# Spotify Data Analysis

STUDY OF SONG FACTORS AND CORRELATION TO STREAMING

JOEY BEIGHTOL

RYAN LUCERO

MATTHEW PARKER

# Datasets

Most Streamed Spotify Songs 2023: This dataset is made up of the 943 most famous songs on Spotify for 2023 and was collected by Kaggle user Nidula Elgiriyewithana

Spotify Tracks Dataset: This dataset is made up of 114k Spotify tracks ranging over 125 different genres

# Potential Challenges

Little information about data collection methodology

Unclear definitions for key terms:

| Acousticness | Danceability | Energy | Liveness | Speechiness |
|---|---|---|---|---|

# Research Question

Goal: to analyze what combination of surveyed factors are most positively correlated with streams for a song on Spotify.

Do specific musical features correlate positively to streams based more so than contextual factors (genre, artist, release date etc.)?

Is it possible to predict a song's success based on musical features alone?

# Literature Review

- Carleton University
  - Song features to do not directly correlate to its popularity
  - Contextual factors instead of musical features are stronger indicators of a song's success
  - Elements affecting song popularity change over time
- Stanford University
  - Sonic features of a song are far less predictive of its popularity than its metadata features (artist, genre, etc.)

# Analysis of Data – 2023 Most Streamed Songs

# Cleaning

- Removed columns unrelated to song characteristics

```
#Drop columns
spotifyDF = spotifyDF.drop(columns=['released_year', 'released_month', 'released_day', 'in_spotify_playlists', 'in_spotify_charts',
'in_apple_playlists', 'in_apple_charts', 'in_deezer_playlists', 'in_deezer_charts', 'in_shazam_charts'])
```

- Removed any row in the dataset where the streams were NaN

```
#Check if streams has any value that is not a number
spotifyDF['streams'] = pd.to_numeric(spotifyDF['streams'], errors='coerce')
print(spotifyDF[spotifyDF['streams'].isnull()])
#Removing any types that are not a number in streams column
spotifyDF = spotifyDF.dropna(subset=['streams'])
spotifyDF= spotifyDF.reset_index(drop=True)
spotifyDF['streams'] = spotifyDF['streams'].astype(int)
spotifyDF.dtypes
```

# Cleaning (Cont.)

- Removed Duplicates

```
#Duplication
print(spotifyDF.duplicated().sum())
spotifyDF = spotifyDF.drop_duplicates()
print(spotifyDF.duplicated().sum())
```

- Encoded Key Column

```
label_encoder = LabelEncoder()
spotifyDF['key_encoded'] = label_encoder.fit_transform(spotifyDF['key'])
```

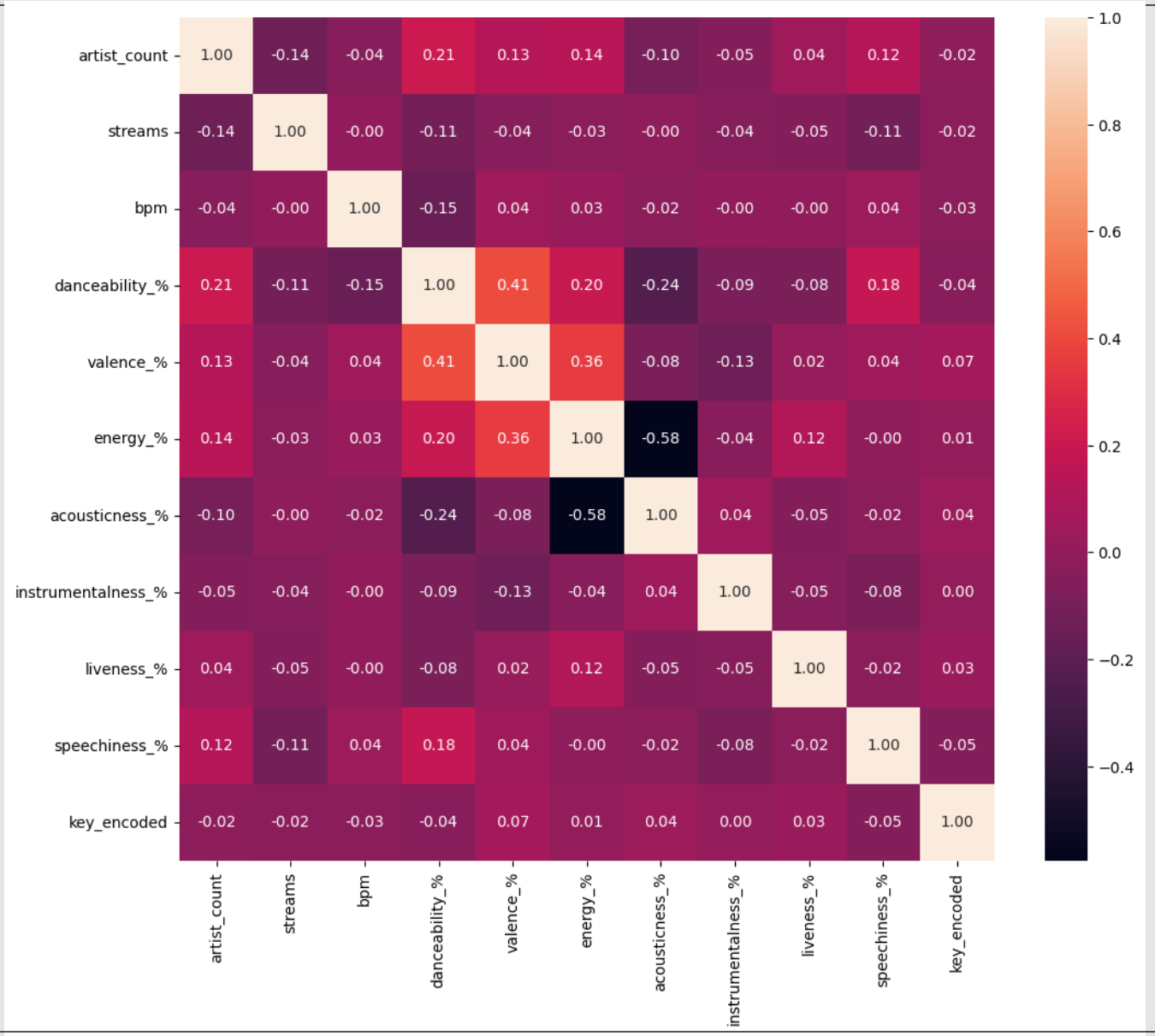
- Standardized Data

```
#Need to standardize the data
scaler = StandardScaler()
# Fit and transform the DataFrame
spotifyNumericStandard = pd.DataFrame(scaler.fit_transform(spotifyNumericDF), columns=spotifyNumericDF.columns)
```

| | artist_count | streams | bpm |
|---|---|---|---|
| 0 | 0.495653 | -0.657242 | 0.086659 |
| 1 | -0.623981 | -0.670765 | -1.089135 |
| 2 | -0.623981 | -0.659672 | 0.549850 |
| 3 | -0.623981 | 0.506203 | 1.690013 |
| 4 | -0.623981 | -0.371691 | 0.763631 |
| ... | ... | ... | ... |
| 946 | -0.623981 | -0.745292 | 0.763631 |
| 947 | -0.623981 | -0.691662 | 1.547493 |

# Correlation

- No apparent correlation between musical features and streams



# Lasso Regression

- Utilized Lasso Regression to find the dominant features
  - Used cross validation to get alpha value for regression
  - Negative is reversely related to streams and positive is related to streams
- MSE for Lasso Regression was high

```
Best alpha: 0.029521551656549 29
Mean Squared Error: 0.749770051505896
                Feature  Coefficient
0          artist_count    -0.079182
1                   bpm    -0.000000
2       danceability_%    -0.034707
3            valence_%     0.000000
4             energy_%     0.000000
5       acousticness_%    -0.000000
6   instrumentalness_%    -0.026348
7            liveness_%    -0.045496
8         speechiness_%    -0.093412
9           key_encoded    -0.000000
```

# Other Regression Analysis

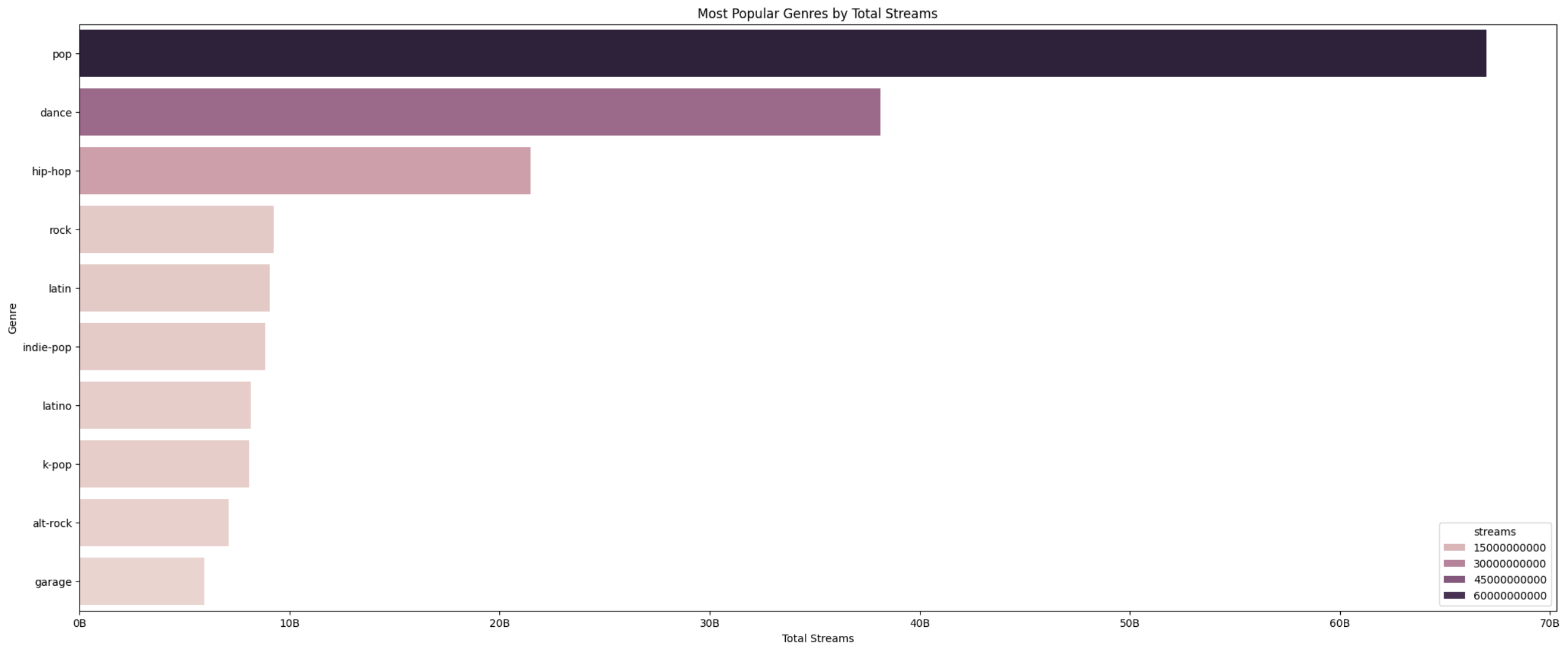
- With strong features identified from Lasso Regression, ran Linear Regression, Random Forrest Regression and Regression Tree
- All had high MSE indicating not strong predictive analysis

```python
#Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)
# Make predictions on the testing set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Linear Regression Mean Squared Error:", mse)

# Random Forrest Regression
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Random Forrest Mean Squared Error:", mse)

#Tree Regression
tree_regressor = DecisionTreeRegressor()
# Fit the regressor to the training data
tree_regressor.fit(X_train, y_train)
# Predict the target values for the test set
y_pred = tree_regressor.predict(X_test)
# Calculate Mean Squared Error (MSE) as the evaluation metric
mse = np.sqrt(mean_squared_error(y_test, y_pred))
print('Tree Regression Mean Squared Error:', mse)
```

```
Linear Regression Mean Squared Error: 0.7527008209059232
Random Forrest Mean Squared Error: 0.8710600853353992
Tree Regression Mean Squared Error: 1.2967288636011656
```

