What makes a song popular? An analysis of Spotify data

December 10, 2019

0.1 Introduction

Since its founding in 2006, Spotify has become "the world's most valuable music company," according to NPR [1]. Thanks to its terrific customer experience, Spotify currently enjoys over 217 million active users worldwide — 100 million of which are paid subscribers. Naturally, this user base has provided an enormous amount of data about the world's taste in music. Spotify generously makes much of this data available to the public via an API.

One feature Spotify publishes is a popularity score for each track between 0 and 100 (100 being highly popular). In this paper, we will analyze Spotify's data on 20,000 tracks released in the last 2 years to see if the other features recorded about each track be used to determine what makes a track popular–according to Spotify's definition.

0.2 Literature Review

This type of analysis on Spotify data has been done before, however the research community seems most interested in using newer methods of machine learning to answer the research question posed here [2], [3], [4]. The results generally show that the Spotify features alone are not a complete enough feature set to explain all the variability in track popularity. However, some statistically significant relationships can be shown between track features and popularity. This analysis differs from others in the choice of sample to be analyzed and the exact methods used for feature selection.

0.3 Gathering and Preparing the Data

0.3.1 Requesting the Data

A friendly band of Spotify lovers in the developer community maintain a package called `Spotipy` which is a wrapper around the Spotify web API. The following is an example of the code written to pull down audio features for this analysis.

`client` and `secret` are credentials that can be obtained at no charge by registering as a developer with Spotify. Note that Spotify places a maximum of 50 tracks that can be retrieved per API call, so this code retrieves 50 tracks at a time.

[3]:
```
# initialize a spotify client to make API calls
client_credentials_manager = SpotifyClientCredentials(client_id=client,␣
 ↪client_secret=secret)
sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)
```

```
def get_audio_features(track_ids: list, filename='features.tsv', batchsize=50)␣
↪-> None:
    """Request audio features from the Spotify API for the tracks
    associated with the given list of ids and save them in a .tsv file"""
    num_ids = len(track_ids)
    for offset in range(0, num_ids, batchsize):
        # query for the batch
        ids_batch = track_ids[offset:offset+batchsize]
        features = sp.audio_features(ids_batch)
        # remove the 'None's from the batch
        features = [ feature for feature in features if feature is not None ]
        # parse to dataframe and write to file
        df = pd.DataFrame(features)
        # append to the the file if it already exists
        if path.exists(filename):
            df.to_csv(filename, mode='a', header=False, sep='\t')
        else:
            df.to_csv(filename, sep='\t')
```

0.3.2 Columns in the Data

With code similar to the above, I queried Spotify's `tracks` and `audio features` REST API endpoints for 20,000 songs released in 2019 and their audio features. Unfortunately, Spotify does not expose the genre of each track over its APIs, but many other features are available. Here are the attributes collected along with Spotify's own descriptions of what they mean:

Feature	Type	Description
id	string	The Spotify ID for the track.
name	string	The name of the track.
artists	array	The artists who performed the track.
year	integer	The year the track was released.
popularity	integer	The popularity of the track. The popularity of a track is a value between 0 and 100, with 100 being the most popular. The popularity is calculated by algorithm and is based, in the most part, on the total number of plays the track has had and how recent those plays are.
duration_ms	integer	The duration of the track in milliseconds.
explicit	boolean	Whether or not the track has explicit lyrics (true = yes it does; false = no it does not OR unknown).
key	integer	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.
mode	integer	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.

Feature	Type	Description
time_signature	integer	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).
instrumentalness	float	Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context.
loudness	float	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track…
tempo	float	float The overall estimated tempo of a track in beats per minute (BPM).
speechiness	float	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.
acousticness	float	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	float	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.
valence	float	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)…
liveness	float	Detects the presence of an audience in the recording.
energy	float	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.

0.3.3 Data Cleaning

Spotify's data is well curated so there is little cleaning to do besides removing duplicates that may have appeared during the query process. Note that this de-duping technique does not remove duplicate songs in different albums. Preliminary data exploration showed that some artists, such as Ludwig Van Beethoven, have hundreds of albums released in 2019 alone. Although these songs are technically duplicates, this is an analysis of individual *tracks*, not songs, so I still consider them valid for analysis.

