform(X_test_cont)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_test_cont.columns)
```

In [26]:
```
# remerge scaled continuous variables with categorical variables
X_train = pd.concat([X_train_scaled, X_train_cat], axis=1)
X_test = pd.concat([X_test_scaled, X_test_cat], axis=1)
```

With the data preprocessed, it can now be used to train various machine learning models

Baseline Models

The machine learning models that will be auditioned include a K-nearest neighbor regressor, a decision tree regressor, a random forest regressor, an XGBoost regressor, and a deep neural network.

The models will be evaluated by their coefficient of determination and compared by their root mean square error and mean absolute error.

In [27]:
```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
```

In [28]:
```
# Define a function to deliver a report of model performance

def report_results(model, X_train=X_train, y_train=y_train, X_test=X_test, y_tes
    train_predict = model.predict(X_train)
    test_predict = model.predict(X_test)

    r2_train = r2_score(y_train, train_predict)
    r2_test = r2_score(y_test, test_predict)

    rmse_train = mean_squared_error(y_train, train_predict, squared=False)
    rmse_test = mean_squared_error(y_test, test_predict, squared=False)
```

```
    mae_train = mean_absolute_error(y_train, train_predict)
    mae_test = mean_absolute_error(y_test, test_predict)

    print('Training r2 Score: ', r2_train)
    print('Test r2 Score: ', r2_test)
    print('-----')
    print('Training RMSE: ', rmse_train)
    print('Test RMSE: ', rmse_test)
    print('-----')
    print('Training MAE: ', mae_train)
    print('Test MAE: ', mae_test)
```

K Nearest Neighbor Regressor

In [29]:
```
# import KNN regressor from scikit-learn
from sklearn.neighbors import KNeighborsRegressor
```

In [30]:
```
# instantiate and fit
knr = KNeighborsRegressor(n_jobs=-1)
knr.fit(X_train, y_train)
```

Out[30]:
```
KNeighborsRegressor(n_jobs=-1)
```

In [31]:
```
# report metrics
report_results(knr)
```

```
Training r2 Score:  0.6133699203313302
Test r2 Score:  0.42492630379979135
-----
Training RMSE:  11.815220445264224
Test RMSE:  14.366418213662186
-----
Training MAE:  8.788310725520446
Test MAE:  10.729348120097375
```

Decision Tree Regressor

In [32]:
```
# import decision tree regressor from scikit-learn
from sklearn.tree import DecisionTreeRegressor
```

In [33]:
```
# instantiate and fit
dtr = DecisionTreeRegressor()
dtr.fit(X_train, y_train)
```

Out[33]:
```
DecisionTreeRegressor()
```

In [34]:
```
# report metrics
report_results(dtr)
```

```
Training r2 Score:  0.9969169542638032
Test r2 Score:  0.1428599301152902
-----
Training RMSE:  1.0550766050607783
Test RMSE:  17.539315977279312
-----
Training MAE:  0.10979291834949133
Test MAE:  12.128304731564132
```

Random Forest Regressor

In [35]:
```
# import random forest regressor from scikit-learn
from sklearn.ensemble import RandomForestRegressor
```

In [36]:
```
# instantiate and fit
rfr = RandomForestRegressor(n_jobs=-1)
rfr.fit(X_train, y_train)
```

Out[36]:
```
RandomForestRegressor(n_jobs=-1)
```

In [37]:
```
# report metrics
report_results(rfr)
```

```
Training r2 Score:  0.9392857052171221
Test r2 Score:  0.5840939932445177
-----
Training RMSE:  4.682087083167327
Test RMSE:  12.217556312894363
-----
Training MAE:  3.382539228958031
Test MAE:  8.939604729396203
```

XGBoost Regressor

In [38]:
```
# import XGBRegressor from xgboost
from xgboost import XGBRegressor
```

In [39]:
```
# instantiate and fit
xgr = XGBRegressor(n_jobs=-1)
xgr.fit(X_train, y_train);
```

In [40]:
```
# report metrics
report_results(xgr)
```

```
Training r2 Score:  0.6457206119711785
Test r2 Score:  0.5689612299486131
-----
Training RMSE:  11.31011321936146
Test RMSE:  12.43783870307074
-----
Training MAE:  8.419818194785599
Test MAE:  9.144786148865695
```

Deep Neural Network

In [41]:
```
# import neural network packages from keras
from keras import models
from keras import layers
from keras import optimizers
```

In [42]:
```
# instantiate sequential neural network
model = models.Sequential()
```

In [43]:
```
# add layers
model.add(layers.Dense(23, activation='relu', input_shape=(23,)))
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='relu'))
```

In [44]:
```
# compile model
```

```
model.compile(optimizer='SGD', loss='mse', metrics=['mse'])
```

In [45]:

```
# fit model to data
model.fit(X_train, y_train, epochs=100, batch_size=500, validation_split=0.25)
```

