# Spotify Song Popularity Prediction

**Using Random Forest Classifier and Logistic Regression!**

*Group 3 - D001,D004,D007,D018,D020*

# Problem Statement

Spotify has become a dominant force in the music industry, with artists and record labels eager to understand the factors that drive a song's popularity. Spotify would like to stay competitive by being able to predict which songs are going to be popular ahead of time so that they can curate even better playlists and sign deals with up-and-coming artists to have exclusivity on their content. This would not only help retain the current subscribers but also help market the platform to new subscribers as well.

Play Follow

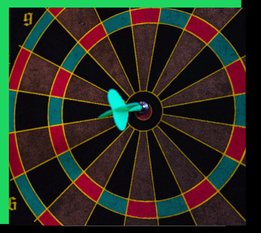
# Our Objectives

**Goal 01**

*Develop a machine learning model to accurately predict whether a song will be popular on Spotify.*

*Evaluate different models and identify key attributes that determine a song's popularity.*

**Goal 02**



# Our Dataset

There are 3 datasets used : The main one (approx. 232,000 tracks), top 50 and top 100 tracks of 2019. The variables are defined by Spotify's API which are defined as follows :

- Genre: Musical genre of the track.
- Artist Name: Name of the artist.
- Track Name: Title of the track.
- Track ID: Unique identifier for the track.
- Popularity: Popularity score (0 to 100).Calculated based on the total number of recent plays and their recency, with higher scores indicating more current popularity
- Acousticness: Confidence measure of the track being acoustic (0.0 to 1.0).
- Danceability: Suitability of the track for dancing (0.0 to 1.0).
- Duration (ms): Length of the track in milliseconds.
- Energy: Measure of intensity and activity (0.0 to 1.0).
- Instrumentalness: Likelihood of the track being instrumental (0.0 to 1.0).
- Key: Musical key of the track.
- Liveness: Presence of an audience (0.0 to 1.0).
- Loudness: Overall loudness in decibels (dB).
- Mode: Modality of the track (Major or Minor).
- Speechiness: Detection of spoken words (0.0 to 1.0).
- Tempo: Estimated tempo in beats per minute (BPM).
- Time Signature: Shows how many beats are in each measure of the track
- Valence: Musical positiveness (0.0 to 1.0).

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations

# EDA

*Ft. Raizel*

# EDA



```
df.describe()
```

| | popularity | acousticness | danceability | duration_ms | energy | instrumentalness | liveness | loudness | speechiness | tempo | valence |
|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 232725.000000 | 232725.000000 | 232725.000000 | 2.327250e+05 | 232725.000000 | 232725.000000 | 232725.000000 | 232725.000000 | 232725.000000 | 232725.000000 | 232725.000000 |
| mean | 41.127502 | 0.368560 | 0.554364 | 2.351223e+05 | 0.570958 | 0.148301 | 0.215009 | -9.569885 | 0.120765 | 117.666585 | 0.454917 |
| std | 18.189948 | 0.354768 | 0.185608 | 1.189359e+05 | 0.263456 | 0.302768 | 0.198273 | 5.998204 | 0.185518 | 30.898907 | 0.260065 |
| min | 0.000000 | 0.000000 | 0.056900 | 1.538700e+04 | 0.000020 | 0.000000 | 0.009670 | -52.457000 | 0.022200 | 30.379000 | 0.000000 |
| 25% | 29.000000 | 0.037600 | 0.435000 | 1.828570e+05 | 0.385000 | 0.000000 | 0.097400 | -11.771000 | 0.036700 | 92.959000 | 0.237000 |
| 50% | 43.000000 | 0.232000 | 0.571000 | 2.204270e+05 | 0.605000 | 0.000044 | 0.128000 | -7.762000 | 0.050100 | 115.778000 | 0.444000 |
| 75% | 55.000000 | 0.722000 | 0.692000 | 2.657680e+05 | 0.787000 | 0.035800 | 0.264000 | -5.501000 | 0.105000 | 139.054000 | 0.660000 |
| max | 100.000000 | 0.996000 | 0.989000 | 5.552917e+06 | 0.999000 | 0.999000 | 1.000000 | 3.744000 | 0.967000 | 242.903000 | 1.000000 |

## Checking Datatypes

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 232725 entries, 0 to 232724
Data columns (total 18 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   genre             232725 non-null  object
 1   artist_name       232725 non-null  object
 2   track_name        232725 non-null  object
 3   track_id          232725 non-null  object
 4   popularity        232725 non-null  int64
 5   acousticness      232725 non-null  float64
 6   danceability      232725 non-null  float64
 7   duration_ms       232725 non-null  int64
 8   energy            232725 non-null  float64
 9   instrumentalness  232725 non-null  float64
 10  key               232725 non-null  object
 11  liveness          232725 non-null  float64
 12  loudness          232725 non-null  float64
 13  mode              232725 non-null  object
 14  speechiness       232725 non-null  float64
 15  tempo             232725 non-null  float64
 16  time_signature    232725 non-null  object
 17  valence           232725 non-null  float64
dtypes: float64(9), int64(2), object(7)
```

## Checking null values

In [9]:

```
#checking for missing values
df.isna().sum()
```

Out[9]:

```
genre             0
artist_name       0
track_name        0
track_id          0
popularity        0
acousticness      0
danceability      0
duration_ms       0
energy            0
instrumentalness  0
key               0
liveness          0
loudness          0
mode              0
speechiness       0
tempo             0
time_signature    0
valence           0
dtype: int64
```

We don't have any missing values in our columns so we will move onto check for duplicated rows.

# EDA



## Checking Duplicated values