[4]:
```python
# read in the data
tracks = pd.read_csv('tracks.tsv', sep='\t')
features = pd.read_csv('features.tsv', sep='\t')
# join the dataframes by Spotify track id
data = pd.merge(tracks, features, left_on='track_id', right_on='id')
# drop the 250 duplicate rows
data = data.drop_duplicates()
# cast boolean types to integers
data *= 1
# show the first five rows of a few columns
```

```
data[['track_name', 'artist_name', 'popularity']].head()
```

```
[4]:                                 track_name   artist_name  popularity
     0                         HIGHEST IN THE ROOM  Travis Scott          99
     2                                     Circles   Post Malone         100
     4   Bandit (with YoungBoy Never Broke Again)     Juice WRLD          94
     6                           Lose You To Love Me  Selena Gomez          98
     8                       223's (feat. 9lokknine)     YNW Melly          92
```

0.4 Limitations

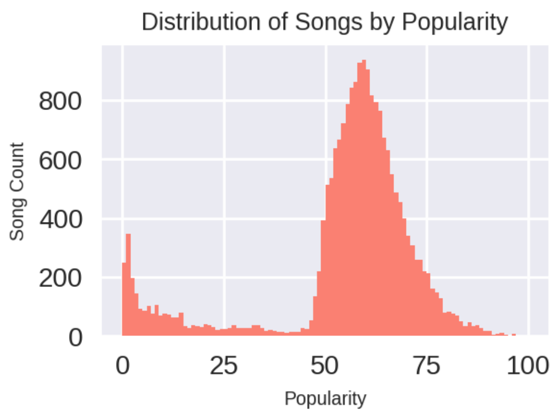
0.4.1 Bias of the Data

For this analysis, I wanted the sample of songs to be as random as possible. However, the various Spotify API endpoints only allow developers to query for tracks by artist, album, name, key words, year, etc. This limitation is totally understandable from the company's standpoint. I chose to query for songs by year so that the queried sample would be most likely to capture songs from all genres, artists, and levels of popularity.

Another limitation is that Spotify imposes a 10,000 maximum number of tracks that can be retrieved from any one query. Running the same query twice returns identical results, so I was only able to take 10,000 results from the years 2018 and 2019 each–totaling 20,000 records.

Below is a visualization of how the 20,000 songs are distributed by popularity. Clearly, the Spotify's algorithm behind the querying API is biased toward popular songs. That makes sense given that this query function is intended for Spotify users, and not analysts.

```
[5]: # visualize distribution of songs by popularity
     with plt.rc_context({'figure.figsize': SMALL_FIG}):
         ax = tracks.popularity.plot.hist(bins=100, title='Popularity',␣
      ↪color='salmon')
         ax.set_xlabel('Popularity', fontsize=SMALL_FONT)
         ax.set_ylabel('Song Count', fontsize=SMALL_FONT)
         ax.set_title('Distribution of Songs by Popularity', fontsize=LARGE_FONT);␣
      ↪plt.show()
```



0.5 Data Exploration

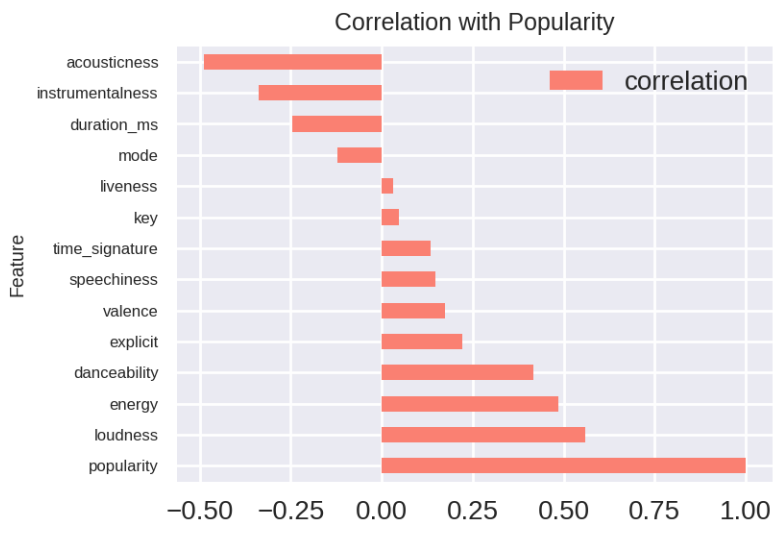
Before fitting a model to the data, we will do some more visualizations to better understand the makeup of data, and how we may need to engineer it for a model to be appropriate.