```
Epoch 1/100
150/150 [==============================] - 0s 1ms/step - loss: 351.7784 - mse: 3
51.7784 - val_loss: 205.5085 - val_mse: 205.5085
Epoch 2/100
150/150 [==============================] - 0s 697us/step - loss: 210.9410 - mse:
210.9410 - val_loss: 192.2332 - val_mse: 192.2332
Epoch 3/100
150/150 [==============================] - 0s 693us/step - loss: 200.2261 - mse:
200.2261 - val_loss: 216.6682 - val_mse: 216.6682
Epoch 4/100
150/150 [==============================] - 0s 707us/step - loss: 195.6051 - mse:
195.6051 - val_loss: 187.3211 - val_mse: 187.3211
Epoch 5/100
150/150 [==============================] - 0s 708us/step - loss: 189.9759 - mse:
189.9759 - val_loss: 177.0479 - val_mse: 177.0479
Epoch 6/100
150/150 [==============================] - 0s 705us/step - loss: 187.4352 - mse:
187.4352 - val_loss: 176.7043 - val_mse: 176.7043
Epoch 7/100
150/150 [==============================] - 0s 765us/step - loss: 189.3237 - mse:
189.3237 - val_loss: 175.2073 - val_mse: 175.2073
Epoch 8/100
150/150 [==============================] - 0s 714us/step - loss: 186.5940 - mse:
186.5940 - val_loss: 180.1704 - val_mse: 180.1704
Epoch 9/100
150/150 [==============================] - 0s 703us/step - loss: 184.9141 - mse:
184.9141 - val_loss: 176.6951 - val_mse: 176.6951
Epoch 10/100
150/150 [==============================] - 0s 705us/step - loss: 181.5814 - mse:
181.5814 - val_loss: 173.3721 - val_mse: 173.3721
Epoch 11/100
150/150 [==============================] - 0s 700us/step - loss: 181.2936 - mse:
181.2936 - val_loss: 181.0435 - val_mse: 181.0435
Epoch 12/100
150/150 [==============================] - 0s 700us/step - loss: 183.2780 - mse:
183.2780 - val_loss: 176.4028 - val_mse: 176.4028
Epoch 13/100
150/150 [==============================] - 0s 709us/step - loss: 180.8637 - mse:
180.8637 - val_loss: 184.8562 - val_mse: 184.8562
Epoch 14/100
150/150 [==============================] - 0s 748us/step - loss: 180.6626 - mse:
180.6626 - val_loss: 181.0325 - val_mse: 181.0325
Epoch 15/100
150/150 [==============================] - 0s 694us/step - loss: 180.8714 - mse:
180.8714 - val_loss: 170.8794 - val_mse: 170.8794
Epoch 16/100
150/150 [==============================] - 0s 705us/step - loss: 178.6671 - mse:
178.6671 - val_loss: 175.0166 - val_mse: 175.0166
Epoch 17/100
150/150 [==============================] - 0s 698us/step - loss: 177.6663 - mse:
177.6663 - val_loss: 169.6962 - val_mse: 169.6962
Epoch 18/100
150/150 [==============================] - 0s 695us/step - loss: 178.3903 - mse:
178.3903 - val_loss: 177.3613 - val_mse: 177.3613
Epoch 19/100
150/150 [==============================] - 0s 716us/step - loss: 178.4150 - mse:
178.4150 - val_loss: 174.7081 - val_mse: 174.7081
Epoch 20/100
150/150 [==============================] - 0s 728us/step - loss: 176.6471 - mse:
```

```
176.6471 - val_loss: 178.4704 - val_mse: 178.4704
Epoch 21/100
150/150 [==============================] - 0s 709us/step - loss: 177.3613 - mse:
177.3613 - val_loss: 170.6755 - val_mse: 170.6755
Epoch 22/100
150/150 [==============================] - 0s 720us/step - loss: 176.2901 - mse:
176.2901 - val_loss: 195.0643 - val_mse: 195.0643
Epoch 23/100
150/150 [==============================] - 0s 698us/step - loss: 176.5429 - mse:
176.5429 - val_loss: 168.5570 - val_mse: 168.5570
Epoch 24/100
150/150 [==============================] - 0s 753us/step - loss: 176.0460 - mse:
176.0460 - val_loss: 176.4481 - val_mse: 176.4481
Epoch 25/100
150/150 [==============================] - 0s 697us/step - loss: 175.1725 - mse:
175.1725 - val_loss: 170.8129 - val_mse: 170.8129
Epoch 26/100
150/150 [==============================] - 0s 705us/step - loss: 176.0723 - mse:
176.0723 - val_loss: 172.8664 - val_mse: 172.8664
Epoch 27/100
150/150 [==============================] - 0s 761us/step - loss: 176.0016 - mse:
176.0016 - val_loss: 168.5790 - val_mse: 168.5790
Epoch 28/100
150/150 [==============================] - 0s 760us/step - loss: 174.9931 - mse:
174.9931 - val_loss: 175.8652 - val_mse: 175.8652
Epoch 29/100
150/150 [==============================] - 0s 696us/step - loss: 174.4761 - mse:
174.4761 - val_loss: 177.1452 - val_mse: 177.1452
Epoch 30/100
150/150 [==============================] - 0s 759us/step - loss: 174.7161 - mse:
174.7161 - val_loss: 173.4108 - val_mse: 173.4108
Epoch 31/100
150/150 [==============================] - 0s 687us/step - loss: 174.0348 - mse:
174.0348 - val_loss: 174.6047 - val_mse: 174.6047
Epoch 32/100
150/150 [==============================] - 0s 703us/step - loss: 174.4077 - mse:
174.4077 - val_loss: 171.6731 - val_mse: 171.6731
Epoch 33/100
150/150 [==============================] - 0s 696us/step - loss: 173.6620 - mse:
173.6620 - val_loss: 172.3681 - val_mse: 172.3681
Epoch 34/100
150/150 [==============================] - 0s 700us/step - loss: 172.5722 - mse:
172.5722 - val_loss: 181.2851 - val_mse: 181.2851
Epoch 35/100
150/150 [==============================] - 0s 748us/step - loss: 174.2926 - mse:
174.2926 - val_loss: 169.0229 - val_mse: 169.0229
Epoch 36/100
150/150 [==============================] - 0s 727us/step - loss: 172.2981 - mse:
172.2981 - val_loss: 174.7924 - val_mse: 174.7924
Epoch 37/100
150/150 [==============================] - 0s 683us/step - loss: 171.8882 - mse:
171.8882 - val_loss: 168.5689 - val_mse: 168.5689
Epoch 38/100
150/150 [==============================] - 0s 691us/step - loss: 172.2738 - mse:
172.2738 - val_loss: 167.6102 - val_mse: 167.6102
Epoch 39/100
150/150 [==============================] - 0s 691us/step - loss: 172.8855 - mse:
172.8855 - val_loss: 166.8164 - val_mse: 166.8164
Epoch 40/100
150/150 [==============================] - 0s 691us/step - loss: 172.3942 - mse:
172.3942 - val_loss: 170.0745 - val_mse: 170.0745
Epoch 41/100
150/150 [==============================] - 0s 678us/step - loss: 171.7955 - mse:
171.7955 - val_loss: 170.1444 - val_mse: 170.1444
Epoch 42/100
```