# Regression Tree

# Analysis of Data – Combined Data Set by Genre




# Total Streams per Genre

- How are genres determined?
  - Should some genres be combined?

# Cleaning

- Adjust columns to match

```
#adjust data frames to match
spotifyDF.rename(columns={'artist(s)_name': 'artists'}, inplace=True)
spotifyLargeDF.rename(columns={'key': 'large_key'}, inplace=True)
spotifyLargeDF['artists'] = spotifyLargeDF['artists'].str.replace(';', ', ')
spotifyDF['artists'] = spotifyDF['artists'].astype(str)
spotifyLargeDF['artists'] = spotifyLargeDF['artists'].astype(str)
```

- Merge data frames

```
spotifyCombined = pd.merge(spotifyDF, spotifyLargeDF[['artists', 'track_name','track_genre', 'popularity', 'large_key']], on=['artists','track_name'])
```

- Remove duplicates

| | track_name | artists | artist_count | streams | bpm | key | mode | danceability_% | valence_% | energy_% | acousticness_% | instrumentalness_% | liveness_% | speechiness_% | key_encoded |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | As It Was | Harry Styles | 1 | 2513188493 | 174 | F# | Minor | 52 | 66 | 73 | 34 | 0 | 31 | 6 | 8 |
| 1 | As It Was | Harry Styles | 1 | 2513188493 | 174 | F# | Minor | 52 | 66 | 73 | 34 | 0 | 31 | 6 | 8 |

```
spotifyCombined = spotifyCombined.drop_duplicates(subset=['artists', 'track_name'])
```

# Cleaning (Cont.)

- Key had 31 Null Values
  - Replaced all null values with key from other DF
  - Set dictionary to replace numerical value with corresponding key
  - Encoded Key

```
keyNull = spotifyCombined['key'].isnull()
spotifyCombined.loc[keyNull, 'key'] = spotifyCombined.loc[keyNull, 'large_key']

keyDict = {
    0: 'C',
    1: 'C#',
    2: 'D',
    3: 'D#',
    4: 'E',
    5: 'F',
    6: 'F#',
    7: 'G',
    8: 'G#',
    9: 'A',
    10: 'A#',
    11: 'B'
}
#replace all values of integers to strings in key column
spotifyCombined['key'] = spotifyCombined['key'].replace(keyDict)
spotifyCombined.drop('large_key', axis=1, inplace=True)
```

# Potential Issue

- Only 253 instances in new DF
- Splitting by Genre decreases this even further
- Will only look at top 8 genres

```
print(spotifyCombined['track_genre'].value_counts())
```

✓ 0.0s

```
track_genre
pop                  46
dance                39
hip-hop              27
k-pop                21
latin                14
latino               14
indie-pop            13
rock                  6
electro               5
alt-rock              5
piano                 5
garage                5
british               5
singer-songwriter     4
funk                  4
indie                 3
soul                  3
folk                  3
hard-rock             3
anime                 3
synth-pop             2
german                2
country               2
alternative           2
...
chill                 1
french                1
r-n-b                 1
```

# Standardize Data

- Similarly, we standardized the combined DF

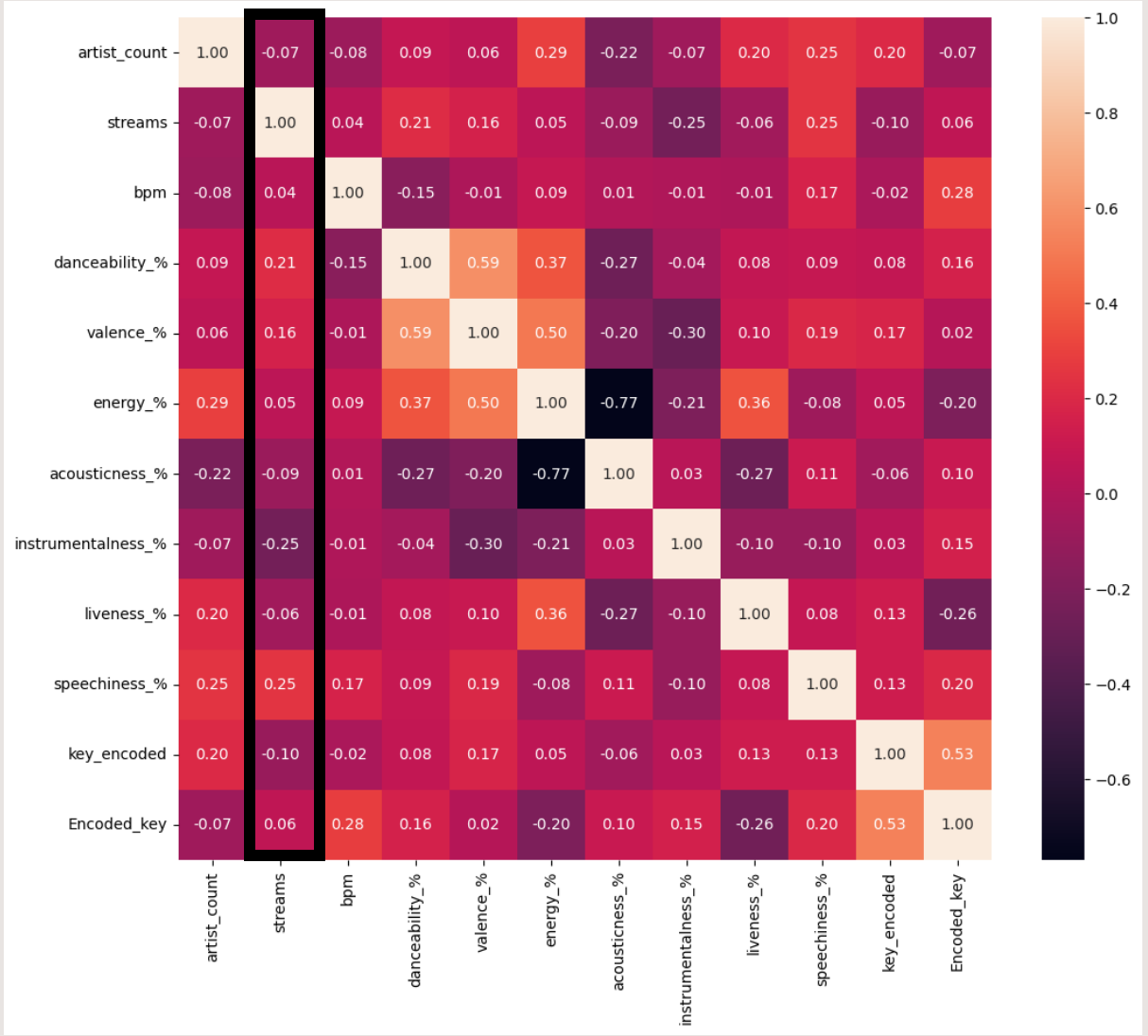
```
#standardize the data
spotifyCombinedNumericDF = spotifyCombined.select_dtypes(include=['int', 'float']) #select only numerical columns
scaler = StandardScaler() #set scaler
spotifyCombinedStandard = pd.DataFrame(scaler.fit_transform(spotifyCombinedNumericDF), columns=spotifyCombinedNumericDF.columns) #standardize the data
spotifyCombinedStandard.head(3)
```

| | artist_count | streams | bpm | danceability_% | valence_% | energy_% | acousticness_% | instrumentalness_% | liveness_% | speechiness_% | Encoded_key |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | -0.406223 | 2.296281 | 1.777979 | -0.847535 | 0.743558 | 0.521659 | 0.382610 | -0.190961 | 1.028333 | -0.352098 | 1.058738 |
| 1 | -0.406223 | 0.539031 | 0.447529 | -1.107131 | -0.253706 | -1.379536 | -0.454829 | 0.039651 | -0.538474 | -0.719189 | -0.688180 |
| 2 | -0.406223 | 0.624183 | -0.882922 | 0.645143 | 0.335587 | 0.215015 | -0.569025 | -0.190961 | -0.381793 | -0.352098 | 0.767585 |
| 3 | -0.406223 | 0.233661 | 0.618099 | 0.385546 | 1.468841 | 1.012290 | -0.492894 | -0.190961 | -0.930175 | -0.474462 | -0.688180 |
| 4 | -0.406223 | 1.046144 | 0.413414 | -0.393242 | 0.063606 | -0.766247 | -0.569025 | -0.190961 | -0.225112 | -0.229734 | -0.397027 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