[6]:
```
# group features by type
target_col = 'popularity'
feature_cols = [
    'duration_ms', 'acousticness', 'danceability', 'energy', 'instrumentalness',
    'liveness', 'loudness', 'mode', 'speechiness', 'time_signature', 'valence',
    'key', 'explicit'
]; categorical = ['time_signature', 'key']; binary = ['mode', 'explicit']
```

0.5.1 Correlations and Feature Engineering

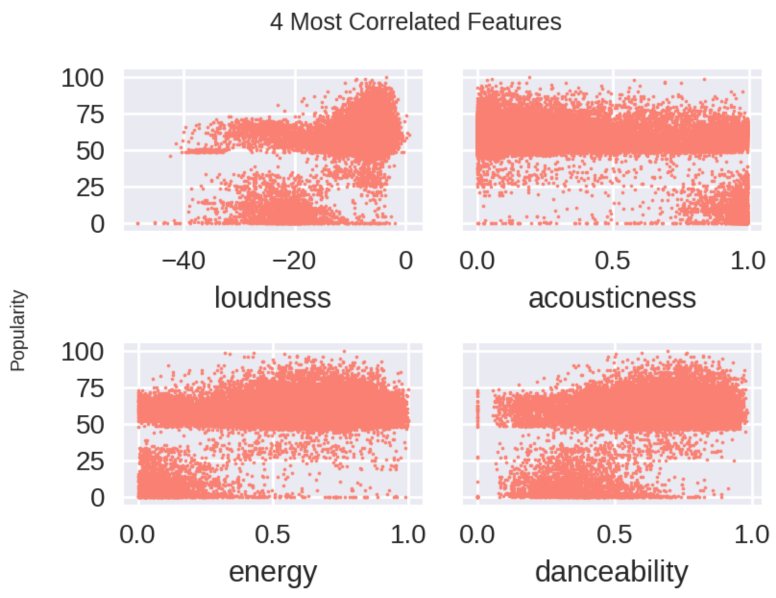
Here is a list of correlation coefficients between track feature and track popularity. Based on this list , it appears that `loudness`, `energy`, `danceability`, and `acousticness` are moderately correlated.

[7]:
```
# convert all data to float type
values = (data[[target_col] + feature_cols].values * 1).astype('float')
# compute correlation between all predictors and popularity
corrs = np.corrcoef(values.T)
# sort features by correlation
corrs_df = pd.DataFrame({'feature': [target_col] + feature_cols, 'correlation':␣
 ↪corrs[0]})
correlations = corrs_df.sort_values('correlation', ascending=False).
 ↪reset_index(drop=True)
# plot correlations of each feature
with plt.rc_context({'figure.figsize': LARGE_FIG, 'ytick.labelsize': 6}):
    correlations.plot.barh(x='feature', y='correlation', color='salmon')
    plt.title('Correlation with Popularity', fontsize=LARGE_FONT)
    plt.ylabel('Feature', fontsize=SMALL_FONT)
    plt.show()
```



The following plots can help us get a better understanding of how strongly these features are correlated with track popularity.

[8]:
```
fig, axes = plt.subplots(2, 2, sharey=True)
# create scatter plots of the 4 most correlated features
top_4_correlated = ['loudness', 'acousticness', 'energy', 'danceability']
for index, feature in enumerate(top_4_correlated):
    row = index // 2
    col = index % 2
    ax = axes[row, col]
    ax.scatter(data[feature], data.popularity, s=1, color='salmon')
    ax.set_xlabel(feature)
# plot
with plt.rc_context({'figure.figsize': SMALL_FIG, 'ytick.labelsize': 6}):
    fig.text(-.03, 0.5, 'Popularity', va='center', rotation='vertical',␣
↪fontsize=SMALL_FONT)
    fig.suptitle('4 Most Correlated Features', fontsize=LARGE_FONT, y=1.05)
    plt.tight_layout()
    plt.show()
```



The lack of datapoints with popularity scores between 30 and 50 is very apparent in these plots. One thought I had after seeing `energy` plotted against `popularity` is that the relationship between the two could be inverse quadratic. I applied a square and a square-root transformation to `energy` and found that in both cases the correlation coefficient improved by ~0.03. This improvement was small enough that I decided to leave `energy` untransformed. One advantage to leaving `energy` alone is that its regression coefficient will remain easy to interpret. No other correlation coefficients improved with similar transformations, so I left them alone as well.