```
In [12]: df[df['track_id'].duplicated()]
```

Out[12]:

| | genre | artist_name | track_name | track_id | popularity | acousticness | danceability | duration_ms | energy | instrumentalness | key | liveness | loudness | mode | speechiness | tempo |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1348** | Alternative | Doja Cat | Go To Town | 6iOvnACn4ChlAw4lWUU4dd | 64 | 0.07160 | 0.710 | 217813 | 0.710 | 0.000001 | C | 0.2060 | -2.474 | Major | 0.0579 | 169.944 |
| **1385** | Alternative | Frank Ocean | Seigfried | 1BViPjTT585XAhkUUrkts0 | 61 | 0.97500 | 0.377 | 334570 | 0.255 | 0.000208 | E | 0.1020 | -11.165 | Minor | 0.0387 | 125.004 |
| **1452** | Alternative | Frank Ocean | Bad Religion | 2pMPWE7PJH1PizfgGRMnR9 | 56 | 0.77900 | 0.276 | 175453 | 0.358 | 0.000003 | A | 0.0728 | -7.684 | Major | 0.0443 | 81.977 |
| **1554** | Alternative | Steve Lacy | Some | 4riDfclV7kPDT9D58FpmHd | 58 | 0.00548 | 0.784 | 118393 | 0.554 | 0.254000 | G | 0.0995 | -6.417 | Major | 0.0300 | 104.010 |
| **1634** | Alternative | tobi lou | Buff Baby | 1F1QmI8TMHir9SUFrooq5F | 59 | 0.19000 | 0.736 | 215385 | 0.643 | 0.000000 | F | 0.1060 | -8.636 | Major | 0.0461 | 156.002 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **232715** | Soul | Emily King | Down | 5cA0vB8c9FMOVDWyJHgf26 | 42 | 0.55000 | 0.394 | 281853 | 0.346 | 0.000002 | E | 0.1290 | -13.617 | Major | 0.0635 | 90.831 |
| **232718** | Soul | Muddy Waters | I Just Want To Make Love To You - Electric Mud... | 2HFczeynfKGiM9KF2z2K7K | 43 | 0.01360 | 0.294 | 258267 | 0.739 | 0.004820 | C | 0.1380 | -7.167 | Major | 0.0434 | 176.402 |
| **232720** | Soul | Slave | Son Of Slide | 2XGLdVl7lGeq8ksM6Al7jT | 39 | 0.00384 | 0.687 | 326240 | 0.714 | 0.544000 | D | 0.0845 | -10.626 | Major | 0.0316 | 115.542 |
| **232722** | Soul | Muddy Waters | (I'm Your) Hoochie Coochie Man | 2ziWXUmQLrXTiYjCg2fZ2t | 47 | 0.90100 | 0.517 | 166960 | 0.419 | 0.000000 | D | 0.0945 | -8.282 | Major | 0.1480 | 84.135 |
| **232723** | Soul | R.LUM.R | With My Words | 6EFsue2YbIG4Qkq8Zr9Rir | 44 | 0.26200 | 0.745 | 222442 | 0.704 | 0.000000 | A | 0.3330 | -7.137 | Major | 0.1460 | 100.031 |

55951 rows × 18 columns

**We have 55,951 duplicated rows that we need to address. Before we can address these duplications though we need to see what the cause of the duplicates are.**

# EDA

| | genre | artist_name | track_name | track_id | popularity | acousticness | danceability | duration_ms | energy | ins |
|---|---|---|---|---|---|---|---|---|---|---|
| **48555** | Blues | Muddy Waters | I Just Want To Make Love To You - Electric Mud... | 2HFczeynfKGiM9KF2z2K7K | 35 | 0.0136 | 0.294 | 258267 | 0.739 | |
| **232718** | Soul | Muddy Waters | I Just Want To Make Love To You - Electric Mud | 2HFczeynfKGiM9KF2z2K7K | 43 | 0.0136 | 0.294 | 258267 | 0.739 | |

In [16]:
```
#generating a list with the genre names
genre_list = list(df['genre'].unique())
```

In [17]:
```
#creating the genre columns using the genre list
for genre in genre_list:
    df[genre] = (df['genre']==genre).astype('int')
```

In [18]:
```
#grouping by track_id number to get rid of duplicates and keeping the maximum values in each column.
df=df.groupby(['track_id']).max()
```

In [19]:
```
#removing redundant genre column
df.drop('genre', axis=1, inplace=True)
df.head()
```

Out[19]:

| track_id | artist_name | track_name | popularity | acousticness | danceability | duration_ms | energy | instrumentalness | key | liveness | ... | Pop | Reggae | Reggaeton | Jazz | Rock | Ska | Comedy | S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00021Wy6AyMbLP2tqij86e | Capcom Sound Team | Zangief's Theme | 13 | 0.234 | 0.617 | 169173 | 0.862 | 0.976000 | G | 0.1410 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 000CzNKC8PEt1yC3L8dqwV | Henri Salvador | Coeur Brisé à Prendre - Remastered | 5 | 0.249 | 0.518 | 130653 | 0.805 | 0.000000 | F | 0.3330 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 000DfZJww8KiixTKuk9usJ | Mike Love | Earthlings | 30 | 0.366 | 0.631 | 357573 | 0.513 | 0.000004 | D | 0.1090 | ... | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |
| 000EWWBkYaREzsBplYjUag | Don Philippe | Fewerdolr | 39 | 0.815 | 0.768 | 104924 | 0.137 | 0.922000 | C# | 0.1130 | ... | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 000xQL6tZNLJzIrIgxqSl | ZAYN | Still Got Time | 70 | 0.131 | 0.748 | 188491 | 0.627 | 0.000000 | G | 0.0852 | ... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |

5 rows × 42 columns

In [20]:
```
#verifying that duplicates have been eliminated
df[df.index =='6iOvnACn4ChlAw4lWUU4dd']
```

Out[20]:

| track_id | artist_name | track_name | popularity | acousticness | danceability | duration_ms | energy | instrumentalness | key | liveness | ... | Pop | Reggae | Reggaeton | Jazz | Rock | Ska | Comedy | So |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6iOvnACn4ChlAw4lWUU4dd | Doja Cat | Go To Town | 64 | 0.0716 | 0.71 | 217813 | 0.71 | 0.000001 | C | 0.206 | ... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |

1 rows × 42 columns

**The attributes of duplicated songs are mostly the same, except for 'popularity' and 'genre'. 'Popularity' can be aggregated as a numerical value, while 'genre' can be handled by creating separate columns for each genre with binary values indicating song genre presence.**

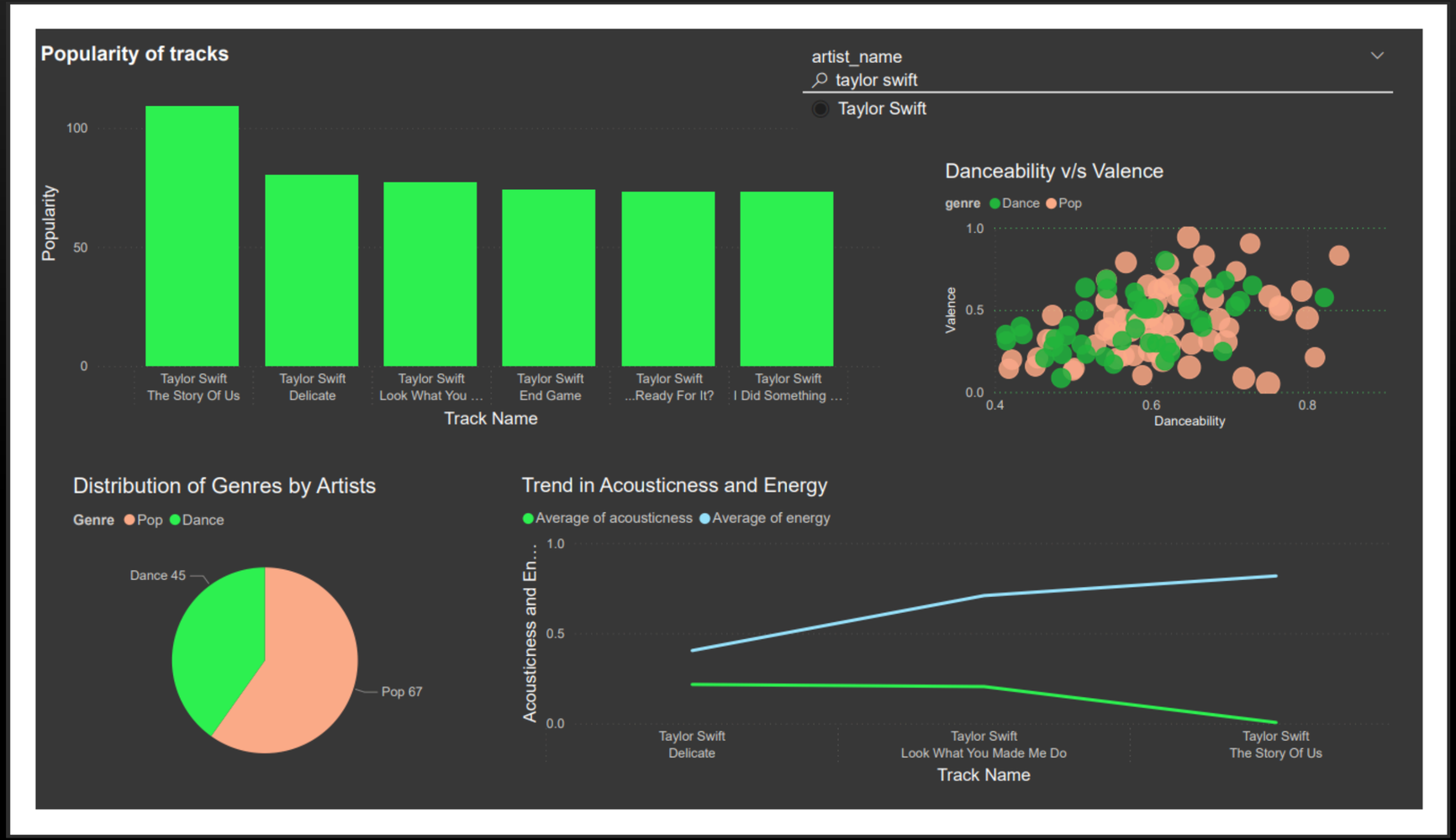
**The genre columns were created and duplicate values were merged, retaining the maximum value in each column to ensure consistency for the same track. This process involves selecting the highest value among duplicates, like choosing 42 over 15, 25, and 38 for a track's popularity score.**

# EDA (Power BI)

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations



# EDA (Power BI)

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

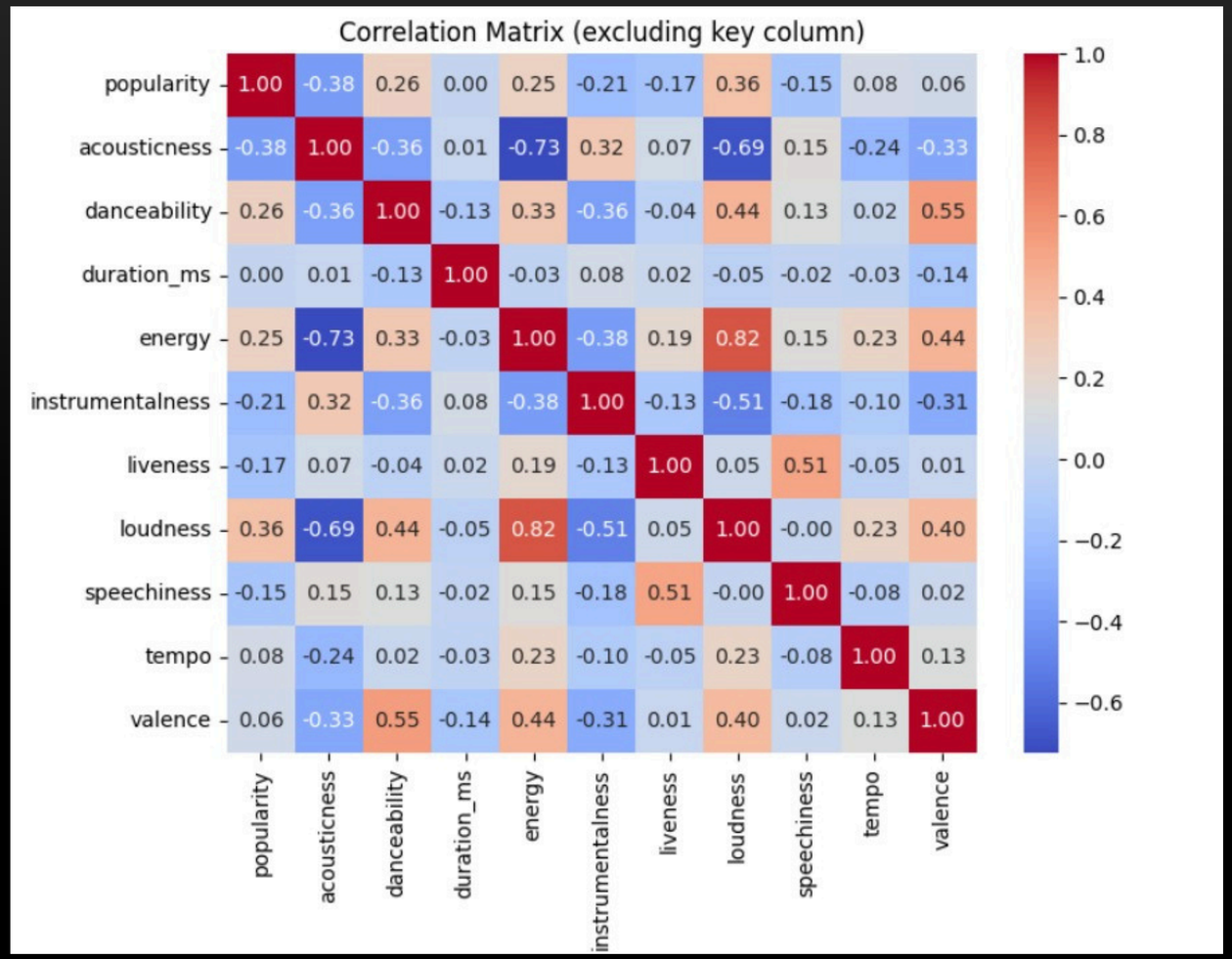
- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations

# EDA

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations




Factors that show a positive correlation with popularity include loudness, danceability, and energy. Conversely, acousticness, instrumentalness, and speechiness exhibit a negative correlation.

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations

# Feature Engineering

*Ft. Nimai*

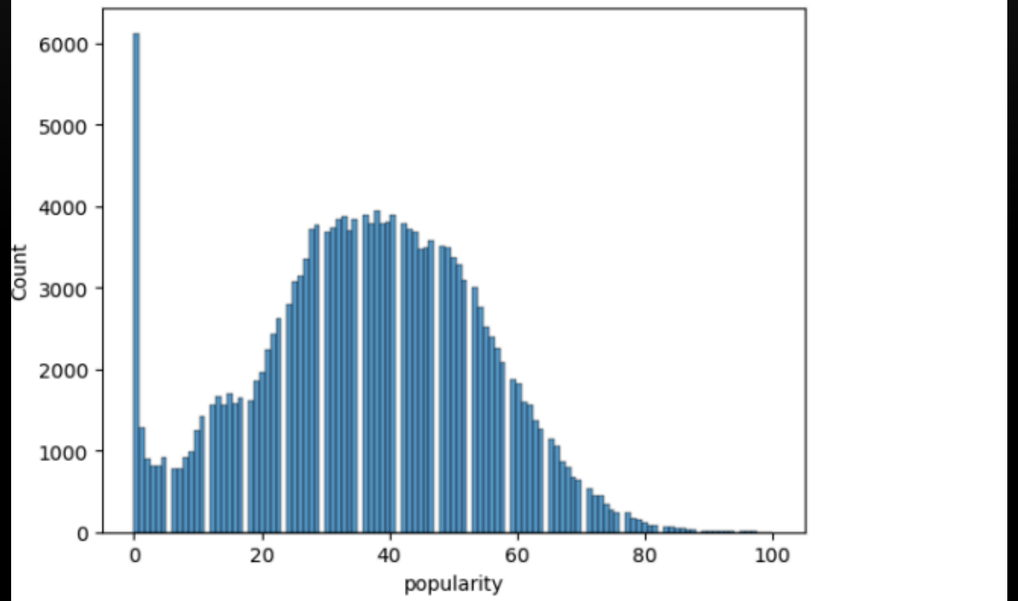
# Feature Engineering



**Given our objective to predict popular tracks, we must create a new column by converting the popularity column into binary form. To achieve this, we must establish a threshold for the popularity score. Songs above this threshold will be classified as "popular," while those below will be labeled "not popular."**