```
150/150 [==============================] - 0s 697us/step - loss: 171.3978 - mse:
171.3978 - val_loss: 172.7475 - val_mse: 172.7475
Epoch 43/100
150/150 [==============================] - 0s 677us/step - loss: 170.7294 - mse:
170.7294 - val_loss: 167.5429 - val_mse: 167.5429
Epoch 44/100
150/150 [==============================] - 0s 684us/step - loss: 170.9226 - mse:
170.9226 - val_loss: 168.3205 - val_mse: 168.3205
Epoch 45/100
150/150 [==============================] - 0s 697us/step - loss: 171.0493 - mse:
171.0493 - val_loss: 169.9691 - val_mse: 169.9691
Epoch 46/100
150/150 [==============================] - 0s 687us/step - loss: 170.3817 - mse:
170.3817 - val_loss: 169.3290 - val_mse: 169.3290
Epoch 47/100
150/150 [==============================] - 0s 727us/step - loss: 170.8854 - mse:
170.8854 - val_loss: 166.7510 - val_mse: 166.7510
Epoch 48/100
150/150 [==============================] - 0s 692us/step - loss: 171.1601 - mse:
171.1601 - val_loss: 169.0878 - val_mse: 169.0878
Epoch 49/100
150/150 [==============================] - 0s 679us/step - loss: 169.3412 - mse:
169.3412 - val_loss: 165.8577 - val_mse: 165.8577
Epoch 50/100
150/150 [==============================] - 0s 677us/step - loss: 169.7985 - mse:
169.7985 - val_loss: 168.1721 - val_mse: 168.1721
Epoch 51/100
150/150 [==============================] - 0s 683us/step - loss: 169.2808 - mse:
169.2808 - val_loss: 167.6420 - val_mse: 167.6420
Epoch 52/100
150/150 [==============================] - 0s 683us/step - loss: 168.9416 - mse:
168.9416 - val_loss: 167.7817 - val_mse: 167.7817
Epoch 53/100
150/150 [==============================] - 0s 689us/step - loss: 168.4018 - mse:
168.4018 - val_loss: 167.1266 - val_mse: 167.1266
Epoch 54/100
150/150 [==============================] - 0s 698us/step - loss: 169.8158 - mse:
169.8158 - val_loss: 167.7739 - val_mse: 167.7739
Epoch 55/100
150/150 [==============================] - 0s 693us/step - loss: 169.5866 - mse:
169.5866 - val_loss: 168.9229 - val_mse: 168.9229
Epoch 56/100
150/150 [==============================] - 0s 702us/step - loss: 169.0457 - mse:
169.0457 - val_loss: 169.7269 - val_mse: 169.7269
Epoch 57/100
150/150 [==============================] - 0s 684us/step - loss: 168.2484 - mse:
168.2484 - val_loss: 167.1270 - val_mse: 167.1270
Epoch 58/100
150/150 [==============================] - 0s 698us/step - loss: 167.7539 - mse:
167.7539 - val_loss: 177.0686 - val_mse: 177.0686
Epoch 59/100
150/150 [==============================] - 0s 676us/step - loss: 168.0561 - mse:
168.0561 - val_loss: 176.6508 - val_mse: 176.6508
Epoch 60/100
150/150 [==============================] - 0s 693us/step - loss: 168.4480 - mse:
168.4480 - val_loss: 173.9220 - val_mse: 173.9220
Epoch 61/100
150/150 [==============================] - 0s 674us/step - loss: 167.7372 - mse:
167.7372 - val_loss: 172.8198 - val_mse: 172.8198
Epoch 62/100
150/150 [==============================] - 0s 701us/step - loss: 167.5704 - mse:
167.5704 - val_loss: 169.5775 - val_mse: 169.5775
Epoch 63/100
150/150 [==============================] - 0s 683us/step - loss: 167.6003 - mse:
167.6003 - val_loss: 174.4690 - val_mse: 174.4690
```