# Pop – Correlation

BETTER CORRELATION, BUT STILL NOT STRONG

RUN LASSO TO SELECT PREDICTORS



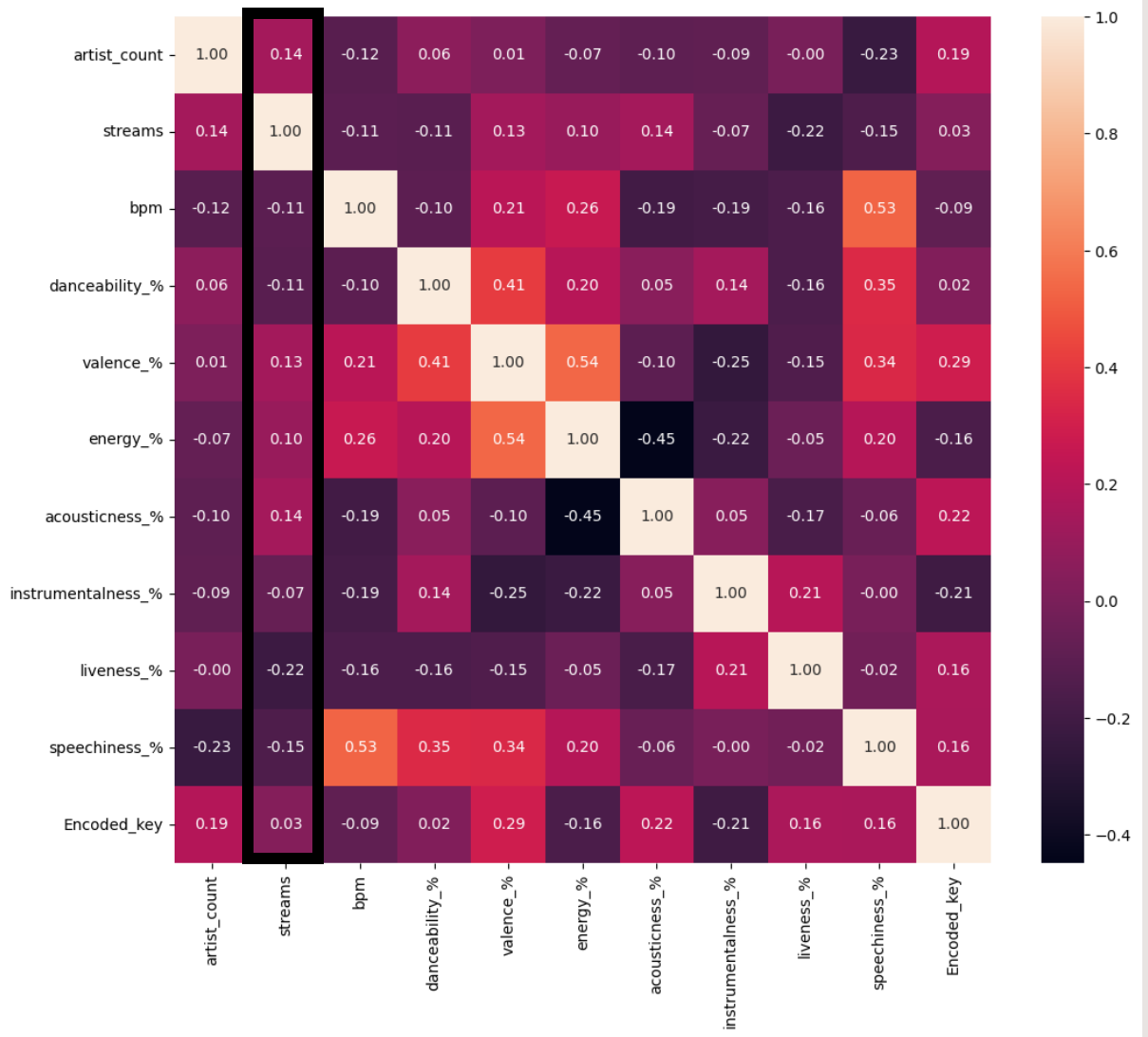
# Pop – Lasso Cross Validation

- HIGH MSE
- NO STRONG FEATURES
- NOT PREDICTIVE MODEL
  - FURTHER ANALYSIS NEEDED

```
Best alpha: 0.21530058514190886
Mean Squared Error: 1.6723679030446312
                Feature  Coefficient
0        artist_count          -0.0
1                 bpm          -0.0
2      danceability_%           0.0
3           valence_%           0.0
4            energy_%          -0.0
5      acousticness_%           0.0
6  instrumentalness_%          -0.0
7           liveness_%           0.0
8        speechiness_%           0.0
```

# Dance- Correlation

RUN LASSO TO SELECT PREDICTORS



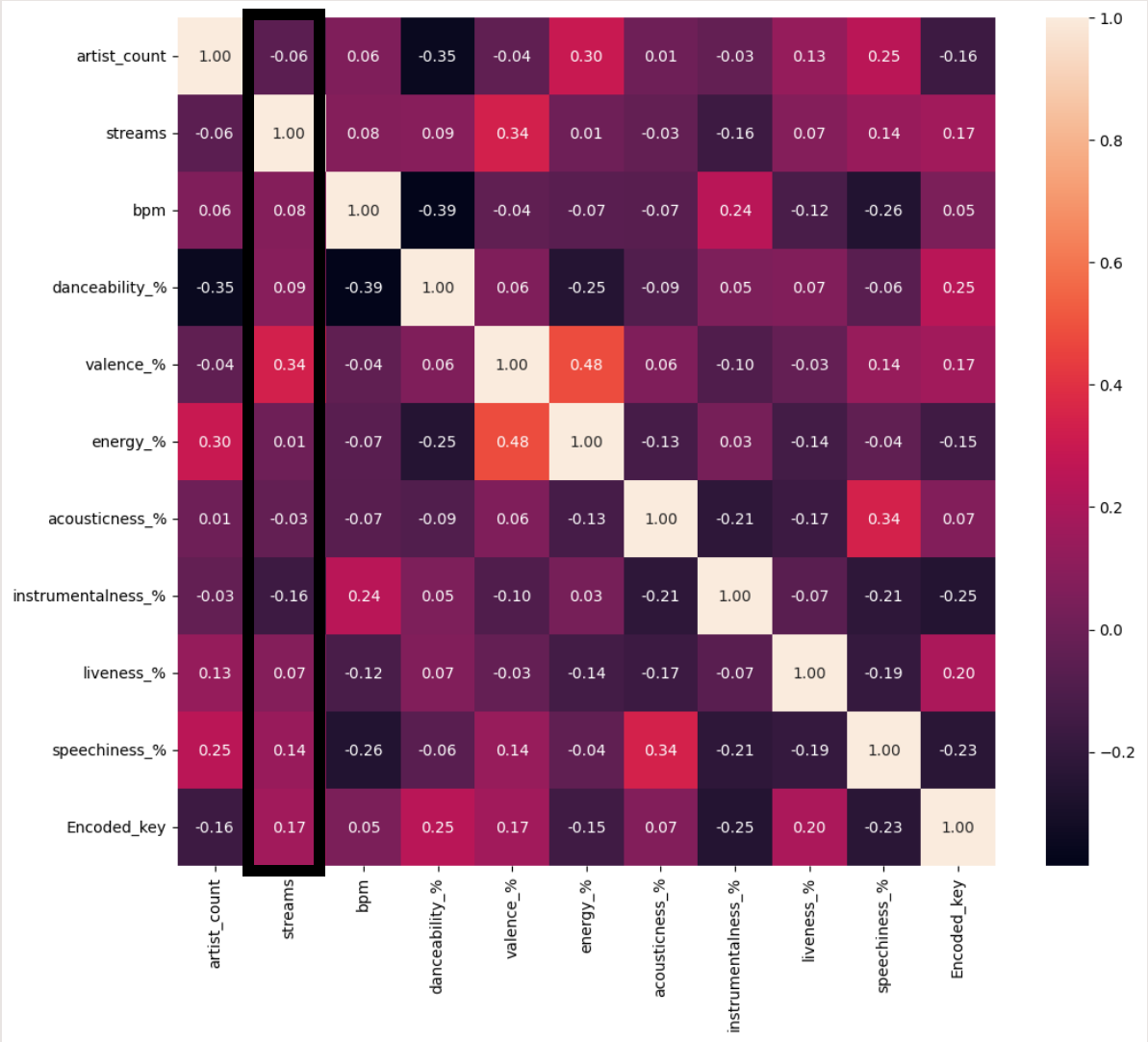
# Dance – Lasso Cross Validation

- HIGH MSE
- NO STRONG FEATURES
- NOT PREDICTIVE MODEL
  - FURTHER ANALYSIS NEEDED

```
Best alpha: 0.3467334223639891
Mean Squared Error: 1.8771712692446538
                Feature   Coefficient
0        artist_count  6.444403e-17
1                 bpm -0.000000e+00
2      danceability_% -0.000000e+00
3           valence_%  0.000000e+00
4            energy_%  0.000000e+00
5      acousticness_%  0.000000e+00
6  instrumentalness_% -0.000000e+00
7           liveness_% -0.000000e+00
8        speechiness_% -0.000000e+00
9          Encoded_key  0.000000e+00
```