```
import matplotlib.pyplot as plt
import seaborn as sns

#creating a histogram to see distribution of popularity scores in the dataset.
sns.histplot(df['popularity'], bins='auto')
```

```
<Axes: xlabel='popularity', ylabel='Count'>
```




**Based on the histogram above, it is evident that we are dealing with a bimodal distribution. One peak is observed at 0, while the other appears to be around 40. To gain further insights into what is trending, we can examine the popularity scores of the Top 50 songs, sourced from 2019 like our main dataset to maintain consistency in the analysis.**

# Feature Engineering

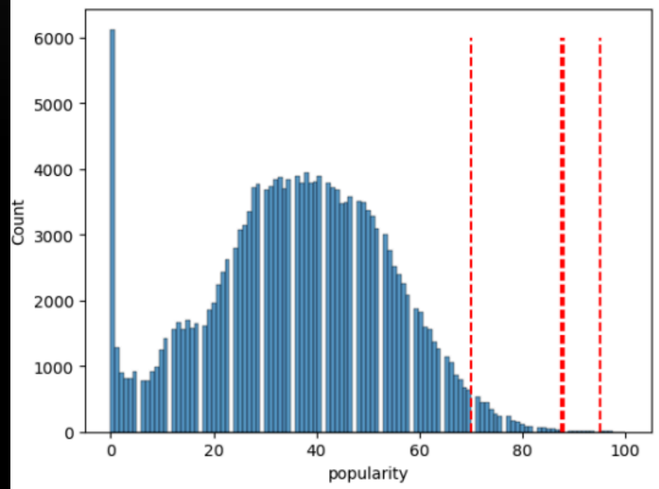


## Top 50 songs - 2019

```
#displaying stats information of Top 50 songs
df_50['Popularity'].describe()
```

```
count    50.000000
mean     87.500000
std       4.491489
min      70.000000
25%      86.000000
50%      88.000000
75%      90.750000
max      95.000000
Name: Popularity, dtype: float64
```

```
fig, ax = plt.subplots()
sns.histplot(df['popularity'], bins='auto', ax=ax)
stats=['mean', '50%', 'min', 'max']
for stat in stats:
    ax.vlines(x=df_50['Popularity'].describe()[stat], ymin=0, ymax=6000, linestyles='dashed', colors='red', label=stat)
```




**In the Top 50 songs, popularity scores ranged from 70 to 95, indicating that any song above 70 could potentially be considered popular. Using median or mean scores as cutoff points would overlook songs with values lower than 87.5 or 88, incorrectly labeling them as unpopular. Our previous project iteration used a cutoff point of 70 for popularity scores, which led to underperforming models due to the small sample size of 50 data points. To address this, we expanded our dataset to include more popular songs for a more reliable analysis.**

# Feature Engineering



**Top 100 songs - 2019**

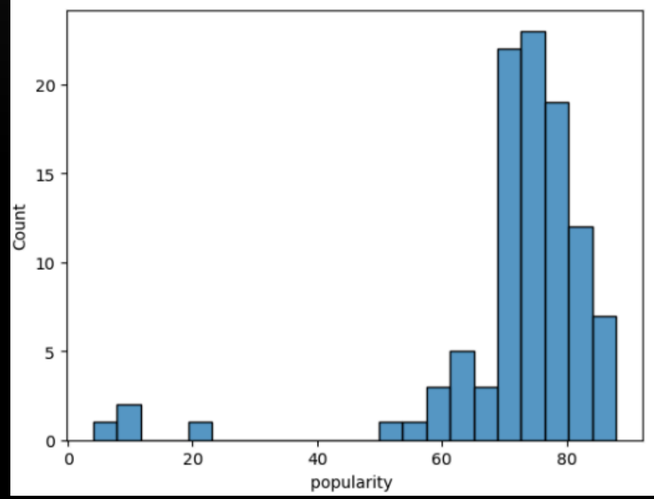
```
df_100 = pd.read_csv('spotify_top_100_2019.csv')
```

```
df_100['popularity '].describe()
```

```
count    100.000000
mean      72.020000
std       14.088451
min        4.000000
25%       70.000000
50%       74.500000
75%       79.000000
max       88.000000
Name: popularity , dtype: float64
```

```
fig, ax = plt.subplots()
sns.histplot(df_100['popularity '], bins='auto', ax=ax)
```

```
<Axes: xlabel='popularity ', ylabel='Count'>
```




```
#Outlier Removal with the IQR method

def find_outliers_IQR(data, return_limits = False):
    df_b=data
    res= df_b.describe()

    IQR = res['75%'] -  res['25%']
    lower_limit = res['25%'] - 1.5*IQR
    upper_limit = res['75%'] + 1.5*IQR

    if return_limits:
        return lower_limit, upper_limit

    else:
        idx_outs = (df_b>upper_limit) | (df_b<lower_limit)
        return idx_outs
```

```
#removing outliers from the popularity column
df_100 = df_100[find_outliers_IQR(df_100['popularity '])==False]
#displaying minimum & maxium values in popularity column
print("Minimum:", df_100['popularity '].min())
print("Maximum:", df_100['popularity '].max())
```

```
Minimum: 58
Maximum: 88
```

**As we imagined the scores within the range 0-25 seem like outliers. We can remove outliers from this dataset with the IQR method to get a better perspective on the data.**

# Feature Engineering

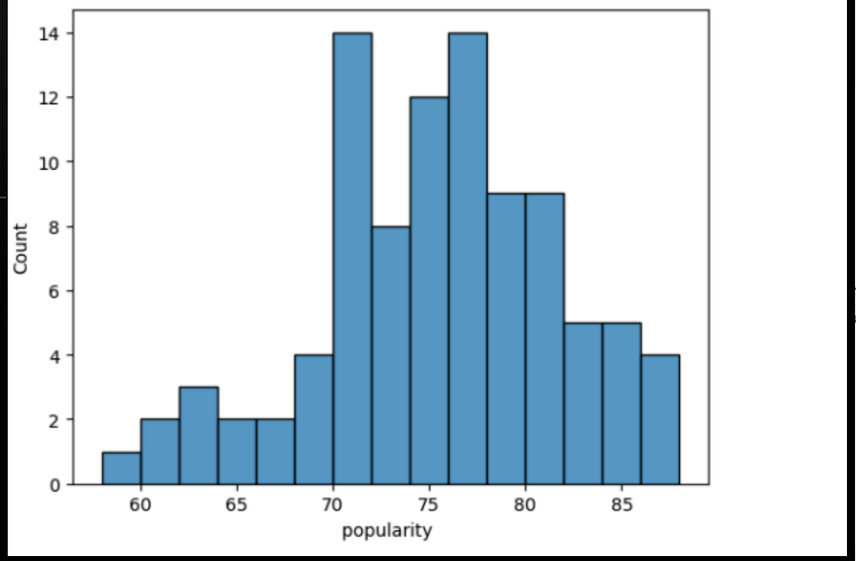


```
#removing outliers from the popularity column
df_100 = df_100[find_outliers_IQR(df_100['popularity '])==False]
#displaying minimum & maxium values in popularity column
print("Minimum:", df_100['popularity '].min())
print("Maximum:", df_100['popularity '].max())
```

```
Minimum: 58
Maximum: 88
```

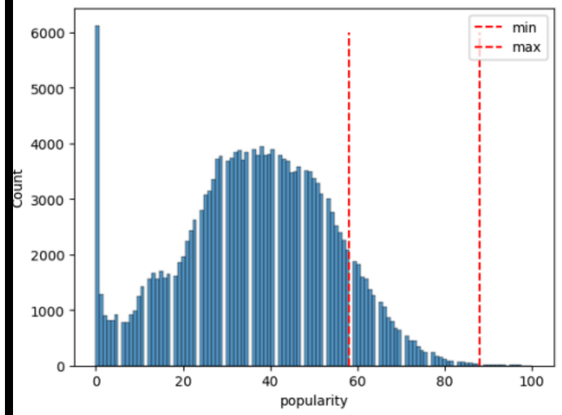
```
fig, ax = plt.subplots()
sns.histplot(df_100['popularity '], bins=15, ax=ax)
```

```
<Axes: xlabel='popularity ', ylabel='Count'>
```




```
#visualizing the min and max popularity scores on the overall dataset histogram
fig, ax = plt.subplots()
sns.histplot(df['popularity'], bins='auto', ax=ax)
ax.vlines(x=df_100['popularity '].min(), ymin=0, ymax=6000, linestyles='dashed', colors='red', label='min')
ax.vlines(x=df_100['popularity '].max(), ymin=0, ymax=6000, linestyles='dashed', colors='red', label='max')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1e0b4070e60>
```




```
#creating is_popular column with our cutoff point
df['is_popular']=(df['popularity']>=58).astype('int')
df.head()
```

| track_id | artist_name | track_name | popularity | acousticness | danceability | duration_ms | energy | instrumentalness | key | liveness | ... | Reggae | Reggaeton | Jazz | Rock | Ska | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00021Wy6AyMbLP2tqij86e | Capcom Sound Team | Zangief's Theme | 13 | 0.234 | 0.617 | 169173 | 0.862 | 0.976000 | G | 0.1410 | ... | 0 | 0 | 0 | 0 | 0 | |
| 000CzNKC8PEt1yC3L8dqwV | Henri Salvador | Coeur Brisé à Prendre - Remastered | 5 | 0.249 | 0.518 | 130653 | 0.805 | 0.000000 | F | 0.3330 | ... | 0 | 0 | 0 | 0 | 0 | |
| 000DfZJww8KiixTKuk9usJ | Mike Love | Earthlings | 30 | 0.366 | 0.631 | 357573 | 0.513 | 0.000004 | D | 0.1090 | ... | 1 | 0 | 0 | 0 | 0 | |

**As anticipated, the top 100 songs exhibit a broader range, resulting in a lower popularity score threshold than the top 50 songs. To define a song as popular, we will consider it worthy of being in the Top 100, setting our cutoff point at 58.**

# Feature Engineering

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations

```
#dropping popularity score column since we will not be using it
df.drop(['popularity', 'artist_name', 'track_name'], axis=1, inplace=True)
df.head()
```

| track_id | acousticness | danceability | duration_ms | energy | instrumentalness | key | liveness | loudness | mode | speechiness | ... | Reggae | Reggaeton | Jazz | Rock | Ska | Comedy | Soul | Soundtr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00021Wy6AyMbLP2tqij86e | 0.234 | 0.617 | 169173 | 0.862 | 0.976000 | G | 0.1410 | -12.855 | Major | 0.0514 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 000CzNKC8PEt1yC3L8dqwV | 0.249 | 0.518 | 130653 | 0.805 | 0.000000 | F | 0.3330 | -6.248 | Major | 0.0407 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 000DfZJww8KiixTKuk9usJ | 0.366 | 0.631 | 357573 | 0.513 | 0.000004 | D | 0.1090 | -6.376 | Major | 0.0293 | ... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 000EWWBkYaREzsBplYjUag | 0.815 | 0.768 | 104924 | 0.137 | 0.922000 | C# | 0.1130 | -13.284 | Minor | 0.0747 | ... | 0 | 0 | 1 | 0 | 0 | 0 | 0 | |
| 000xQL6tZNLJzIrtIgxqSl | 0.131 | 0.748 | 188491 | 0.627 | 0.000000 | G | 0.0852 | -6.029 | Major | 0.0644 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

5 rows × 40 columns

**We have eliminated the popularity scores since we have transformed that column into binary form. Additionally, 'artist_name' and 'track_name' are being excluded from our analysis as we aim to concentrate on the song's composition rather than its performer or title. Our goal is to identify songs with the potential to become popular regardless of the artist's name, enabling us to uncover tracks by up-and-coming artists.**

# Train-Test Split & One-hot Encoding

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations

```
#splitting the data to training and test sets in order to be able to measure performance
from sklearn.model_selection import train_test_split
y=df['is_popular']
X=df.drop('is_popular',axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)
```

```
#Check to see how many more columns we will be creating by OHE the cat_cols.
df.nunique()
```

```
acousticness          4734
danceability          1295
duration_ms          70749
energy                2517
instrumentalness      5400
key                     12
liveness              1732
loudness             27923
mode                     2
speechiness           1641
tempo                78509
time_signature           5
valence               1692
Movie                    2
R&B                      2
A Capella                2
Alternative              2
Country                  2
Dance                    2
Electronic               2
Anime                    2
Folk                     2
Blues                    2
Opera                    2
Hip-Hop                  2
Children's Music         2
Rap                      2
Indie                    2
Classical                2
Pop                      2
Reggae                   2
Reggaeton                2
Jazz                     2
Rock                     2
Ska                      2
Comedy                   2
Soul                     2
Soundtrack               2
World                    2
is_popular               2
dtype: int64
```

```
#define categorical columns
cat_cols = ['key', 'mode', 'time_signature']
```

**We will be creating 2 (mode) + 5 (time_signature) + key (12) - 3 (drop_first) = 16 columns.**

```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

# Define the OneHotEncoder with desired parameters
encoder = OneHotEncoder(sparse_output=False, drop='first')

# Training set
data_ohe_train = encoder.fit_transform(X_train[cat_cols])
df_ohe_train = pd.DataFrame(data_ohe_train, columns=encoder.get_feature_names_out(cat_cols), index=X_train.index)

# Testing set
data_ohe_test = encoder.transform(X_test[cat_cols])
df_ohe_test = pd.DataFrame(data_ohe_test, columns=encoder.get_feature_names_out(cat_cols), index=X_test.index)
```

```
pd.set_option("display.max_columns", None)
df_ohe_train
```

| track_id | key_A# | key_B | key_C | key_C# | key_D | key_D# | key_E | key_F | key_F# | key_G | key_G# | mode_Minor | time_signa |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5SbN8lXhPno4BRrFb9yqkF | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| 34n3eoeqVaXAgtMqy8Ncyz | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| 3QbxHo2OTwBVDZbaJaMniP | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| 6Y8aA0SWBMB5XTZIXlDpYv | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | |
| 16h3GCdEJ9lgiOyox4LJQA | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | |

# Train-Test Split & One-hot Encoding

```
#merging OHE columns with numerical columns
X_train = pd.concat([X_train.drop(cat_cols, axis=1), df_ohe_train], axis=1)
X_test = pd.concat([X_test.drop(cat_cols, axis=1), df_ohe_test], axis=1)
X_train.tail()
```

| track_id | acousticness | danceability | duration_ms | energy | instrumentalness | liveness | loudness | speechiness | tempo | valence | Movie | R&B | A Capella | Alternative | Country | Dance | Electron |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5H23I3K3TUXQMsLg2FzCiY | 0.23100 | 0.461 | 326680 | 0.252 | 0.000084 | 0.106 | -17.082 | 0.2740 | 74.768 | 0.481 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 4YnYtYWBmDM8YjfMMK0cqs | 0.00571 | 0.309 | 201947 | 0.820 | 0.000368 | 0.159 | -4.844 | 0.0636 | 174.762 | 0.399 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 5o5xTG5Mh3JAm2BZv4nOI7 | 0.01210 | 0.691 | 265587 | 0.834 | 0.001480 | 0.115 | -4.378 | 0.0432 | 129.982 | 0.330 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 6T1oL7ed1wUEqlCR1iCpIR | 0.35500 | 0.364 | 224387 | 0.733 | 0.000007 | 0.981 | -5.071 | 0.0299 | 145.394 | 0.244 | 0 | 0 | 0 | 0 | 1 | 0 | |
| 5MnEYPkZ5HC7BQ988kBKqp | 0.01190 | 0.478 | 211800 | 0.695 | 0.000000 | 0.119 | -5.923 | 0.0529 | 89.801 | 0.297 | 0 | 1 | 0 | 0 | 0 | 0 | |

**The dataset has been successfully divided into training and testing sets.**
**Now, we are all set for modeling!**

# Get ready for Modelling!

*Ft. Aamir*

# SMOTENC



```
#class imbalance percentages
y_train.value_counts(normalize=True)
```

```
is_popular
0    0.885503
1    0.114497
Name: proportion, dtype: float64
```

```
#looking at column names to extract categorical column indices for SMOTENC
X_train.columns
```

```
Index(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',
       'valence', 'Movie', 'R&B', 'A Capella', 'Alternative', 'Country',
       'Dance', 'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       'Children's Music', 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World', 'key_A#', 'key_B', 'key_C', 'key_C#', 'key_D', 'key_D#',
       'key_E', 'key_F', 'key_F#', 'key_G', 'key_G#', 'mode_Minor',
       'time_signature_01/4', 'time_signature_03/4', 'time_signature_04/4',
       'time_signature_05/4'],
      dtype='object')
```

```
#creating a list of categorical column indices
cat_cols = list(range(10, len(X_train.columns)))
X_train.columns[cat_cols]
```

```
Index(['Movie', 'R&B', 'A Capella', 'Alternative', 'Country', 'Dance',
       'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       'Children's Music', 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World', 'key_A#', 'key_B', 'key_C', 'key_C#', 'key_D', 'key_D#',
       'key_E', 'key_F', 'key_F#', 'key_G', 'key_G#', 'mode_Minor',
       'time_signature_01/4', 'time_signature_03/4', 'time_signature_04/4',
       'time_signature_05/4'],
      dtype='object')
```

```
#Using SMOTENC to address class imbalance. We are not using SMOTE since we have categorical columns.
from imblearn.over_sampling import SMOTE, SMOTENC

sm = SMOTENC(categorical_features=cat_cols, random_state=42)

X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train)
y_train_sm.value_counts(normalize=True)
```

```
is_popular
0    0.5
1    0.5
```

***SMOTENC*: Synthetic Minority Over-sampling Technique for Nominal and Continuous variables**

**To tackle class imbalance in our dataset, we applied SMOTENC, which generates synthetic samples specifically for the minority class. By identifying and preserving categorical features, SMOTENC creates new, balanced instances in the feature space. This approach ensures that the model isn't biased towards the majority class, leading to more accurate and reliable predictions**

# Confusion Matrix and ROC Curve

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

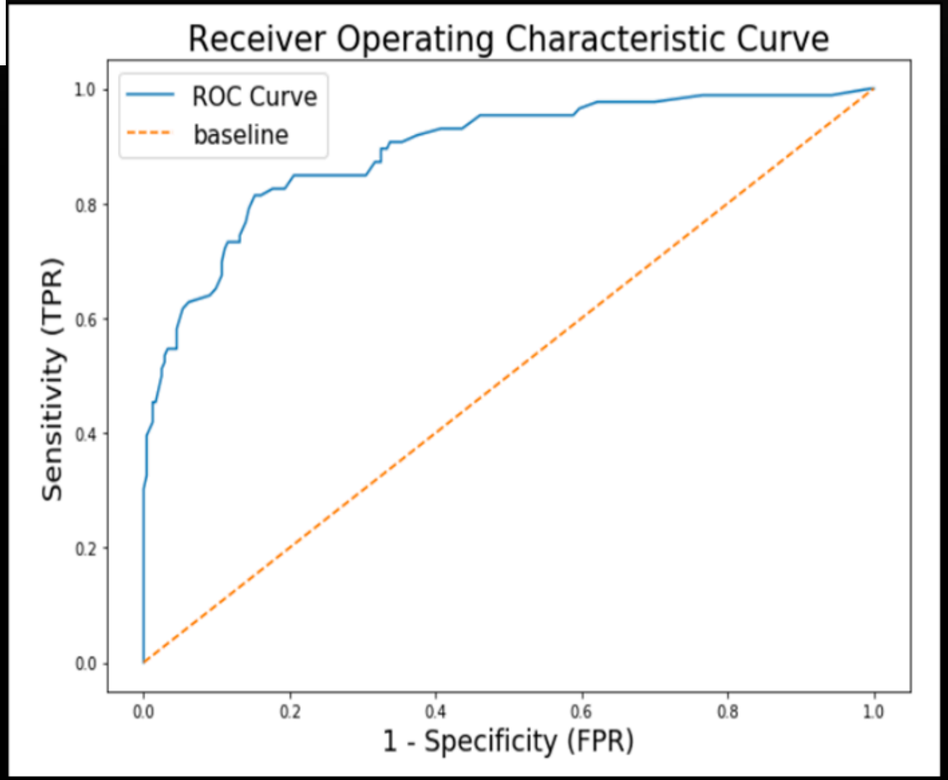
- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations

| | | Predicted: 0 | Predicted: 1 |
|---|---|---|---|
| Actual | 0 | TN | FP |
| | 1 | FN | TP |

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

The ROC curve evaluates a model's performance by balancing sensitivity and 1-specificity. It is recommended to assume p > 0.5 for plotting ROC. AUC is a key metric, indicating classification power – higher AUC signifies better model performance wrt classification



# FUNCTIONS

- Problem Statement
- Objective
- Our Dataset
- EDA
- Feature Engineering

---

- Train-Test Split & One hot encoding
- Get ready for modelling
- Random Forest Classifier
- Logistic Regression
- Feature Importance
- Insights
- Takeaways
- Recommendations

## *Classification Report Function*