```
Epoch 64/100
150/150 [==============================] - 0s 688us/step - loss: 167.4617 - mse: 167.4617 - val_loss: 167.4082 - val_mse: 167.4082
Epoch 65/100
150/150 [==============================] - 0s 689us/step - loss: 167.4471 - mse: 167.4471 - val_loss: 171.0927 - val_mse: 171.0927
Epoch 66/100
150/150 [==============================] - 0s 689us/step - loss: 167.4270 - mse: 167.4270 - val_loss: 167.6852 - val_mse: 167.6852
Epoch 67/100
150/150 [==============================] - 0s 676us/step - loss: 167.2720 - mse: 167.2720 - val_loss: 170.1932 - val_mse: 170.1932
Epoch 68/100
150/150 [==============================] - 0s 675us/step - loss: 167.1749 - mse: 167.1749 - val_loss: 167.3627 - val_mse: 167.3627
Epoch 69/100
150/150 [==============================] - 0s 682us/step - loss: 166.5287 - mse: 166.5287 - val_loss: 166.1725 - val_mse: 166.1725
Epoch 70/100
150/150 [==============================] - 0s 675us/step - loss: 167.4212 - mse: 167.4212 - val_loss: 165.5488 - val_mse: 165.5488
Epoch 71/100
150/150 [==============================] - 0s 674us/step - loss: 167.3370 - mse: 167.3370 - val_loss: 166.7239 - val_mse: 166.7239
Epoch 72/100
150/150 [==============================] - 0s 680us/step - loss: 166.9459 - mse: 166.9459 - val_loss: 168.3331 - val_mse: 168.3331
Epoch 73/100
150/150 [==============================] - 0s 683us/step - loss: 166.7694 - mse: 166.7694 - val_loss: 168.6425 - val_mse: 168.6425
Epoch 74/100
150/150 [==============================] - 0s 709us/step - loss: 167.3252 - mse: 167.3252 - val_loss: 169.4808 - val_mse: 169.4808
Epoch 75/100
150/150 [==============================] - 0s 693us/step - loss: 165.7251 - mse: 165.7251 - val_loss: 168.8421 - val_mse: 168.8421
Epoch 76/100
150/150 [==============================] - 0s 685us/step - loss: 167.1560 - mse: 167.1560 - val_loss: 168.2562 - val_mse: 168.2562
Epoch 77/100
150/150 [==============================] - 0s 675us/step - loss: 166.9107 - mse: 166.9107 - val_loss: 169.2257 - val_mse: 169.2257
Epoch 78/100
150/150 [==============================] - 0s 687us/step - loss: 165.9405 - mse: 165.9405 - val_loss: 168.1043 - val_mse: 168.1043
Epoch 79/100
150/150 [==============================] - 0s 680us/step - loss: 165.7083 - mse: 165.7083 - val_loss: 166.9538 - val_mse: 166.9538
Epoch 80/100
150/150 [==============================] - 0s 684us/step - loss: 165.9134 - mse: 165.9134 - val_loss: 166.4990 - val_mse: 166.4990
Epoch 81/100
150/150 [==============================] - 0s 684us/step - loss: 166.2901 - mse: 166.2901 - val_loss: 170.1022 - val_mse: 170.1022
Epoch 82/100
150/150 [==============================] - 0s 691us/step - loss: 165.6559 - mse: 165.6559 - val_loss: 167.3288 - val_mse: 167.3288
Epoch 83/100
150/150 [==============================] - 0s 684us/step - loss: 166.7839 - mse: 166.7839 - val_loss: 176.6037 - val_mse: 176.6037
Epoch 84/100
150/150 [==============================] - 0s 690us/step - loss: 166.2729 - mse: 166.2729 - val_loss: 167.5784 - val_mse: 167.5784
Epoch 85/100
150/150 [==============================] - 0s 690us/step - loss: 166.3580 - mse:
```