# Hip Hop – Correlation

- VALENCE IS DOMINANT HERE, BUT STILL NOT STRONG
- RUN LASSO TO SELECT PREDICTORS



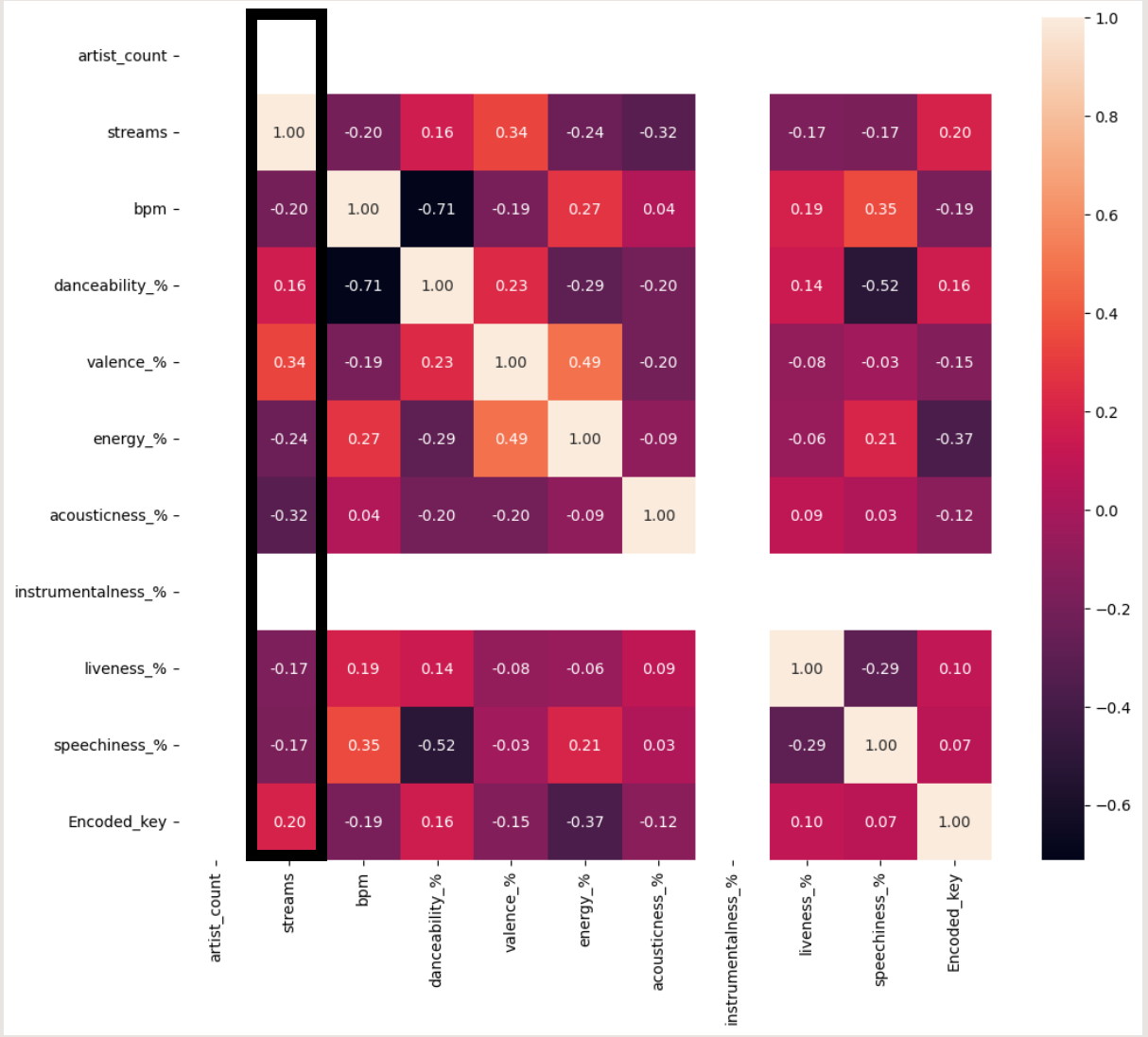
# Hip Hop – Lasso Cross Validation

- LOWER MSE, BUT STILL HIGH
- NO STRONG FEATURES
- NOT PREDICTIVE MODEL
  - FURTHER ANALYSIS NEEDED

```
Best alpha: 0.339699876175211
Mean Squared Error: 0.642470033764113
                Feature  Coefficient
0        artist_count          -0.0
1                 bpm           0.0
2      danceability_%           0.0
3           valence_%           0.0
4            energy_%          -0.0
5      acousticness_%           0.0
6  instrumentalness_%          -0.0
7           liveness_%           0.0
8        speechiness_%           0.0
9         Encoded_key           0.0
```

# K Pop – Correlation

- VALENCE AND ACOUSTICNESS ARE DOMINANT HERE, BUT STILL NOT STRONG
- RUN LASSO TO SELECT PREDICTORS



# K Pop – Lasso Cross Validation

- HIGH MSE
- POSITIVE FEATURES
  - VALENCE
  - ENCODED KEY
- NEGATIVE FEATURES
  - ENERGY
  - ACOUSTICNESS
  - LIVENESS

```
Best alpha: 0.09377895220232463
Mean Squared Error: 0.9653418747897439
                Feature  Coefficient
0          artist_count     0.000000
1                   bpm    -0.000000
2        danceability_%     0.000000
3            valence_%     0.603031
4             energy_%    -0.346632
5        acousticness_%    -0.526961
6    instrumentalness_%     0.000000
7            liveness_%    -0.051317
8         speechiness_%    -0.000000
9           Encoded_key     0.014165
```

# K Pop – Regression

- RANDOM FORREST HAD A GOOD MSE
- USE THIS TO LOOK AT STRONGEST PREDICTORS

```python
# Extract features (X) and target values (y) from the dataframe
X =  kpopLasso.drop('streams', axis=1)
y = kpopLasso['streams']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)
# Make predictions on the testing set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Linear Regression Mean Squared Error:", mse)

# Random Forrest Regression
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Random Forrest Mean Squared Error:", mse)

#Tree Regression
tree_regressor = DecisionTreeRegressor()
# Fit the regressor to the training data
tree_regressor.fit(X_train, y_train)
# Predict the target values for the test set
y_pred = tree_regressor.predict(X_test)
# Calculate Mean Squared Error (MSE) as the evaluation metric
mse = np.sqrt(mean_squared_error(y_test, y_pred))
print('Tree Regression Mean Squared Error:', mse)
```

```
Linear Regression Mean Squared Error: 5.10192888080169
Random Forrest Mean Squared Error: 0.096775547860462 91
Tree Regression Mean Squared Error: 0.24704800425297244
```

# K Pop – Regression (Cont.)

- STRONGEST FEATURES
  - ACOUSTICNESS
    - WANT LESS
  - ENERGY
    - WANT LESS
  - VALENCE
    - WANT MORE

```
Top 3 most important features:
acousticness_%: Importance Score = 0.3671
energy_%: Importance Score = 0.3300
valence_%: Importance Score = 0.3029
```

```
Best alpha: 0.09377895220232463
Mean Squared Error: 0.9653418747897439
              Feature  Coefficient
0        artist_count     0.000000
1                 bpm    -0.000000
2      danceability_%     0.000000
3           valence_%     0.603031
4            energy_%    -0.346632
5      acousticness_%    -0.526961
6  instrumentalness_%     0.000000
7           liveness_%    -0.051317
8        speechiness_%    -0.000000
9          Encoded_key     0.014165
```

# Latin – Correlation

- BPM, DANCEABILITY, SPEECHINESS, AND KEY ARE STRONGER, BUT STILL NOT HIGH
- RUN LASSO TO SELECT PREDICTORS

# Latin – Lasso Cross Validation

- HIGH MSE
- NEGATIVE FEATURES
  - BPM
  - VALENCE

```
Best alpha: 0.4197977723613463
Mean Squared Error: 1.6674730007418745
              Feature  Coefficient
0        artist_count    -0.000000
1                 bpm    -0.075971
2      danceability_%     0.000000
3           valence_%    -0.124208
4            energy_%     0.000000
5      acousticness_%     0.000000
6  instrumentalness_%     0.000000
7           liveness_%    -0.000000
8        speechiness_%    -0.000000
9         Encoded_key     0.000000
```

# Latin – Regression

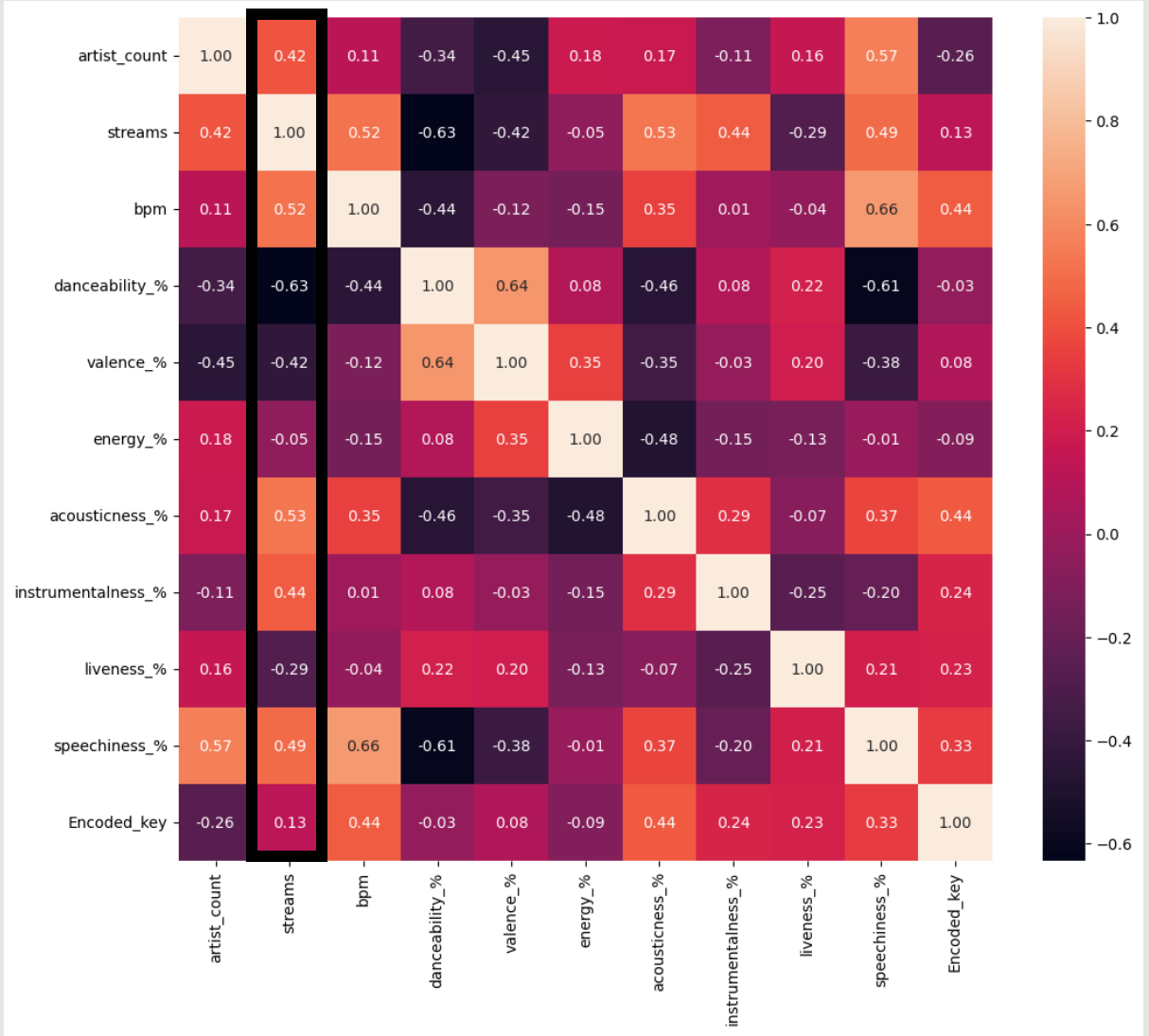
- HIGH MSE FOR ALL MODELS
- NOT PREDICTIVE
  - FURTHER ANALYSIS NEEDED

```
# Extract features (X) and target values (y) from the dataframe
X =  latinLasso.drop('streams', axis=1)
y = latinLasso['streams']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)
# Make predictions on the testing set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Linear Regression Mean Squared Error:", mse)

# Random Forrest Regression
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Random Forrest Mean Squared Error:", mse)

#Tree Regression
tree_regressor = DecisionTreeRegressor()
# Fit the regressor to the training data
tree_regressor.fit(X_train, y_train)
# Predict the target values for the test set
y_pred = tree_regressor.predict(X_test)
# Calculate Mean Squared Error (MSE) as the evaluation metric
mse = np.sqrt(mean_squared_error(y_test, y_pred))
print('Tree Regression Mean Squared Error:', mse)
```

```
Linear Regression Mean Squared Error: 2.994249923458512
Random Forrest Mean Squared Error: 3.1735103955594823
Tree Regression Mean Squared Error: 2.355282201145918
```