```python
from sklearn.metrics import classification_report
from sklearn.metrics import ConfusionMatrixDisplay, RocCurveDisplay
import matplotlib.pyplot as plt

def classification(y_true, y_pred, X, clf):
    """This function shows the classification report,
    the confusion matrix as well as the ROC curve for evaluation of model quality.

    y_true: Correct y values, typically y_test that comes from the train_test_split performed at the beginning of model development.
    y_pred: Predicted y values by the model.
    clf: classifier model that was fit to training data.
    X: X_test values"""

    # Classification report
    print("CLASSIFICATION REPORT")
    print("-----------------------------------------")
    print(classification_report(y_true=y_true, y_pred=y_pred))

    # Creating a figure/axes for confusion matrix and ROC curve
    fig, ax = plt.subplots(ncols=2, figsize=(12, 5))

    # Plotting the normalized confusion matrix
    ConfusionMatrixDisplay.from_estimator(estimator=clf, X=X, y=y_true, cmap='Blues', normalize='true', ax=ax[0])
    ax[0].set_title("Normalized Confusion Matrix")

    # Plotting the ROC curve
    RocCurveDisplay.from_estimator(estimator=clf, X=X, y=y_true, ax=ax[1])
    ax[1].plot([0,1], [0,1], ls='--', color='orange', label='Random Chance')
    ax[1].set_title("ROC Curve")
    ax[1].legend(loc="lower right")

    plt.tight_layout()
    plt.show()
```

## *Result function*

```python
import pandas as pd
from sklearn.metrics import recall_score

def add_results(model_name, df, y_test, y_pred):
    # Create a new DataFrame with the results
    new_row = pd.DataFrame({
        'Model Name': [model_name],
        'Recall Score': [round(recall_score(y_test, y_pred), 2)]
    })

    # Append the new row to the existing results DataFrame using concat
    df = pd.concat([df, new_row], ignore_index=True)

    return df
```

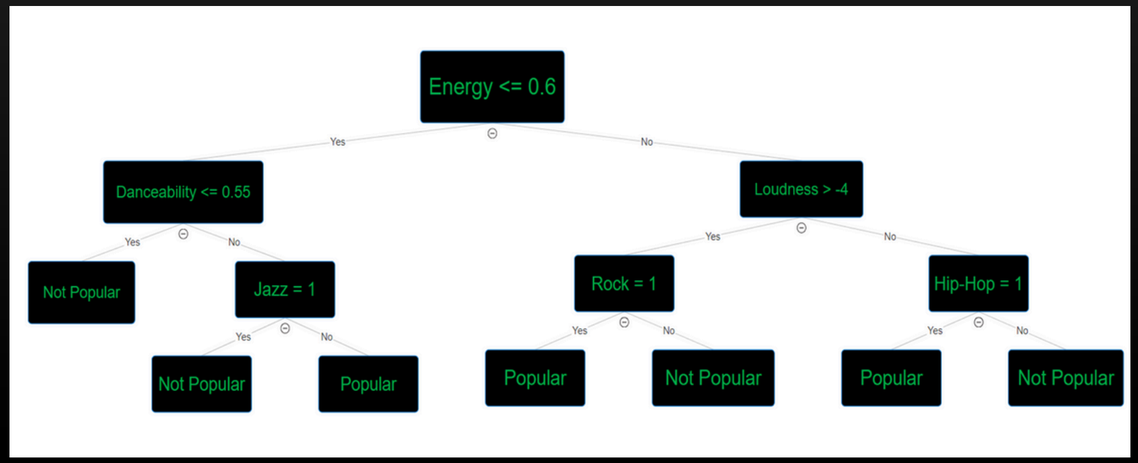
# What is a Random Forest

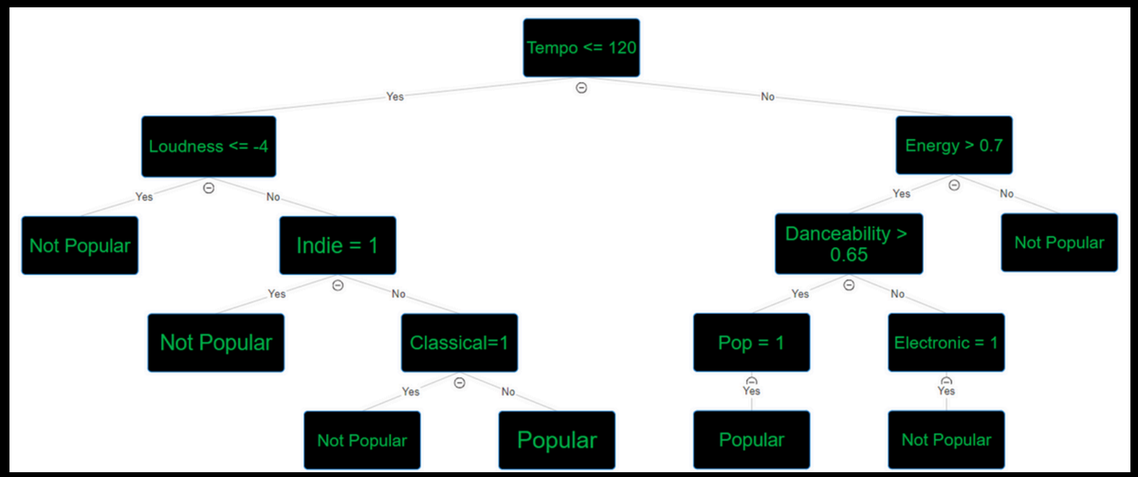
1. What is a Random Forest?
   • An ensemble learning method that builds multiple decision trees and merges their results to improve accuracy and reduce overfitting.
   • It can be used for classification, regression, and other prediction tasks.

2. How Does a Random Forest Work ?
   •For classification, the Random Forest picks the class that most decision trees agree on.

3. How is it Used in Predicting Song Popularity?
   •Random Forest predicts song popularity by training multiple decision trees on song features and then aggregating their predictions. It identifies important features and combines the insights from all trees to classify songs as popular or not.

# Example of Decision Trees



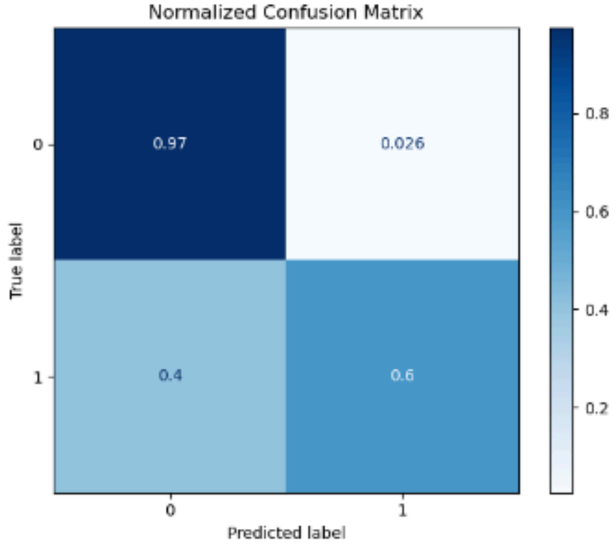
# Random Forest Classifier

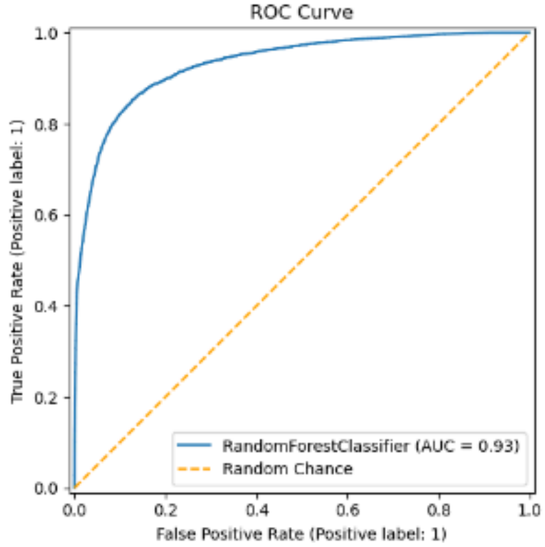


```python
from sklearn.model_selection import GridSearchCV

clf = RandomForestClassifier()
grid = {'criterion': ['gini', 'entropy'],
        'max_depth': [10, 20, None],
        'min_samples_leaf': [1, 2, 3]        }

gridsearch = GridSearchCV(estimator=clf, param_grid = grid, scoring='recall')

gridsearch.fit(X_train_sm,  y_train_sm)
gridsearch.best_params_
# Results: {'criterion': 'entropy', 'max_depth': None, 'min_samples_leaf': 2}
```

```python
clf_rf_tuned = RandomForestClassifier(criterion='entropy', max_depth=None,
                                      min_samples_leaf=2, class_weight='balanced',
                                      random_state=42)
clf_rf_tuned.fit(X_train_sm, y_train_sm)

y_pred = clf_rf_tuned.predict(X_test)
classification(y_test, y_pred, X_test, clf_rf_tuned)
```

```
CLASSIFICATION REPORT
--------------------------------------------
              precision    recall  f1-score   support