```
166.3580 - val_loss: 166.9449 - val_mse: 166.9449
Epoch 86/100
150/150 [==============================] - 0s 689us/step - loss: 165.8499 - mse:
165.8499 - val_loss: 167.1544 - val_mse: 167.1544
Epoch 87/100
150/150 [==============================] - 0s 690us/step - loss: 165.7136 - mse:
165.7136 - val_loss: 168.6930 - val_mse: 168.6930
Epoch 88/100
150/150 [==============================] - 0s 683us/step - loss: 166.1711 - mse:
166.1711 - val_loss: 169.9350 - val_mse: 169.9350
Epoch 89/100
150/150 [==============================] - 0s 690us/step - loss: 165.3042 - mse:
165.3042 - val_loss: 169.5502 - val_mse: 169.5502
Epoch 90/100
150/150 [==============================] - 0s 678us/step - loss: 165.2449 - mse:
165.2449 - val_loss: 168.7765 - val_mse: 168.7765
Epoch 91/100
150/150 [==============================] - 0s 681us/step - loss: 166.0034 - mse:
166.0034 - val_loss: 166.4844 - val_mse: 166.4844
Epoch 92/100
150/150 [==============================] - 0s 693us/step - loss: 165.7995 - mse:
165.7995 - val_loss: 165.6263 - val_mse: 165.6263
Epoch 93/100
150/150 [==============================] - 0s 679us/step - loss: 165.5230 - mse:
165.5230 - val_loss: 164.8560 - val_mse: 164.8560
Epoch 94/100
150/150 [==============================] - 0s 685us/step - loss: 165.7806 - mse:
165.7806 - val_loss: 168.4503 - val_mse: 168.4503
Epoch 95/100
150/150 [==============================] - 0s 686us/step - loss: 165.1403 - mse:
165.1403 - val_loss: 167.8744 - val_mse: 167.8744
Epoch 96/100
150/150 [==============================] - 0s 695us/step - loss: 165.0028 - mse:
165.0028 - val_loss: 172.2683 - val_mse: 172.2683
Epoch 97/100
150/150 [==============================] - 0s 684us/step - loss: 165.2846 - mse:
165.2846 - val_loss: 170.0264 - val_mse: 170.0264
Epoch 98/100
150/150 [==============================] - 0s 703us/step - loss: 165.5491 - mse:
165.5491 - val_loss: 175.4542 - val_mse: 175.4542
Epoch 99/100
150/150 [==============================] - 0s 685us/step - loss: 166.0434 - mse:
166.0434 - val_loss: 167.5639 - val_mse: 167.5639
Epoch 100/100
150/150 [==============================] - 0s 698us/step - loss: 165.0826 - mse:
165.0826 - val_loss: 165.4540 - val_mse: 165.4540
```

Out[45]: `<tensorflow.python.keras.callbacks.History at 0x7f9788982c10>`

In [46]:

```
# report metrics
report_results(model)
```

```
Training r2 Score:  0.5481256053171382
Test r2 Score:  0.5395059376425498
-----
Training RMSE:  12.77329304735371
Test RMSE:  12.855790025360303
-----
Training MAE:  9.399962305872167
Test MAE:  9.42652765803781
```

Because neural networks can be implimented in so many ways, a separate notebook called `neural_network_lab` has been created to attempt to optimize the neural network.

Even after many itterations of neural networks, the baseline random forest regressor performed better than any other auditioned model.

Model Tuning

The best performing baseline model is the random forest regressor. The model will be analyzed to prevent overfitting and the model's hyperparameters will be tuned to improve the model's performance.

NOTE: The grid search in this section can take a long time. For convenience, best parameters found in this grid search will be hard coded at the end of the section.

In [47]:
```
# import gridsearchCV from skikit-learn
from sklearn.model_selection import GridSearchCV
```

In [48]:
```
# define model hyperparameters to audition
param_grid = {'n_estimators': [10, 50, 100],
              'max_depth': [None, 2, 3, 4, 5, 6],
              'min_samples_split': [2, 5, 10],
              'min_samples_leaf': [1, 2, 3, 4, 5, 6]}
```

In [49]:
```
# instantiate model
rfr_tune = RandomForestRegressor()
```

In [50]:
```
# THIS CELL CAN TAKE A LONG TIME TO RUN. RESULTS ARE REPORTED BELOW

# rf_grid_search = GridSearchCV(rfr_tune, param_grid,
#                               n_jobs=-1, cv=3,
#                               return_train_score=True)

# rf_grid_search.fit(X_train, y_train)
```

In [51]:
```
# CELL IS DEPENDENT ON PREVIOUS CELL

# rf_grid_search.best_params_
```

In [52]:
```
# CELL IS DEPENDENT ON PREVIOUS CELL

# report_results(rf_grid_search)
```

In [53]:
```
# Results of grid search
rfr_parameters = {'max_depth': None,
                  'min_samples_leaf': 2,
                  'min_samples_split': 5,
                  'n_estimators': 100}
```

Model Evaluation

A random forest regressor with tuned hyperparameters is instantiated below.

This model will be used to evaluate performance and to explore any possible anomalies.

In [54]:
```
# final model instantiated
```