Finally, I plotted some of the other features, which confirms what the correlation coefficients imply–that there is no obvious relationship between them and popularity.

0.5.2 One-hot Encoding Non-numeric Features

In the Spotify API output, `key` and `time_signature` are recorded as integers. The problem with leaving them this way in a model is that it does not make sense to say that one key is "greater" than another. For this reason, I chose to one-hot encode these two features.

```
[9]: def one_hot_encode(col_name, df):
         """
         One hot encode a string valued column in place.
         Drop one of the columns if no NaNs appear
         to preserve identifiability of the data matrix.
         """
         # one hot encode the columns
         encoded_cols = df[col_name].str.get_dummies()
         col = df[col_name]
         df.drop(columns=[col_name], inplace=True)
```

```
    # if there are no nans, drop one of the one-hots
    if col.isna().sum() == 0:
        for encoded_col in encoded_cols.columns[:-1]:
            new_col_name = col_name + '_' + encoded_col
            df[new_col_name] = encoded_cols[encoded_col]
    else:
        for encoded_col in encoded_cols.columns:
            new_col_name = col_name + '_' + encoded_col
            df[new_col_name] = encoded_cols[encoded_col]
```

[10]:
```
# change key and time_signature to str dtype
data.key = data.key.apply(str)
data.time_signature = data.time_signature.apply(str)
# one-hot encode key and time_signature
one_hot_encode('key', data)
one_hot_encode('time_signature', data)
```

[11]:
```
# these are the new features
feature_cols = [
    # original features
    'duration_ms', 'acousticness', 'danceability', 'energy', 'explicit',
    'instrumentalness', 'liveness', 'loudness', 'mode', 'speechiness',␣
 ↪'valence',
    # keys (missing 'key_9')
    'key_0', 'key_1', 'key_2', 'key_3', 'key_4', 'key_5', 'key_6', 'key_7',␣
 ↪'key_8', 'key_10', 'key_11',
    # time signatures (missing 'time_signature_2')
    'time_signature_0', 'time_signature_1', 'time_signature_3',␣
 ↪'time_signature_4']
```

0.6 Regression Analysis

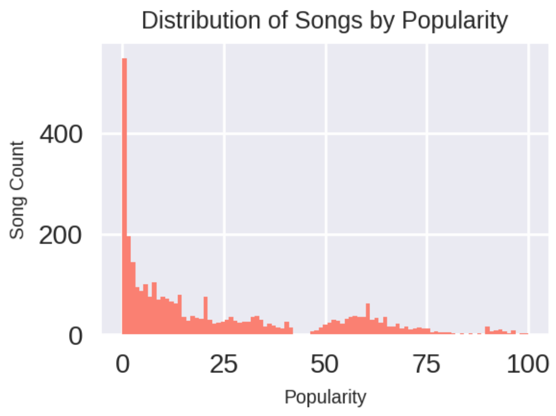
I will now resample the data to reduce the bias towards popular songs and construct many linear models. Then, I will select the set of features that minimizes the AIC metric, examine the coefficients of the corresponding linear model, and check the standard regression assumptions to make a final statement about how effective this data is for predicting popularity.

0.6.1 Resample Data to Mitigate Bias

Earlier I displayed a plot showing that the data retrieved from the Spotify API is heavily biased toward popular songs. To address this, I am going to make an assumption that the true number of popular songs in the world is tiny relative to the number of obscure songs.