# Latino – Correlation

- HIGHER CORRELATION FOR SONG FEATURES, BUT STILL NOT THE BEST
- RUN LASSO TO SELECT PREDICTORS



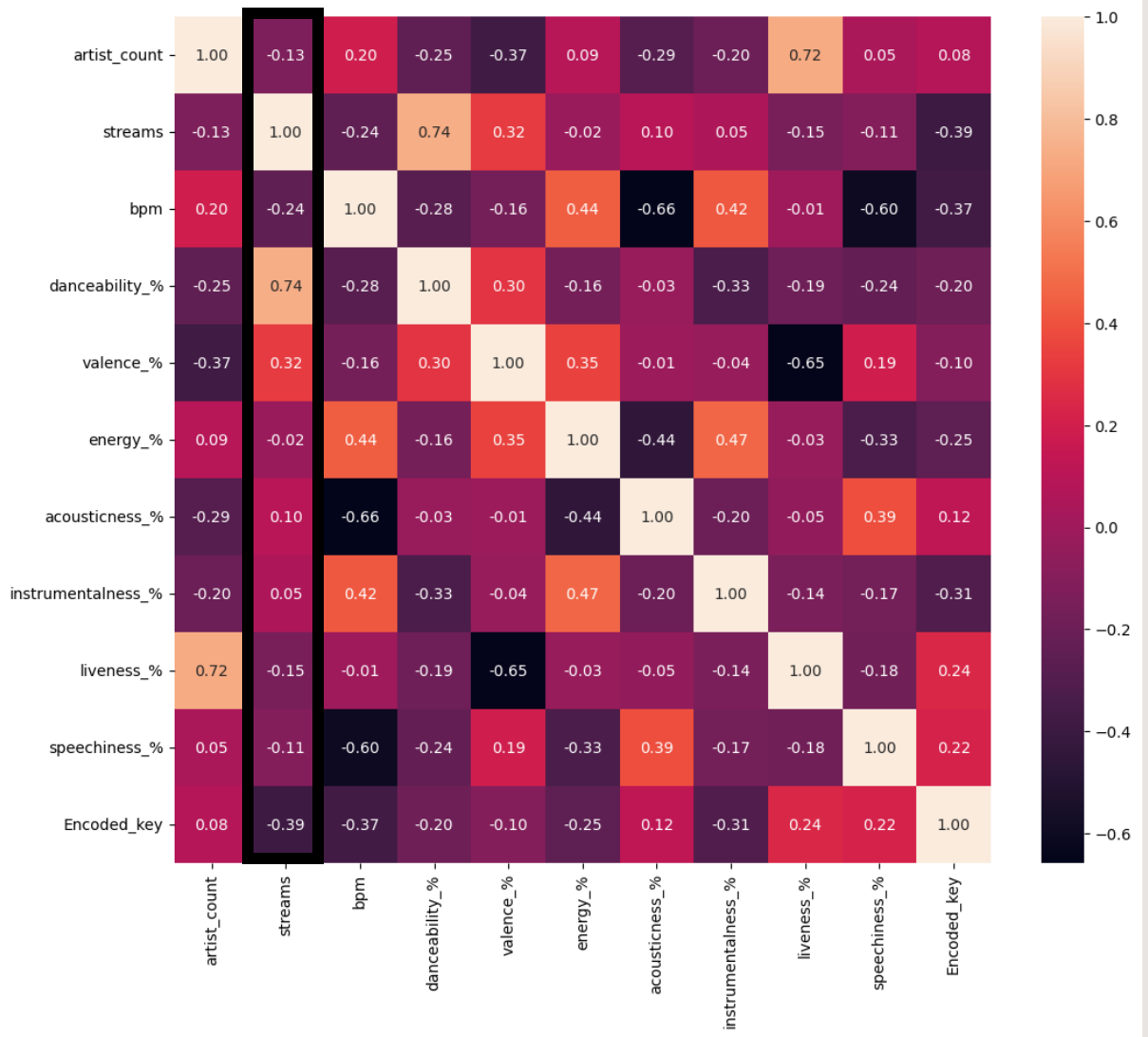
# Latino – Lasso Cross Validation

- HIGH MSE
- NO STRONG FEATURES
- NOT PREDICTIVE MODEL
  - FURTHER ANALYSIS NEEDED

```
Best alpha: 0.6686572689073627
Mean Squared Error: 1.516294274281356
                Feature  Coefficient
0          artist_count          0.0
1                   bpm          0.0
2       danceability_%         -0.0
3            valence_%         -0.0
4             energy_%         -0.0
5       acousticness_%          0.0
6   instrumentalness_%          0.0
7            liveness_%         -0.0
8         speechiness_%          0.0
9           Encoded_key          0.0
```

# Indie Pop – Correlation

- DANCEABILITY HAS HIGH CORRELATION
- RUN REGRESSION ON ONLY DANCEABILITY



# Indie Pop – Regression analysis with Danceability

- ALL HAVE HIGH MSE
- NOT PREDICTIVE, FURTHER ANALYSIS NEEDED

```
# Extract features (X) and target values (y) from the dataframe
X =  ipopStandard[['danceability_%']]
y = ipopStandard['streams']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)
# Make predictions on the testing set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Linear Regression Mean Squared Error:", mse)

# Random Forrest Regression
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Random Forrest Mean Squared Error:", mse)

#Tree Regression
tree_regressor = DecisionTreeRegressor()
# Fit the regressor to the training data
tree_regressor.fit(X_train, y_train)
# Predict the target values for the test set
y_pred = tree_regressor.predict(X_test)
# Calculate Mean Squared Error (MSE) as the evaluation metric
mse = np.sqrt(mean_squared_error(y_test, y_pred))
print('Tree Regression Mean Squared Error:', mse)
```

```
Linear Regression Mean Squared Error: 3.8909207487830404
Random Forrest Mean Squared Error: 3.882578305325634
Tree Regression Mean Squared Error: 1.9466376260193865
```

# Indie Pop – Lasso

- VERY HIGH MSE
- POSITIVE FEATURES
  - INSTRUMENTALNESS
- NEGATIVE FEATURES
  - SPEECHINESS

```
Best alpha: 0.06785970047377593
Mean Squared Error: 4.350815006884012
                Feature  Coefficient
0        artist_count    -0.000000
1                 bpm     0.000000
2      danceability_%     0.000000
3           valence_%     0.000000
4            energy_%     0.000000
5      acousticness_%    -0.000000
6  instrumentalness_%     0.122124
7           liveness_%     0.000000
8       speechiness_%    -0.037141
9         Encoded_key     0.000000
```

# Indie Pop – Regression analysis with Lasso Predictors

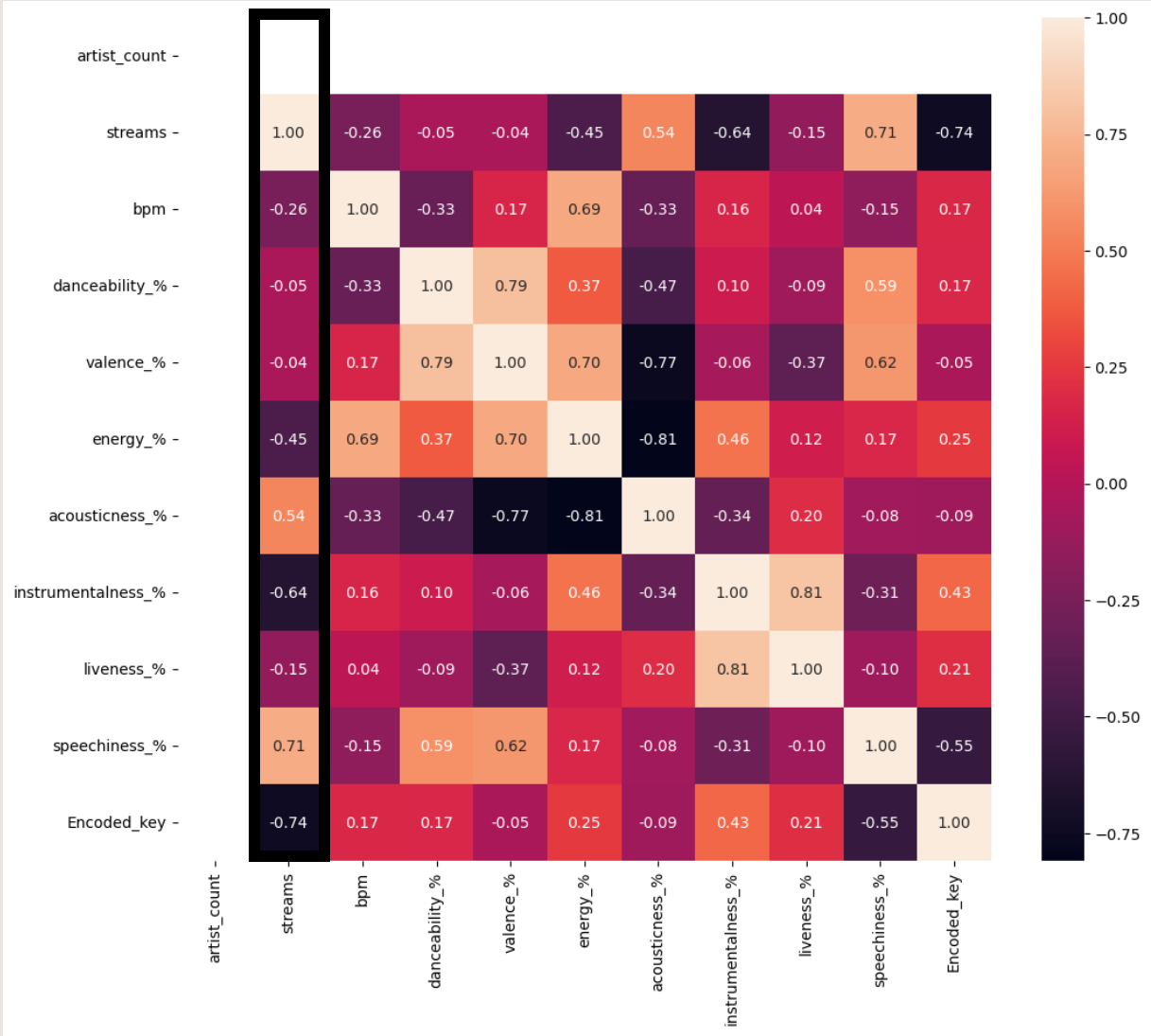
- HIGH MSE ACROSS ALL MODELS
- SMALL DATASET COULD BE ISSUE IN HIGH MSE

```
# Extract features (X) and target values (y) from the dataframe
X =  ipopLasso.drop('streams', axis=1)
y = ipopLasso['streams']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)
# Make predictions on the testing set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Linear Regression Mean Squared Error:", mse)

# Random Forrest Regression
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Random Forrest Mean Squared Error:", mse)

#Tree Regression
tree_regressor = DecisionTreeRegressor()
# Fit the regressor to the training data
tree_regressor.fit(X_train, y_train)
# Predict the target values for the test set
y_pred = tree_regressor.predict(X_test)
```

```
Linear Regression Mean Squared Error: 4.468971540972175
Random Forrest Mean Squared Error: 4.7882752256444725
Tree Regression Mean Squared Error: 2.149122118932764
```