```
rfr_final = RandomForestRegressor(max_depth=None,
                                  min_samples_leaf=2,
                                  min_samples_split=5,
                                  n_estimators=100,
                                  n_jobs=-1)
```

In [55]:
```
# model fit to training data
rfr_final.fit(X_train, y_train)
```

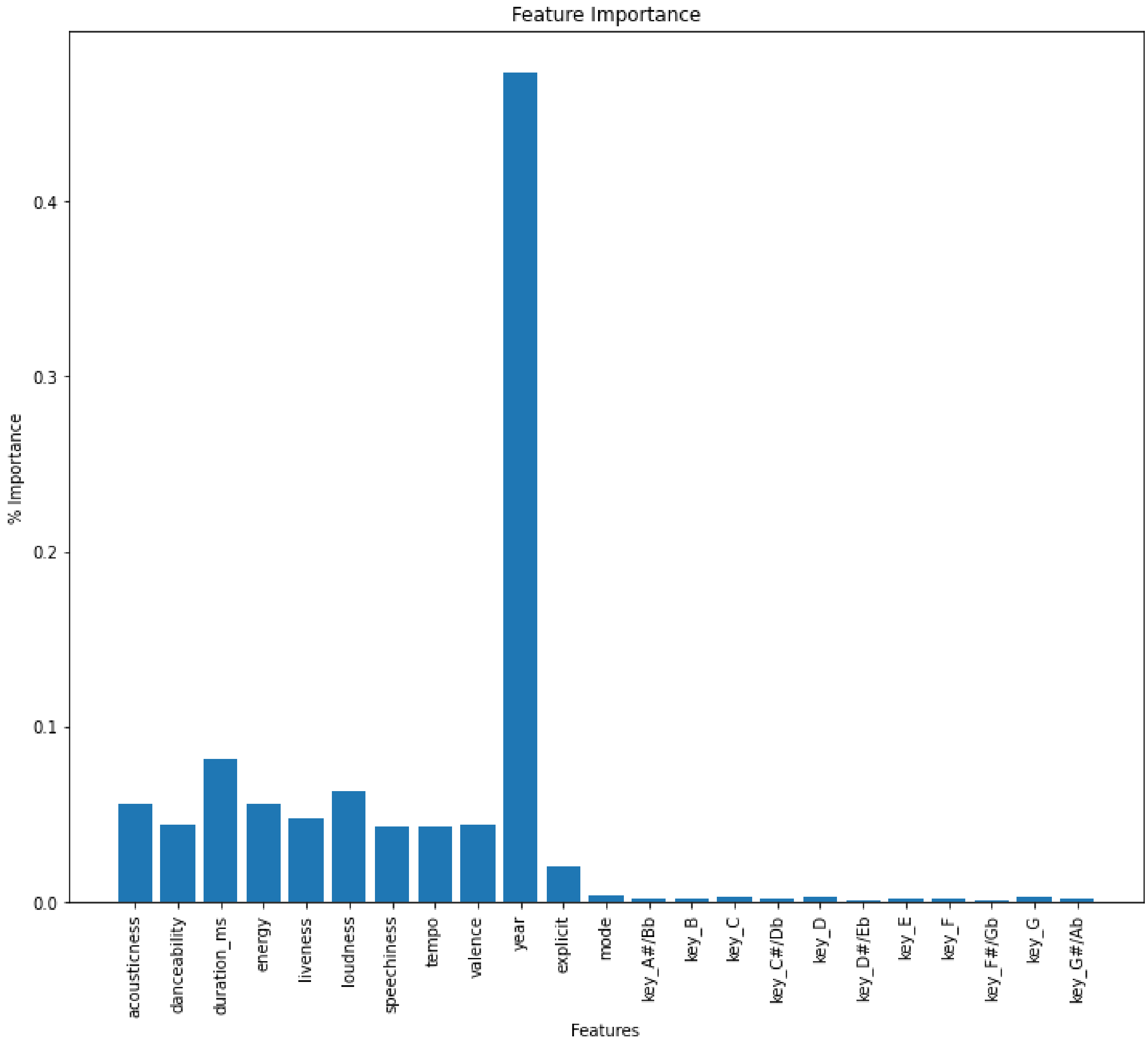
Out[55]:
```
RandomForestRegressor(min_samples_leaf=2, min_samples_split=5, n_jobs=-1)
```

In [56]:
```
# report metrics
report_results(rfr_final)
```

```
Training r2 Score:  0.9021018658232456
Test r2 Score:  0.5857929623218445
-----
Training RMSE:  5.94540189052529
Test RMSE:  12.192576519741792
-----
Training MAE:  4.122760817899827
Test MAE:  8.891543572922862
```

In [57]:
```
# save ordered list of feature importances
feature_importance = rfr_final.feature_importances_
```

In [58]:
```
# display feature importance
plt.figure(figsize=(12,10))
plt.bar(X_train.columns, feature_importance)
plt.xticks(rotation=90);
plt.title('Feature Importance');
plt.xlabel('Features');
plt.ylabel('% Importance');
```



After tuning, performance of the model on test data remained relatively consistant but performance dropped slightly when predicting values for training data, suggesting the model may have been overfit with default hyperparameters.

The feature importance suggests an possible overreliance on release year and very little effect due to a song's key or mode. That hypothesis is explored below.

Models With Lean Features

To investigate if a model performs well without the features of low importance in our tuned model, a baseline will be created without those features and compared to the tuned and baseline random forest models.

Additionally, another baseline model will be created without the features of the highest importance and without the `'year'` column to investigate if the high reliance on that factor may be hindering the model performance.

In [59]:
```
# create training data without song key columns
new_columns = X_train.columns[:11]
```