From the previous visualization of the popularity scores, it looks like there is a massive spike between the values 42 and 90. To bring the data into alignment with my assumption about popular songs being rare, I chose to randomely select 4% of the songs in this range to keep. The visualization below shows the resampled distribution of songs by popularity score. While this new distribution may still not reflect the true relative proportions of song popularity scores, it is certainly closer to it than the raw pull from the API.

[12]:
```
# resample the dataframe to decrease the bias towards popular songs
over_repd_range = (data.popularity > 42) & (data.popularity < 90)
popular_songs = data[over_repd_range].sample(frac=0.04)
resample = pd.concat([data[~over_repd_range], popular_songs])
# plot density of popularity
with plt.rc_context({'figure.figsize': SMALL_FIG}):
    ax = resample.popularity.plot.hist(bins=95, color='salmon')
    ax.set_xlabel('Popularity', fontsize=SMALL_FONT)
    ax.set_ylabel('Song Count', fontsize=SMALL_FONT)
    ax.set_title('Distribution of Songs by Popularity', fontsize=LARGE_FONT)
    plt.show()
```



0.6.2 Feature Selection

After feature engineering, there are 26 potential features that could be included in a linear model. In order to find the feature set that results in the best model, a brute force approach would require fitting $2^{26} = 67,108,864$ different models. This would take a long time to compute, so instead I started by using LASSO regression to narrow down the feature set, and then used a brute forch approach on the narrowed set. We can see that the LASSO regression reduced the number of features from 26 to 9, or $2^9 = 512$ models to fit.

[13]:
```
# get data
y = resample[target_col]
X = resample[feature_cols]
# fit LASSO regression model
model = sm.OLS(y, X)
model = model.fit_regularized(alpha=.1, L1_wt=1)
print('Features from most important to least important:')
```

```
important_features = model.params[model.params > 0] \
    .sort_values(ascending=False)
# display features and coefficients
pd.DataFrame({'Feature': important_features.index,
             'L1 Coefficient': important_features.values})
```

```
Features from most important to least important:
```

[13]:
```
            Feature  L1 Coefficient
0            energy       41.580708
1      danceability       34.191543
2          explicit       11.874290
3            key_11        2.188664
4             key_6        2.131442
5  instrumentalness        1.437542
6             key_1        1.189339
7             key_7        0.984137
8       duration_ms        0.000003
```

Discover the subset of features that minimizes the AIC in a linear model.

[14]:
```
# prepare data for fitting many models
important_features = important_features.index.tolist()
y = resample.popularity
features_copy = resample.copy()[important_features]
features_copy['y-int'] = 1
# get all combinations of features
combos = []
for r in range(1, len(feature_cols)):
    combos += list(it.combinations(important_features, r))
# find the combination of features with the smallest AIC
min_aic = math.inf
best_feature_set = None
for feature_set in combos:
    # get the subset of features (include a y-intercept!)
    X = features_copy[list(feature_set) + ['y-int']]
    # fit a model
    model = sm.OLS(y, X).fit()
    # check if this aic is less than current min_aic
    if model.aic < min_aic:
        best_feature_set = feature_set
        min_aic = model.aic
best_feature_set = list(best_feature_set)
```

Sometimes after resampling the data to control for popularity bias, LASSO regression does not remove all of the `time_signature` one-hot encodings. But, the minimizing the AIC always ends up removing them. Apparently, keeping track of time signature only adds noise to the model.

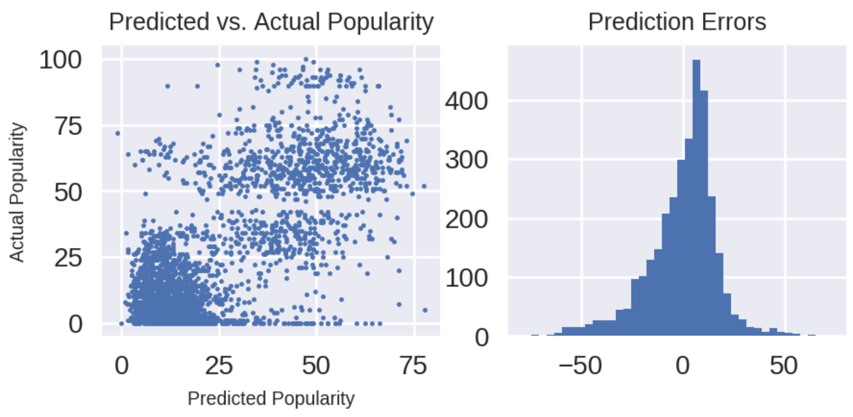
Below are the selected features. It is interesting to see that explicitness and key remain in the set.