           0       0.95      0.97      0.96     47002
           1       0.75      0.60      0.66      6031

    accuracy                           0.93     53033
   macro avg       0.85      0.79      0.81     53033
weighted avg       0.93      0.93      0.93     53033
```




**Employing a grid search will help optimize the recall score. Prioritizing recall is key as our main focus is accurately identifying potentially popular songs, accepting a margin of error in selecting songs that may not gain popularity.**

**Recall Score is 0.6**

# Logistic Regression

*Ft. Senorita*

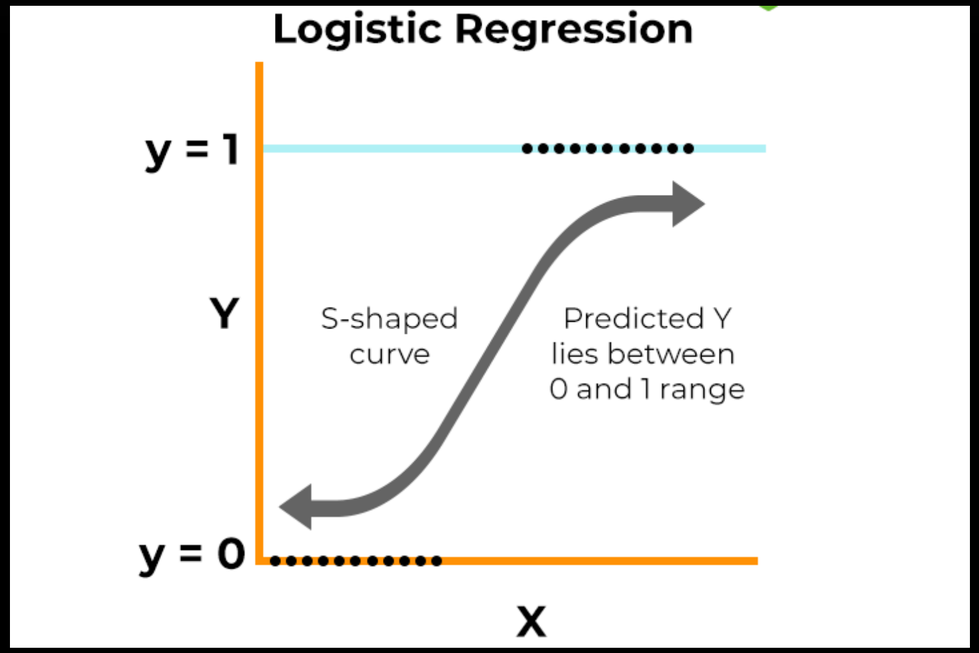
# Logistic Regression



**Logistic regression predicts the output of a categorical dependent variable. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.**

$$\hat{y} = b_1 \cdot x_1 + b_2 \cdot x_2 + \ldots + b_k \cdot x_k + a$$

$$f(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(b_1 \cdot x_1 + \ldots + b_k \cdot x_k + a)}}$$




**In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.**

# Logistic Regression



## Removing Outliers

```
#separating out the numerical columns for outlier removal
num_cols = ['acousticness', 'danceability', 'duration_ms', 'energy', 'instrumentalness',
            'liveness', 'loudness', 'speechiness', 'tempo', 'valence']
num_cols
```

```
['acousticness',
 'danceability',
 'duration_ms',
 'energy',
 'instrumentalness',
 'liveness',
 'loudness',
 'speechiness',
 'tempo',
 'valence']
```

```
#Concatenating the training and testing sets together for outlier removal
df_train = pd.concat([X_train, y_train], axis=1)
df_test = pd.concat([X_test, y_test], axis=1)
```

```
#finding and removing outliers based on X_train (df_train) to avoid data leakage

original_length_train = len(df_train)
original_length_test = len(df_test)

for col in num_cols:

    lower_limit, upper_limit = find_outliers_IQR(df_train[col], return_limits=True)

    df_train = df_train[(df_train[col]>lower_limit) & (df_train[col]<upper_limit)]
    df_test = df_test[(df_test[col]>lower_limit) & (df_test[col]<upper_limit)]

print(f'{original_length_train - len(df_train)} outliers removed from training set')
print(f'{original_length_test - len(df_test)} outliers removed from test set')
```

```
55567 outliers removed from training set
23796 outliers removed from test set
```

```
#Separating out the X and y values for training and test sets

y_train = df_train['is_popular']
X_train = df_train.drop('is_popular', axis=1)

y_test = df_test['is_popular']
X_test = df_test.drop('is_popular', axis=1)
```

**Since the Logistic Regression models are potentially sensitive to outliers and need scaled data we will need to process our data one more time to remove outliers and scale it**

# Logistic Regression



## *Addressing Class Imbalance with SMOTENC & Scaling the Data*

```
y_train.value_counts(normalize=True)
```

```
is_popular
0    0.842345
1    0.157655
Name: proportion, dtype: float64
```

```
X_train.columns
```

```
Index(['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',
       'valence', 'Movie', 'R&B', 'A Capella', 'Alternative', 'Country',
       'Dance', 'Electronic', 'Anime', 'Folk', 'Blues', 'Opera', 'Hip-Hop',
       'Children's Music', 'Rap', 'Indie', 'Classical', 'Pop', 'Reggae',
       'Reggaeton', 'Jazz', 'Rock', 'Ska', 'Comedy', 'Soul', 'Soundtrack',
       'World', 'key_A#', 'key_B', 'key_C', 'key_C#', 'key_D', 'key_D#',
       'key_E', 'key_F', 'key_F#', 'key_G', 'key_G#', 'mode_Minor',
       'time_signature_01/4', 'time_signature_03/4', 'time_signature_04/4',
       'time_signature_05/4'],
      dtype='object')
```

```
cat_cols = list(range(10,len(X_train.columns)))
```

```
sm = SMOTENC(categorical_features=cat_cols, random_state=42)

X_train_sm, y_train_sm = sm.fit_resample(X_train, y_train)
y_train_sm.value_counts(normalize=True)
```

```
is_popular
0    0.5
1    0.5
Name: proportion, dtype: float64
```

Scaling the data

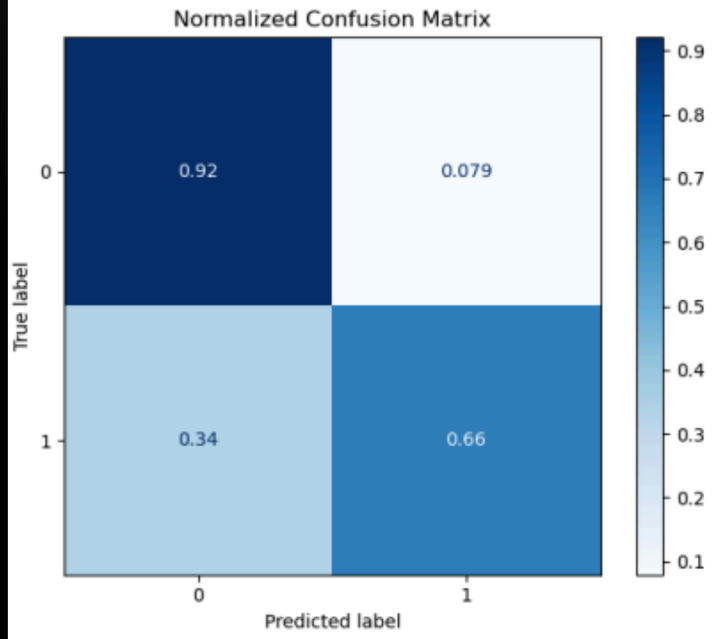
```
#Using Standard Scaler to scale the smote'd data
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_sm_sc = scaler.fit_transform(X_train_sm)
X_test_sc = scaler.transform(X_test)
```

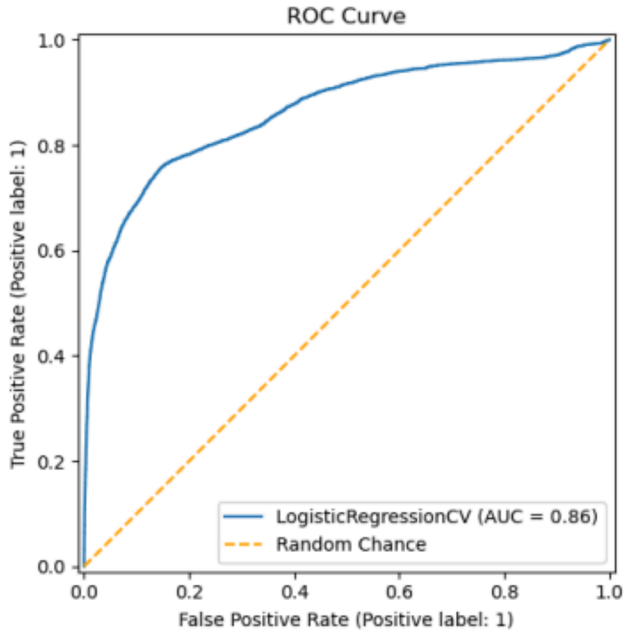
# Logistic Regression



```
from sklearn.linear_model import LogisticRegressionCV
clf_logregcv = LogisticRegressionCV(cv=5, random_state=42)
clf_logregcv.fit(X_train_sm_sc,  y_train_sm)
y_pred = clf_logregcv.predict(X_test_sc)
classification(y_test, y_pred, X_test_sc, clf_logregcv)
```

```
CLASSIFICATION REPORT
-----------------------------------------
              precision    recall  f1-score   support

           0       0.93      0.92      0.93     24588
           1       0.61      0.66      0.63      4649