# Rock – Correlation

- HIGH CORRELATION ON SOME FEATURES
- RUN REGRESSION ON ONLY THESE



# Rock – Regression analysis with

- ALL HAVE HIGH MSE
- NOT PREDICTIVE, FURTHER ANALYSIS NEEDED

```python
# Extract features (X) and target values (y) from the dataframe
X =  rockStandard[['Encoded_key', 'speechiness_%', 'instrumentalness_%', 'acousticness_%', 'energy_%']]
y = rockStandard['streams']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)
# Make predictions on the testing set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Linear Regression Mean Squared Error:", mse)

# Random Forrest Regression
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Random Forrest Mean Squared Error:", mse)

#Tree Regression
tree_regressor = DecisionTreeRegressor()
# Fit the regressor to the training data
tree_regressor.fit(X_train, y_train)
# Predict the target values for the test set
y_pred = tree_regressor.predict(X_test)
# Calculate Mean Squared Error (MSE) as the evaluation metric
mse = np.sqrt(mean_squared_error(y_test, y_pred))
print('Tree Regression Mean Squared Error:', mse)
```

```
Linear Regression Mean Squared Error: 1.3531963698750857
Random Forrest Mean Squared Error: 1.58858753774637
Tree Regression Mean Squared Error: 2.0397512470618007
```

# Rock – Lasso

- HIGH MSE
- POSITIVE FEATURES
  - ACOUSTICNESS
- NEGATIVE FEATURES
  - ENERGY
  - INSTRUMENTALNESS

```
Best alpha: 0.0008758349327843l9
Mean Squared Error: 1.5997659716324801
                Feature  Coefficient
0          artist_count     0.000000
1                   bpm    -0.000000
2        danceability_%    -0.000000
3             valence_%    -0.000000
4              energy_%    -0.099312
5        acousticness_%     0.388976
6    instrumentalness_%    -0.384986
7             liveness_%    -0.000000
8          speechiness_%     0.000000
9            Encoded_key    -0.000000
```

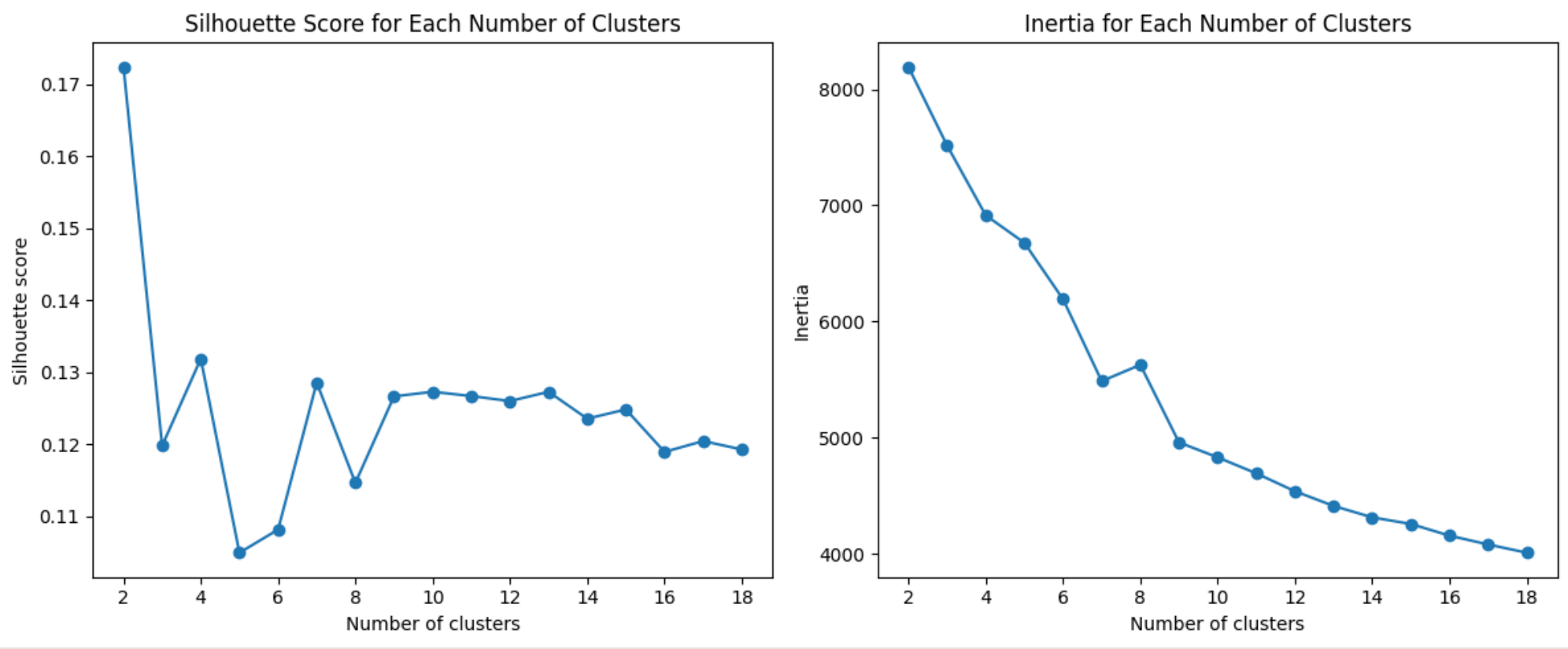
# Rock – Regression

- HIGH MSE ACROSS ALL MODELS
- SMALL DATASET COULD BE ISSUE IN HIGH MSE

```
# Extract features (X) and target values (y) from the dataframe
X =  rockLasso.drop('streams', axis=1)
y = rockLasso['streams']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Linear Regression
model = LinearRegression()
model.fit(X_train, y_train)
# Make predictions on the testing set
y_pred = model.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Linear Regression Mean Squared Error:", mse)

# Random Forrest Regression
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
# Train the model
rf_regressor.fit(X_train, y_train)
# Make predictions
y_pred = rf_regressor.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Random Forrest Mean Squared Error:", mse)

#Tree Regression
tree_regressor = DecisionTreeRegressor()
# Fit the regressor to the training data
tree_regressor.fit(X_train, y_train)
# Predict the target values for the test set
y_pred = tree_regressor.predict(X_test)
# Calculate Mean Squared Error (MSE) as the evaluation metric
mse = np.sqrt(mean_squared_error(y_test, y_pred))
print('Tree Regression Mean Squared Error:', mse)
```

```
Linear Regression Mean Squared Error: 1.5998339423377035
Random Forrest Mean Squared Error: 2.5402303885836717
Tree Regression Mean Squared Error: 2.055632287054489
```

# Cluster Analysis



# Cluster Analysis

```python
# Instantiate and fit KMeans model
kmeans = KMeans(n_clusters=7, random_state=42)
kmeans.fit(spotifySmallKmeans)
# Get cluster centers
cluster_centers = pd.DataFrame(kmeans.cluster_centers_, columns=spotifySmallKmeans.columns)

# Predict cluster labels for each customer
cluster_labels = kmeans.predict(spotifySmallKmeans)
spotifySmallKmeans = spotifySmallKmeans * spotifyNumericDF.drop(columns=['streams','artist_count']).std() + spotifyNumericDF.drop(columns=['streams','artist_count']).mean()
spotifySmallKmeans['Cluster'] = cluster_labels

# Analyze cluster characteristics
cluster_summary = spotifySmallKmeans.groupby('Cluster').agg({
    'bpm': ['mean', 'min', 'max', 'count'],
    'danceability_%': ['mean', 'min', 'max', 'count'],
    'valence_%': ['mean', 'min', 'max', 'count'],
    'energy_%': ['mean', 'min', 'max', 'count'],
    'acousticness_%': ['mean', 'min', 'max', 'count'],
    'instrumentalness_%': ['mean', 'min', 'max', 'count'],
    'liveness_%': ['mean', 'min', 'max', 'count'],
    'speechiness_%': ['mean', 'min', 'max', 'count'],
    'key_encoded': ['mean', 'min', 'max', 'count'],
}).reset_index()

print("\nCluster Summary:")
display(cluster_summary)
```