The most predictive features

['energy', 'danceability', 'explicit', 'key_6' (F), 'key_8' (G)]

0.6.3 Regression Assumptions

[15]:
```python
y = resample.popularity
X = resample.copy()[best_feature_set]
X['y-int'] = 1
# fit a model
model = sm.OLS(y, X).fit()
# compute prediction errors
predictions = model.predict(X)
errors = predictions - y
with plt.rc_context({'figure.figsize': [5, 2]}):
    fig, axes = plt.subplots(1, 2)
    # plot prredicted popularity vs. actual popularity
    axes[0].scatter(predictions, y, s=2)
    axes[0].set_xlabel('Predicted Popularity', fontsize=SMALL_FONT)
    axes[0].set_ylabel('Actual Popularity', fontsize=SMALL_FONT)
    axes[0].set_title('Predicted vs. Actual Popularity', fontsize=LARGE_FONT)
    # plot histogram of model errors
    axes[1].hist(errors, bins=40)
    axes[1].set_title('Prediction Errors', fontsize=LARGE_FONT)
```



The two plots above support the validity of a regression approach for the question of whether the Spotify features can reasonably be used to estimate the popularity of a track from 2018-2019.

Equal Variance - The “Predicted vs. Actual Popularity” plot shows that as actual popularity increases from 0 to 100, the variance of the predicted values remains relatively constant. This indicates that the “equal variance” assumption for linear regression is satisfied by this model.

Normality of Errors - The second plot demonstrates that the prediction errors are very close to normally distributed with mean 0.

Linear Relationship between Explanatory and Response Variables - I examined this in the correlation plot "4 Most Correlated Features" above. Clearly the relationship is not perfectly linear, but the scatter plots do not seem to be so non-linear that a linear analysis would be invalid.

Independence of the Data - Each track is not independent from every other track because many groups of tracks are written by the same artist. Therefore, the asumption of independence is not met and final model interpretation should be informed of this.

0.6.4 Model Interpretation

```
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
energy          43.6839      1.474     29.640      0.000      40.794      46.574
danceability    24.5392      2.062     11.902      0.000      20.497      28.582
explicit        15.3929      1.168     13.175      0.000      13.102      17.684
key_6 (F)        2.3658      1.500      1.577      0.115      -0.575       5.307
key_8 (G)        2.2871      1.339      1.709      0.088      -0.337       4.912
y-int           -1.6844      0.815     -2.066      0.039      -3.283      -0.086
```

These are the final features and their linear coefficients. It seems intuitive that having high danceability and energy will have a high impact on popularity. It is also interesting that being explicit seems to be positively related to popularity as well.

The keys F and G are not statistifally significant in this model, but these two do have the strongest positive relationship with popularity. I thought this was interesting because F and G are two of the chords in Axis of Awesome's 4 Chords medley which the group created based on a popular theory that almost all hit songs share the same chord progression.

0.7 Conclusion

The "Regression Assumptions" section discusses how the assumptions of linear regression are not perfectly satisfied from this data sample. This was due to the fact that a track artist is likely to have a strong influence on track popularity irrespective of the other audio features that Spotify measures. Therefore, if artist and genre are ignored, this analysis shows that for songs released in 2018 and 2019, energy and dancability have a strong relationship with the popularity of a track.

0.8 References

[1] Spotify Is, For Now, The World's Most Valuable Music Company Stephen Witt .
[2] Nijkamp, Rutger. Prediction of Product Success: Explaining Song Popularity by Audio Features from Spotify Data. 10 July 2018.
[3] Kai Middlebrook: "Song Hit Prediction: Predicting Billboard Hits Using Spotify Data", 2019; arXiv:1908.08609.
[4] Georgieva, Elena et al. "HITPREDICT : PREDICTING HIT SONGS USING SPOTIFY DATA STANFORD COMPUTER SCIENCE 229 : MACHINE LEARNING." (2018).