    accuracy                           0.88     29237
   macro avg       0.77      0.79      0.78     29237
weighted avg       0.88      0.88      0.88     29237
```



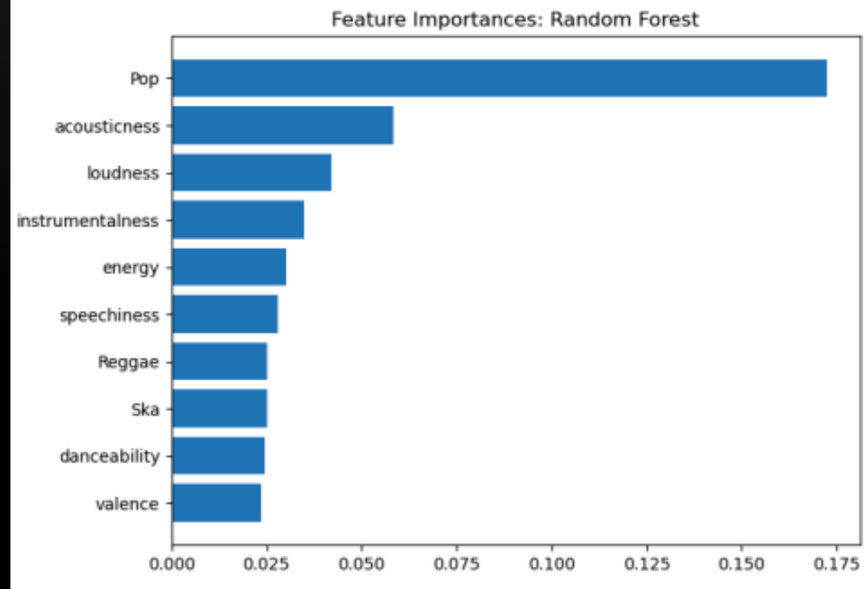

**Recall Score is 0.66**

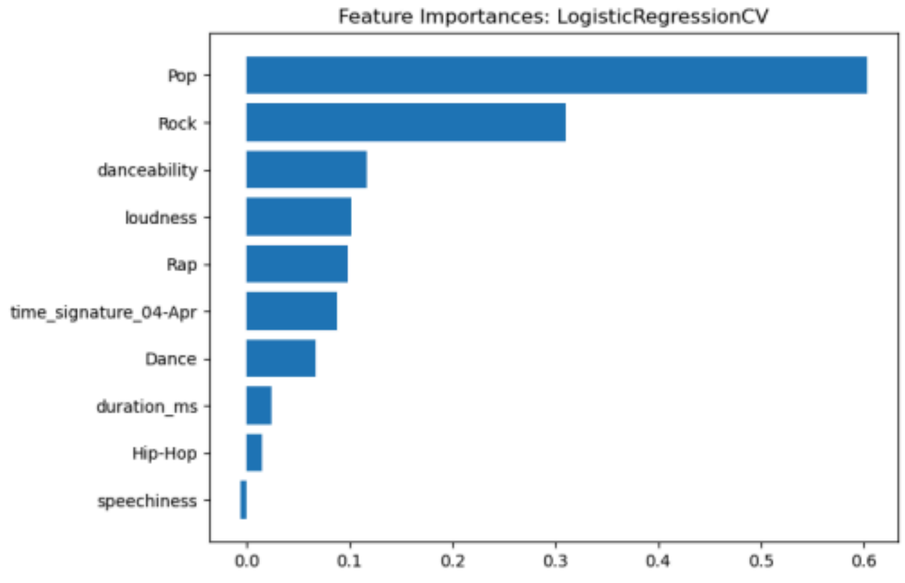
**Logistic Regression performs better!**

# Feature Importance Comparison



```
#plotting feature importances for all models for comparison

fig, ax = plt.subplots(ncols=2, figsize=(15,5))

rf_importances_df = rf_importances_df.sort_values(by='RF-Importance', ascending=True).tail(10)
ax[0].barh(rf_importances_df['RF-Attribute'], rf_importances_df['RF-Importance'])
ax[0].set_title('Feature Importances: Random Forest')

logregcv_importances_df = logregcv_importances_df.sort_values(by='LogReg-Importance', ascending=True).tail(10)
ax[1].barh(logregcv_importances_df['LogReg-Attribute'], logregcv_importances_df['LogReg-Importance'])
ax[1].set_title('Feature Importances: LogisticRegressionCV')
plt.tight_layout()
```



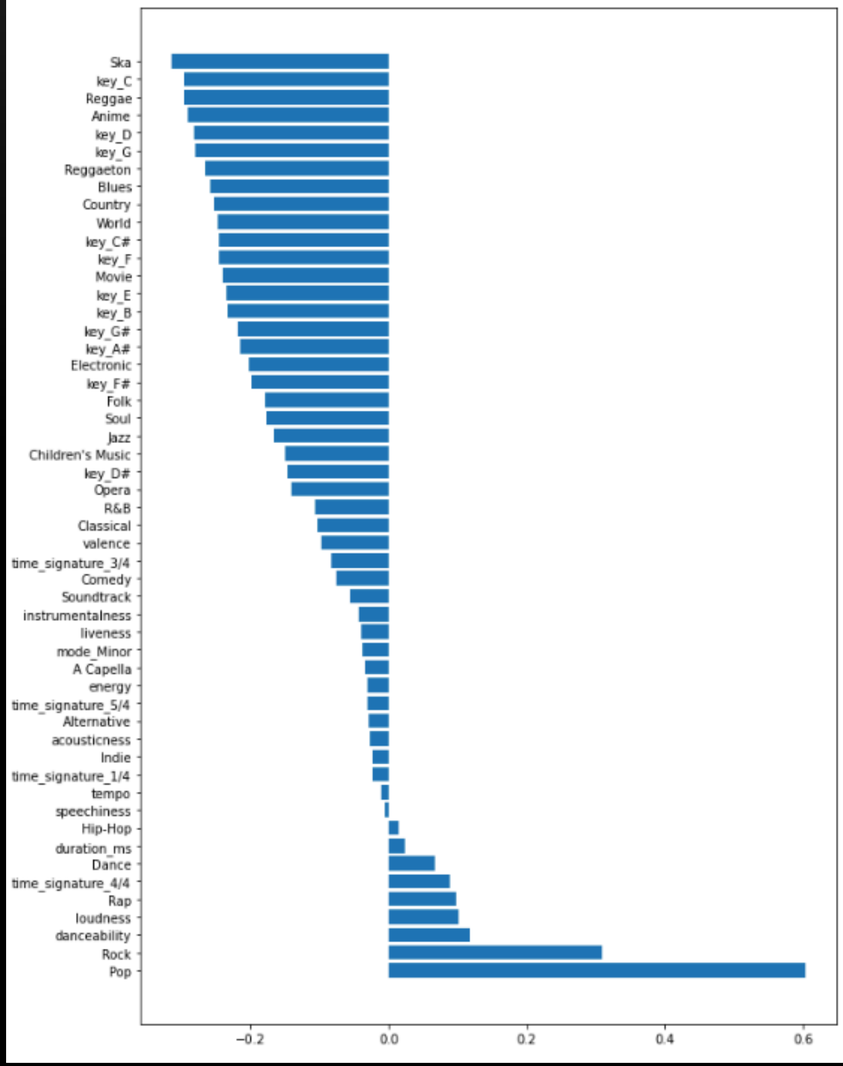

**Out of the two models we created, it is evident that a song's genre has the most significant influence on its popularity. Across all two models, songs categorized as Pop had the highest impact on their popularity, which aligns with the inherently popular nature of Pop music. Other attributes such as danceability, energy, various genres, and acousticness also significantly contribute to a song's popularity. Moving forward, we will delve into the complete range of feature importances in logistic regression for further analysis.**

# Feature Importance Comparison



```
logregcv_importances_df = pd.Series(clf_logregcv_tuned.coef_[0], index=X_train.columns).sort_values(ascending=False)
#parsing the series to a dataframe
logregcv_importances_df = logregcv_importances_df.reset_index()
logregcv_importances_df.columns = ['Attribute', 'Importance']

fig, ax = plt.subplots(figsize=(10,15))
ax.barh(logregcv_importances_df['Attribute'], logregcv_importances_df['Importance'])
```

```
<BarContainer object of 52 artists>
```




**Here we observe that attributes like 'Pop', 'Rock', and 'Danceability' had a positive impact on the prediction, whereas 'Ska', 'Anime', and 'key_G' had a negative influence. Moving forward, we will analyze our processed dataframe to investigate these characteristics in popular and unpopular songs to draw conclusions.**

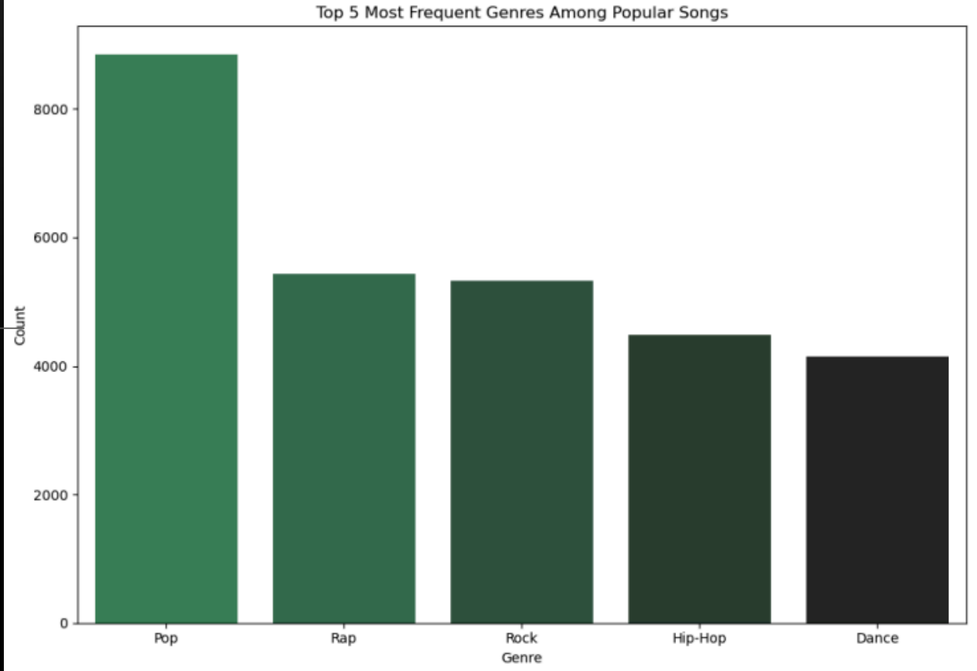
# Insights

*Ft. Jai*

# Insights

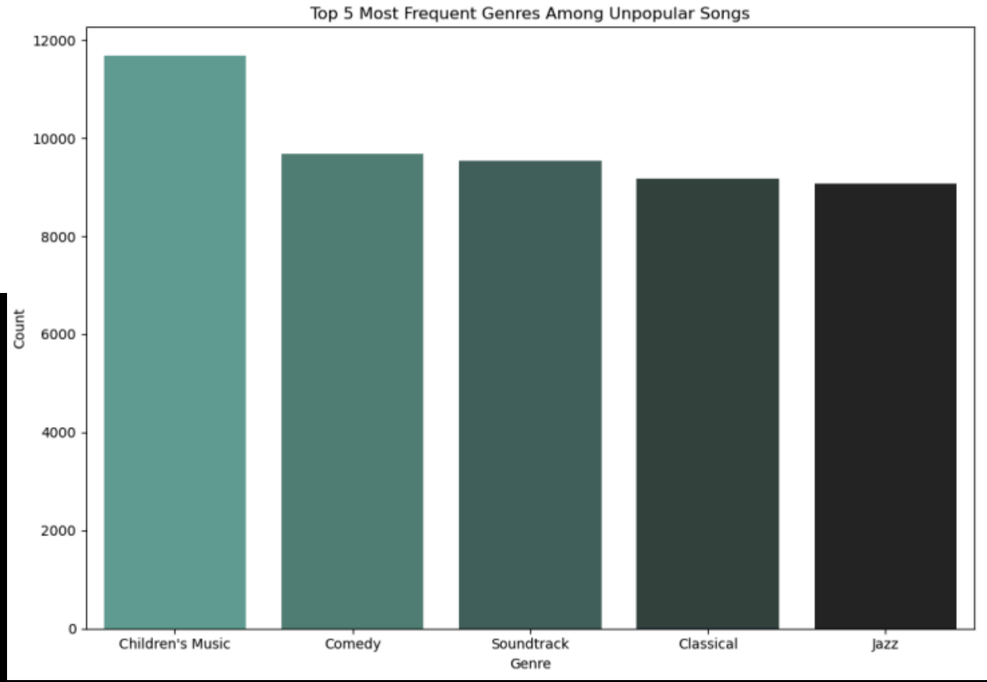