```
X_train_2 = X_train[new_columns]
X_test_2 = X_test[new_columns]
```

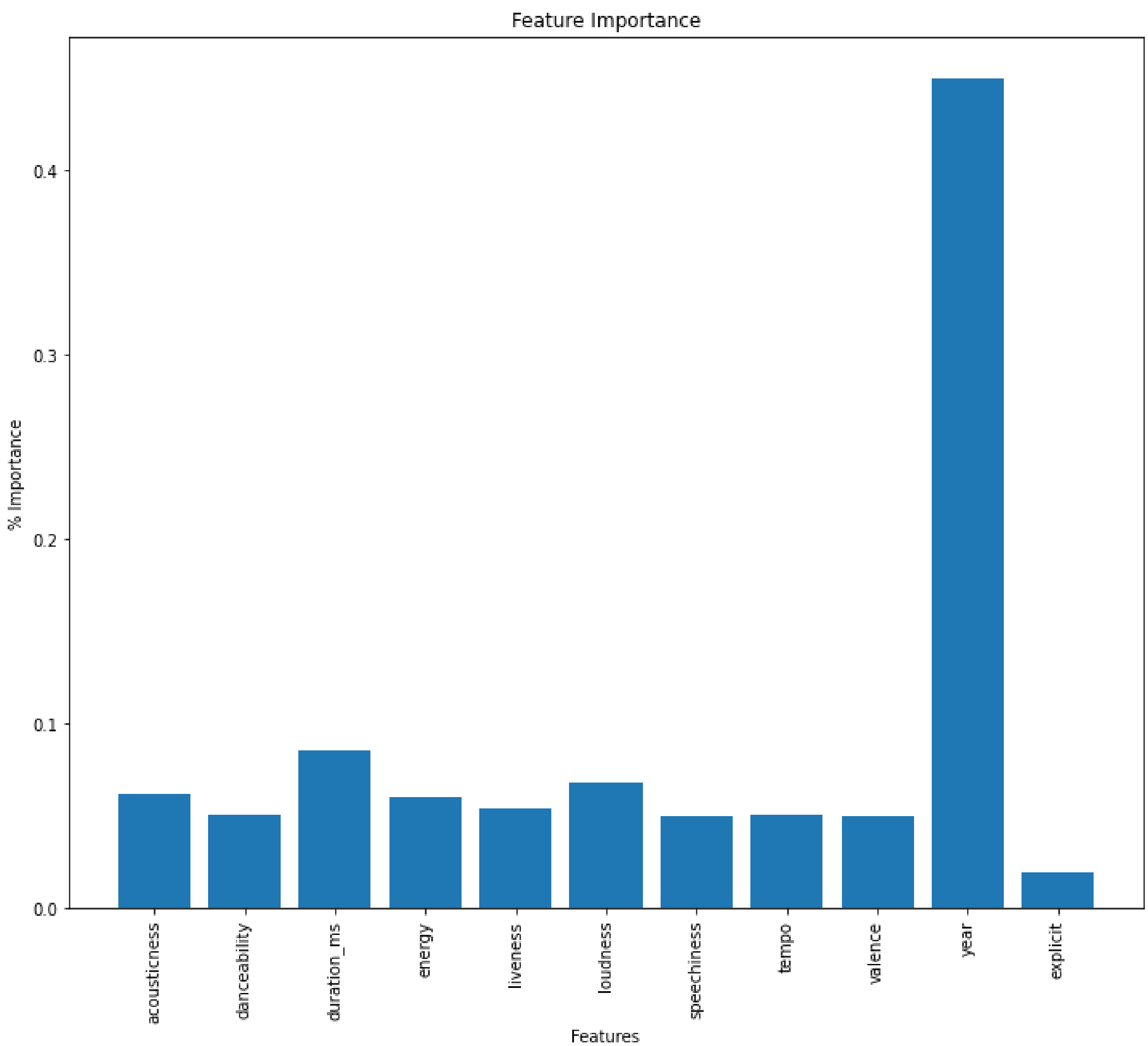
In [60]:

```
# instantiate model, fit, and report results
alt_rfr = RandomForestRegressor(n_jobs=-1)
alt_rfr.fit(X_train_2, y_train)
report_results(alt_rfr, X_train_2, y_train, X_test_2, y_test)
```

```
Training r2 Score:  0.9392213993828016
Test r2 Score:  0.5837056485370622
-----
Training RMSE:  4.684565954374459
Test RMSE:  12.223258943080435
-----
Training MAE:  3.3876128616961654
Test MAE:  8.939540570814126
```

In [61]:

```
# display feature importance
plt.figure(figsize=(12,10))
plt.bar(X_train_2.columns, alt_rfr.feature_importances_)
plt.xticks(rotation=90);
plt.title('Feature Importance');
plt.xlabel('Features');
plt.ylabel('% Importance');
```

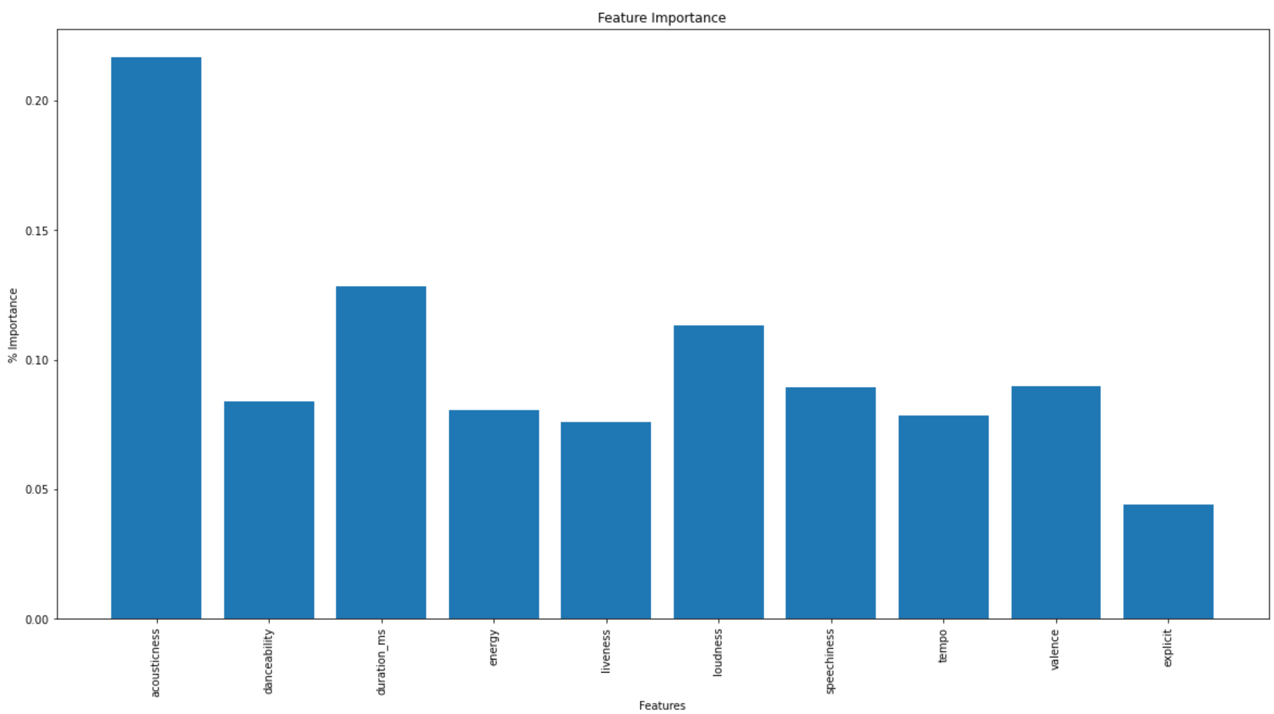


In [62]:
```
# create training data without song key or year info
X_train_3 = X_train_2.drop('year', axis=1)
X_test_3 = X_test_2.drop('year', axis=1)
```

In [63]:
```
# instantiate and fit model. report metrics
alt_2_rfr = RandomForestRegressor(n_jobs=-1)
alt_2_rfr.fit(X_train_3, y_train)
report_results(alt_2_rfr, X_train_3, y_train, X_test_3, y_test)
```

```
Training r2 Score:  0.9092024537928177
Test r2 Score:  0.38381808175699506
-----
Training RMSE:  5.725732646215906
Test RMSE:  14.87103610396275
-----
Training MAE:  4.421185618092161
Test MAE:  11.616396294463309
```

In [64]:
```
# display feature importance
plt.figure(figsize=(20,10))
plt.bar(X_train_3.columns, alt_2_rfr.feature_importances_)
plt.xticks(rotation=90);
plt.title('Feature Importance');
plt.xlabel('Features');
plt.ylabel('% Importance');
```



The ratios of feature importance for the two alternative models stayed relatively unchanged. Removing the song key information decreased model performance by a negligible amount, but removing the song release year severely decreased the fit of the model.

This shows that the model can perform well without data on song key, but year released is critical for the model.

Conclusions

The model that works best to predict a songs popularity on Spotify is a Random Forest Regressor.

The model developed in this notebook explains the variance of 58% of the data. This model's predictions tend to fall within 8-12 points of the actual popularity score, making this model a potentially invaluable asset to a production team that ghost writes and pitches songs.

The model places a lot of influence on year released, which suggests that a period of music changes the way a song's qualities should be optimized.

Finally, a song's key and even its major/minor voicing has little to nothing to do with its success on the Spotify platform, potentially debunking the idea that different keys illicit different emotions. It may be true to the performing musicians as playing in different places on a keyboard or guitar neck feels different, but to the majority of listeners the even temperment of modern music steralizes any contributing factors a different root note might have on a song's success.

Deployment

To make the model easier to utilize, it will be exported from this notebook as a `.pickle` file and incorporated into a python script that predicts a song's popularity based on user defined inputs.

In [65]:

```
# Create one line dataframe with placeholder values to be modified by user input
X_dict = {}
for column in X_train.columns:
    X_dict[column] = X_train[column].median()
X_user = pd.DataFrame(X_dict, index=[0])
```

NOTE: `rfr_final.pickle` exceeds github's file size limit, so to run the `.py` model this notebook must be run locally first to create the exported assets.

To safeguard against accidentally committing a repository with the `.pickle` files and having to backtrack and filter out the files to upload to github, the following cell is commented out.

Uncomment all lines but the first to export the models.

In [66]:

```
# # export dataframe, scaler, and model using pickle

# import pickle

# with open('deployment/X_user.pickle', 'wb') as f:
#     pickle.dump(X_user, f)

# with open('deployment/rfr_final.pickle', 'wb') as f:
#     pickle.dump(rfr_final, f)

# with open('deployment/scaler.pickle', 'wb') as f:
#     pickle.dump(std, f)
```

The exported dataframe, model, and scaler are used in a python app that can be found in the deployment folder of this repository.

Future Work

- Refine the model to take a song's genre into consideration.
- Explore the method Spotify uses to quantify continuous attributes like `danceability`, `energy`, and `acousticness`.
- Explore different neural network structures.
- Develop a model that incorporates the future work above and takes input in the form of .wav or .mp3 files, finds song attributes autonomously, and returns a predicted popularity value.