| Cluster | bpm | | | | danceability_% | | | | valence_% | ... | liveness_% | | speechiness_% | | | | key_encoded | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | mean | min | max | count | mean | min | max | count | mean | ... | max | count | mean | min | max | count | mean | min | max | count |
| 0 | 117.765836 | 77.976549 | 192.036534 | 177 | 76.807435 | 48.990547 | 95.014751 | 177 | 68.285724 | ... | 46.014629 | 177 | 7.897125 | 1.995716 | 24.007292 | 177 | 2.229789 | -0.003024 | 6.000133 | 177 |
| 1 | 120.158153 | 76.976023 | 189.034955 | 207 | 73.520355 | 40.986337 | 96.015277 | 207 | 68.235915 | ... | 41.011998 | 207 | 7.230353 | 2.996242 | 29.009923 | 207 | 8.972711 | 6.000133 | 11.002764 | 207 |
| 2 | 122.764809 | 78.977076 | 180.030220 | 17 | 60.349461 | 33.982654 | 92.013172 | 17 | 32.225218 | ... | 30.006210 | 17 | 5.409276 | 2.996242 | 9.999925 | 17 | 5.941279 | -0.003024 | 11.002764 | 17 |
| 3 | 118.817904 | 64.969709 | 206.043900 | 161 | 53.837816 | 22.976866 | 78.005806 | 161 | 37.365298 | ... | 64.024100 | 161 | 6.619265 | 1.995716 | 38.014658 | 161 | 5.894488 | -0.003024 | 11.002764 | 161 |
| 4 | 117.316683 | 66.970761 | 206.043900 | 72 | 66.263519 | 32.982128 | 92.013172 | 72 | 52.542275 | ... | 97.041464 | 72 | 8.290693 | 2.996242 | 30.010449 | 72 | 7.014556 | -0.003024 | 11.002764 | 72 |
| 5 | 130.250346 | 72.973919 | 204.042848 | 203 | 59.128883 | 30.981075 | 88.011067 | 203 | 31.413142 | ... | 41.011998 | 203 | 6.475905 | 1.995716 | 29.009923 | 203 | 5.216470 | -0.003024 | 11.002764 | 203 |
| 6 | 129.301787 | 70.972866 | 196.038638 | 114 | 73.714075 | 45.988968 | 95.014751 | 114 | 52.035430 | ... | 53.018312 | 114 | 32.292351 | 20.005187 | 64.028339 | 114 | 5.254127 | -0.003024 | 11.002764 | 114 |

# Visualization Summary

- No clear patterns were evident from the various charts for streams and individual musical features
- Possible correlation between genre and number of streams

# Path Forward

| Collect | Run | Consider |
| --- | --- | --- |
| Collect more top streamed songs with genre included | Run analysis on popularity and song features | Consider other regression analysis |

# Potential Future Analysis

- What if instead of trying to predict Streams, analysis was done to predict Genre instead?
- Running a K-Nearest Neighbors algorithm on entire dataset worth of genres

```
from sklearn.neighbors import KNeighborsClassifier

X = genreDF[['danceability', 'energy', 'loudness', 'speechiness','acousticness','instrumentalness','liveness','valence','tempo', 'duration_ms']].values
Y = genreDF['track_genre'].values

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.20)

scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, y_train)

y_predict = classifier.predict(X_test)

print(classification_report(y_test, y_predict))
```

# Results of K-Nearest Neighbors Algorithm

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| acoustic | 0.06 | 0.18 | 0.08 | 196 |
| afrobeat | 0.08 | 0.23 | 0.12 | 181 |
| alt-rock | 0.05 | 0.15 | 0.07 | 211 |
| alternative | 0.07 | 0.14 | 0.10 | 203 |
| ambient | 0.15 | 0.36 | 0.21 | 181 |
| anime | 0.06 | 0.11 | 0.08 | 229 |
| black-metal | 0.17 | 0.36 | 0.23 | 193 |
| bluegrass | 0.12 | 0.26 | 0.17 | 202 |
| blues | 0.06 | 0.11 | 0.07 | 200 |
| brazil | 0.02 | 0.04 | 0.02 | 218 |
| breakbeat | 0.12 | 0.21 | 0.15 | 223 |
| british | 0.09 | 0.17 | 0.12 | 188 |
| cantopop | 0.08 | 0.19 | 0.11 | 184 |
| chicago-house | 0.25 | 0.39 | 0.30 | 222 |
| children | 0.27 | 0.40 | 0.32 | 191 |
| chill | 0.08 | 0.12 | 0.10 | 221 |
| classical | 0.30 | 0.43 | 0.35 | 192 |
| club | 0.12 | 0.18 | 0.14 | 198 |
| comedy | 0.70 | 0.78 | 0.74 | 212 |
| country | 0.23 | 0.42 | 0.30 | 196 |
| dance | 0.21 | 0.39 | 0.27 | 198 |
| dancehall | 0.12 | 0.19 | 0.15 | 190 |
| death-metal | 0.15 | 0.20 | 0.17 | 209 |
| deep-house | 0.11 | 0.17 | 0.13 | 205 |
| detroit-techno | 0.34 | 0.33 | 0.33 | 203 |
| disco | 0.09 | 0.15 | 0.11 | 178 |
| disney | 0.19 | 0.24 | 0.21 | 199 |
| drum-and-bass | 0.37 | 0.52 | 0.43 | 198 |
| electro | 0.16 | 0.22 | 0.19 | 180 |
| electronic | 0.03 | 0.02 | 0.03 | 183 |
| emo | 0.06 | 0.08 | 0.07 | 181 |
| folk | 0.06 | 0.08 | 0.07 | 223 |
| forro | 0.29 | 0.38 | 0.33 | 183 |
| french | 0.05 | 0.06 | 0.06 | 161 |
| funk | 0.15 | 0.12 | 0.14 | 211 |
| garage | 0.06 | 0.06 | 0.06 | 190 |
| german | 0.14 | 0.11 | 0.12 | 217 |
| gospel | 0.08 | 0.07 | 0.08 | 216 |
| goth | 0.07 | 0.06 | 0.06 | 191 |
| grindcore | 0.61 | 0.59 | 0.60 | 218 |
| groove | 0.06 | 0.04 | 0.05 | 210 |
| grunge | 0.08 | 0.08 | 0.08 | 199 |
| guitar | 0.27 | 0.22 | 0.24 | 204 |
| happy | 0.31 | 0.25 | 0.27 | 201 |
| hard-rock | 0.06 | 0.06 | 0.06 | 194 |
| hardcore | 0.13 | 0.09 | 0.11 | 197 |
| hardstyle | 0.34 | 0.26 | 0.30 | 210 |
| heavy-metal | 0.16 | 0.10 | 0.12 | 199 |
| hip-hop | 0.17 | 0.13 | 0.15 | 199 |
| honky-tonk | 0.36 | 0.44 | 0.40 | 203 |
| house | 0.17 | 0.11 | 0.14 | 209 |
| idm | 0.23 | 0.13 | 0.16 | 199 |
| indian | 0.10 | 0.10 | 0.10 | 183 |
| indie | 0.16 | 0.11 | 0.13 | 201 |
| indie-pop | 0.08 | 0.03 | 0.04 | 194 |
| industrial | 0.14 | 0.05 | 0.08 | 213 |
| iranian | 0.41 | 0.19 | 0.26 | 224 |
| j-dance | 0.18 | 0.12 | 0.15 | 198 |
| reggaeton | 0.12 | 0.06 | 0.08 | 209 |
| rock | 0.28 | 0.20 | 0.23 | 200 |
| rock-n-roll | 0.13 | 0.06 | 0.09 | 201 |
| rockabilly | 0.33 | 0.14 | 0.20 | 211 |
| romance | 0.34 | 0.24 | 0.28 | 215 |
| sad | 0.12 | 0.07 | 0.09 | 182 |
| salsa | 0.56 | 0.42 | 0.48 | 212 |
| samba | 0.21 | 0.12 | 0.15 | 206 |
| sertanejo | 0.36 | 0.19 | 0.25 | 207 |
| show-tunes | 0.09 | 0.04 | 0.06 | 180 |
| singer-songwriter | 0.13 | 0.07 | 0.10 | 201 |
| ska | 0.08 | 0.03 | 0.04 | 203 |
| sleep | 0.83 | 0.72 | 0.77 | 203 |
| songwriter | 0.04 | 0.01 | 0.02 | 185 |
| soul | 0.38 | 0.25 | 0.30 | 219 |
| spanish | 0.14 | 0.05 | 0.07 | 206 |
| study | 0.51 | 0.63 | 0.57 | 202 |
| swedish | 0.30 | 0.11 | 0.16 | 189 |
| synth-pop | 0.25 | 0.12 | 0.16 | 204 |
| tango | 0.48 | 0.40 | 0.44 | 188 |
| techno | 0.12 | 0.06 | 0.08 | 211 |
| trance | 0.28 | 0.21 | 0.24 | 192 |
| trip-hop | 0.17 | 0.06 | 0.09 | 200 |
| turkish | 0.02 | 0.00 | 0.01 | 211 |
| world-music | 0.25 | 0.17 | 0.20 | 201 |
| accuracy | | | 0.18 | 22800 |
| macro avg | 0.20 | 0.18 | 0.18 | 22800 |
| weighted avg | 0.20 | 0.18 | 0.18 | 22800 |

# K-Nearest Neighbors Results

- Precision: out of all the marked positives, how many were correct
- Recall: out of all the true positives, how many were marked positive
- After running a K-Nearest Neighbors algorithm on the entire dataset, including every genre, the average precision and recall are low at around 0.20
- However, there are several genres that stand out as being able to predict well (much higher precision and recall than average)
- Some examples: comedy, drum and bass, grindcore, honky-tonk, salsa, and sleep.
- Perhaps lowering the number of genres could give better results

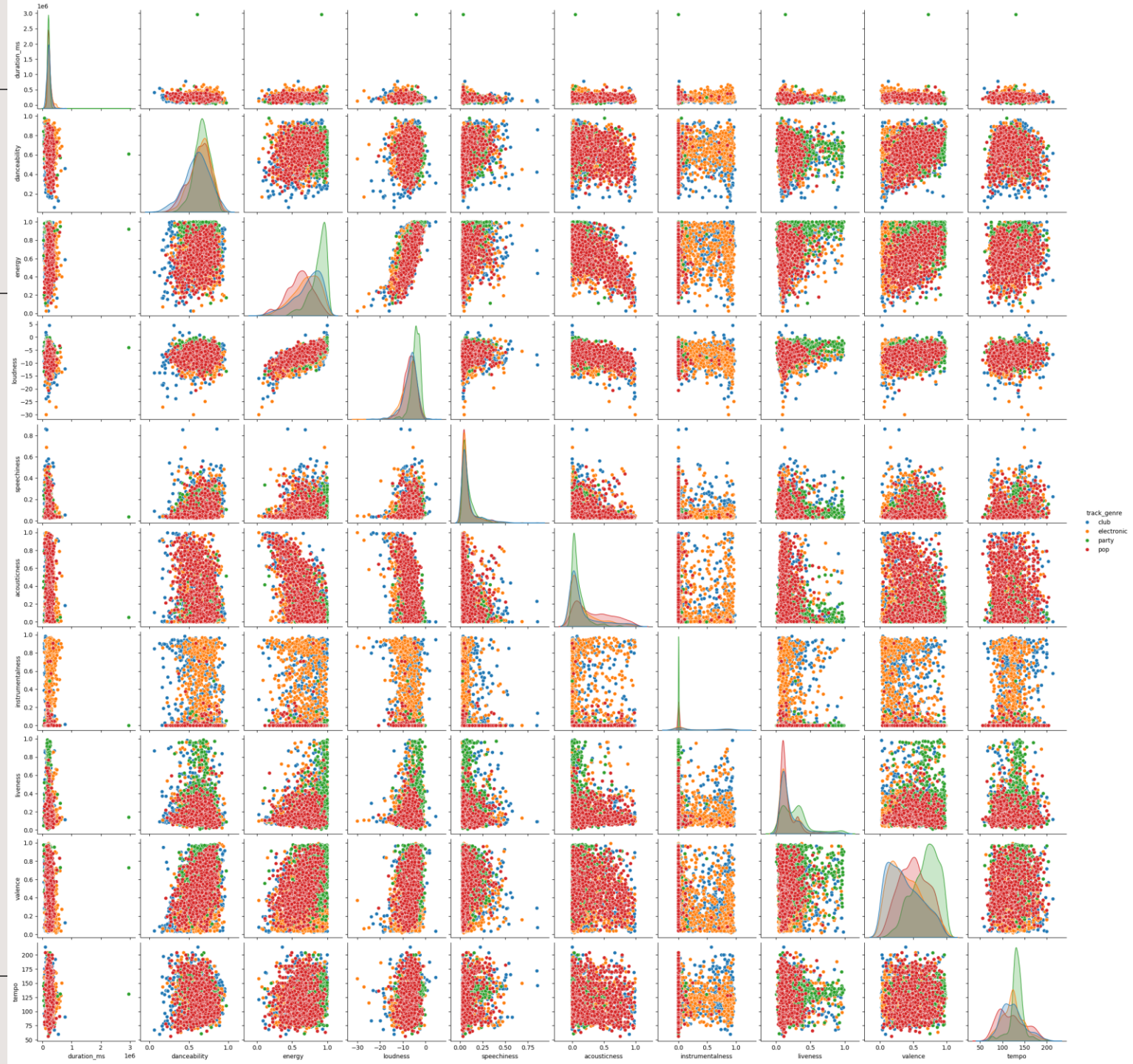
# Select Similar Genres First

- First, select some similar genres to test a baseline improvement to using every genre
- Selecting pop, party, electronic and club

```
similar_genre_list = ['pop', 'party', 'electronic', 'club']
similar_genre_df = genreDF.loc[genreDF['track_genre'].isin(similar_genre_list)]

sns.pairplot(data=similar_genre_df, hue='track_genre')
```

# Pairplot for Similar Genres

- Lots of overlap in many of the graphs
- Primarily looking at the diagonal portion, we are only seeing a big difference in valence and tempo in the bottom right.



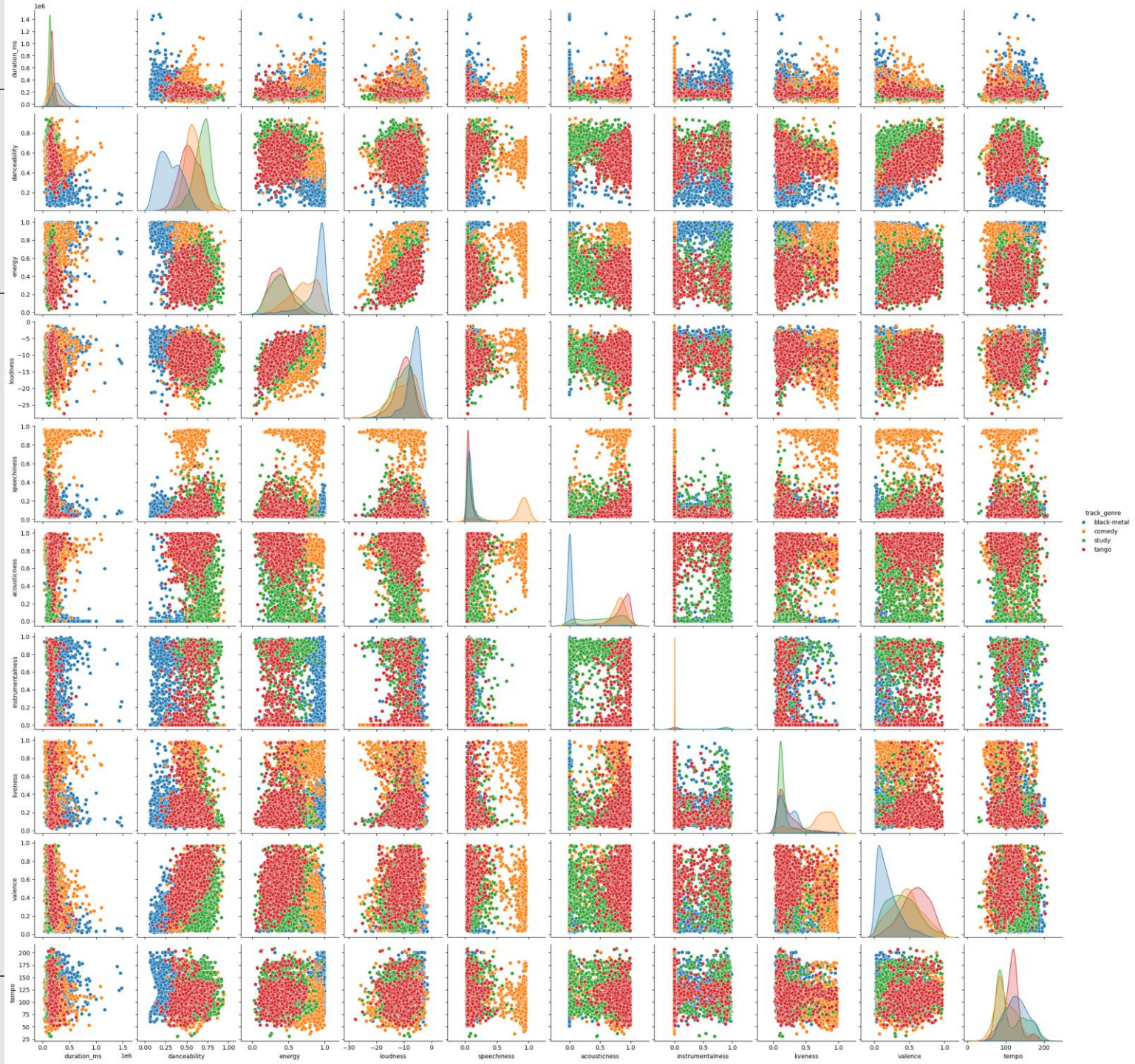
# Select Differentiated Genres

- Next, select some well differentiated genres to test the improvement to a baseline of similar genres
- Selecting tango, black metal, comedy, and study

```
differentiated_genre_list = ['tango', 'black-metal', 'comedy', 'study']
diff_genres_df = genreDF.loc[genreDF['track_genre'].isin(differentiated_genre_list)]

sns.pairplot(data=diff_genres_df, hue='track_genre')
```

# Pairplot for Different Genres

- Not as much overlap as previous pairplot
- Almost every diagonal graph has good differentiation
- From top to bottom the features are: duration, danceability, energy, loudness, speechiness, acousticness, instrumentalness, liveness, valence, and tempo



# K-Nearest Neighbors on Less Genres

- Very promising results for both analyses.
- Even the similar genres predicted with about 0.60 average precision and recall.
- The differentiated genres predicted extremely well with 0.95 average.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| club | 0.60 | 0.54 | 0.57 | 213 |
| electronic | 0.48 | 0.46 | 0.47 | 198 |
| party | 0.68 | 0.80 | 0.74 | 205 |
| pop | 0.63 | 0.61 | 0.62 | 184 |
| accuracy | | | 0.60 | 800 |
| macro avg | 0.60 | 0.60 | 0.60 | 800 |
| weighted avg | 0.60 | 0.60 | 0.60 | 800 |

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| black-metal | 0.99 | 0.98 | 0.99 | 209 |
| comedy | 0.97 | 0.94 | 0.95 | 186 |
| study | 0.93 | 0.93 | 0.93 | 192 |
| tango | 0.91 | 0.94 | 0.92 | 213 |
| accuracy | | | 0.95 | 800 |
| macro avg | 0.95 | 0.95 | 0.95 | 800 |
| weighted avg | 0.95 | 0.95 | 0.95 | 800 |

# Takeaways from Analysis on Genre

- The problem with the larger dataset is likely a big overlap between several genres in which many of the features are nearly identical
- Hard to tell the difference between "rock" and "rock-n-roll" or "metal" and "heavy-metal"
- Perhaps there are missing features that could describe the subtle differences in music
- Clustering genres together could likely produce a better result on the entire dataset worth of genres
- Future analysis could include reducing genres to similar characteristics for better predictability

# References and Links


- Automatic Music Popularity Prediction System
  - Stanford University. (2015). Automatic Music Popularity Prediction System: https://cs229.stanford.edu/proj2015/140_report.pdf
- Big Data Could Help Predict the Next Chart-Topping Hit
  - Carleton University. (2019, February 1). Big Data Could Help Predict the Next Chart-Topping Hit [News article]. https://newsroom.carleton.ca/story/big-data-predict-song-popularity/
- Top Spotify Songs 2023
  - Nelsiriyawithana, Y. (n.d.). Top Spotify Songs 2023. [Dataset]. https://www.kaggle.com/datasets/nelgiriyewithana/top-spotify-songs-2023
- Spotify Tracks Dataset
  - Maharishi Pandya. (n.d.). Spotify Tracks Dataset. [Dataset]. https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset
- Git: https://github.com/joey-beightol/spotify_stream_analysis