```
fig, ax = plt.subplots(figsize=(10, 7))
sns.barplot(x=popular_genre_df['genre'].head(5), y=popular_genre_df['count'].head(5),
            palette='dark:seagreen_r')

ax=plt.gca()
ax.set_xlabel('Genre')
ax.set_ylabel('Count')
ax.set_title('Top 5 Most Frequent Genres Among Popular Songs')
plt.tight_layout()
```




**The most common genres found in popular songs are Pop, followed by Rap, Rock, Hip-Hop, and Dance, as mentioned earlier.**

```
fig, ax = plt.subplots(figsize=(10,7))
sns.barplot(x=unpopular_genre_df['genre'].head(5), y=unpopular_genre_df['count'].head(5),
            palette='dark:#5A9_r')

ax=plt.gca()
ax.set_xlabel('Genre')
ax.set_ylabel('Count')
ax.set_title('Top 5 Most Frequent Genres Among Unpopular Songs')
plt.tight_layout();
```




**The less popular songs often fall into common genres that cater to specific audiences, such as "Children's Music," which are not frequently listened to. These patterns are understandable given the nature of these genres.**

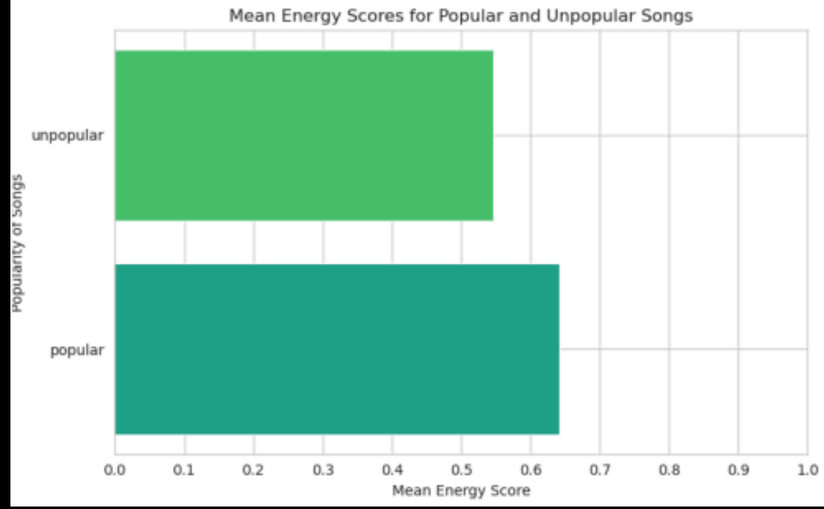
# Insights



```
#removing outliers from energy scores and separating them to Series for popular and unpopular songs
popular_energy_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['energy'])==False]
print(popular_energy_clean['energy'].describe())

unpopular_energy_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs_df['energy'])==False]
print(unpopular_energy_clean['energy'].describe())
```

```
count    20040.000000
mean         0.642509
std          0.195809
min          0.074000
25%          0.511000
50%          0.662000
75%          0.796000
max          0.999000
Name: energy, dtype: float64
count    156575.000000
mean          0.546617
std           0.282264
min           0.000020
25%           0.318000
50%           0.578000
75%           0.788000
max           0.999000
Name: energy, dtype: float64
```

```
import numpy as np

#storing mean energy scores in dict
mean_energy = {'popular': popular_energy_clean['energy'].mean(),
               'unpopular': unpopular_energy_clean['energy'].mean()}
#visualizing mean scores
with sns.axes_style("whitegrid"):
    fig, ax = plt.subplots(figsize=(8,5))
    ax.barh(y=list(mean_energy.keys()),
            width=list(mean_energy.values()),
            color=[sns.color_palette('viridis')[3],sns.color_palette('viridis')[4]])
    ax.set_xlim(0, 1)
    ax.set_xticks(np.arange(0,1.1,0.1))
    ax.set_ylabel('Popularity of Songs')
    ax.set_xlabel('Mean Energy Score')
    ax.set_title('Mean Energy Scores for Popular and Unpopular Songs')
    plt.tight_layout()
```




**Analyzing the 'Energy' feature in popular and unpopular songs, we removed outliers and calculated the mean energy scores for each group. The bar chart shows that popular songs tend to have a higher mean energy score compared to unpopular songs, indicating that higher energy levels may contribute to a song's popularity**

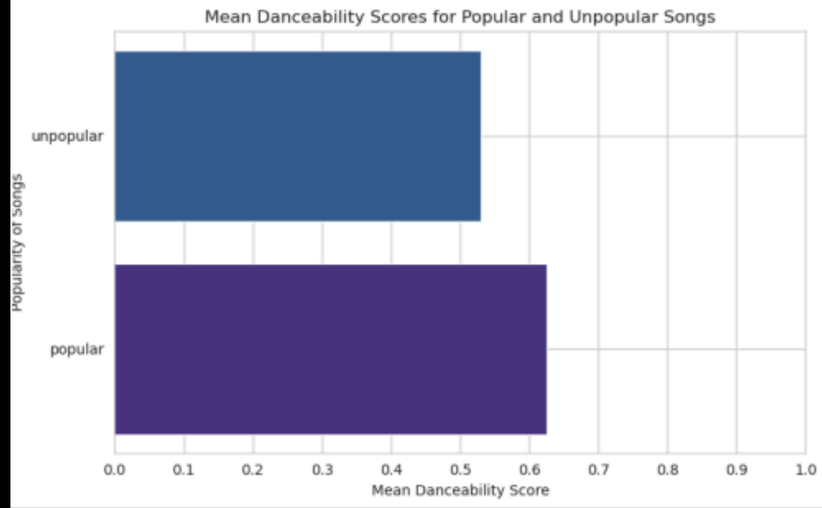
# Insights



```
#removing outliers from danceability scores and separating them to Series for popular and unpopular songs
popular_dance_clean = popular_songs_df[find_outliers_IQR(popular_songs_df['danceability'])==False]
print(popular_dance_clean['danceability'].describe())

unpopular_dance_clean = unpopular_songs_df[find_outliers_IQR(unpopular_songs_df['danceability'])==False]
print(unpopular_dance_clean['danceability'].describe())
```

```
count    20094.000000
mean         0.625974
std          0.151130
min          0.196000
25%          0.523000
50%          0.636000
75%          0.738000
max          0.985000
Name: danceability, dtype: float64
count    156575.000000
mean          0.530440
std           0.191956
min           0.056900
25%           0.401000
50%           0.547000
75%           0.674000
max           0.989000
Name: danceability, dtype: float64
```

```
#storing mean danceability scores in dict
mean_danceability = {'popular': popular_dance_clean['danceability'].mean(),
                     'unpopular': unpopular_dance_clean['danceability'].mean()}

#visualizing mean scores
with sns.axes_style("whitegrid"):
    fig, ax = plt.subplots(figsize=(8,5))
    ax.barh(y=list(mean_danceability.keys()),
            width=list(mean_danceability.values()),
            color=[sns.color_palette('viridis')[0],sns.color_palette('viridis')[1]])
    ax.set_xlim(0, 1)
    ax.set_xticks(np.arange(0,1.1,0.1))
    ax.set_ylabel('Popularity of Songs')
    ax.set_xlabel('Mean Danceability Score')
    ax.set_title('Mean Danceability Scores for Popular and Unpopular Songs')
    plt.tight_layout()
```

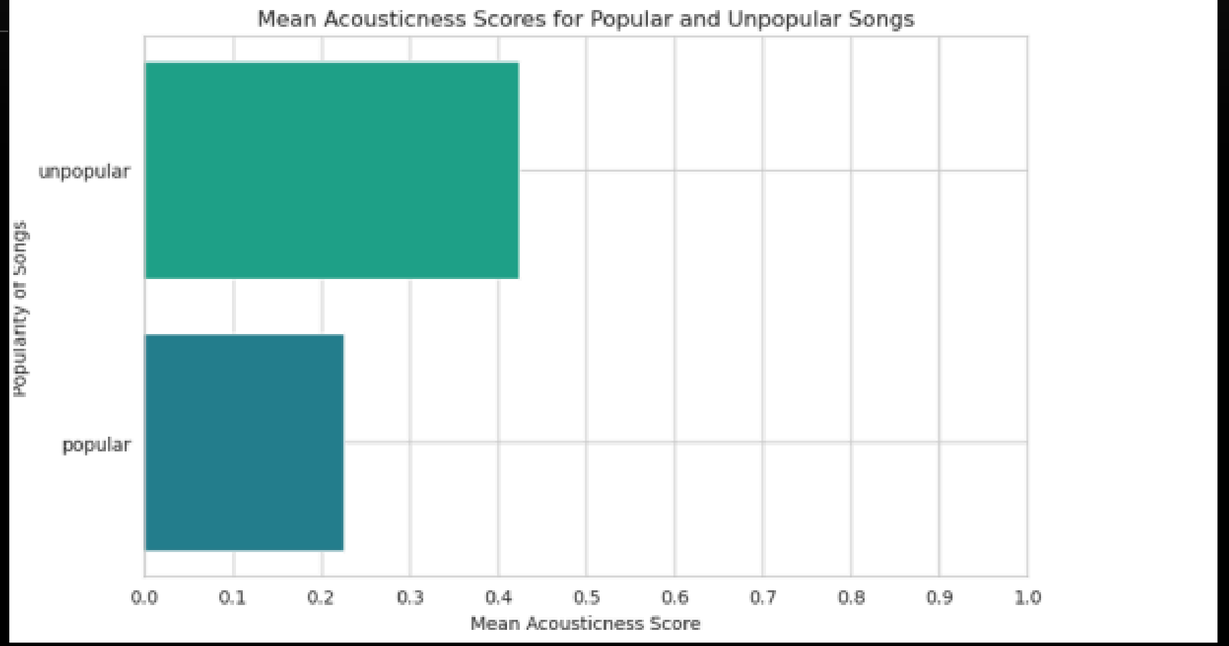



**Investigating the 'Danceability' feature in both popular and less popular songs involved eliminating outliers and calculating the average danceability scores for each category. The analysis indicates that, on average, popular songs tend to have slightly higher danceability scores compared to less popular songs. This implies that a song's danceability may influence its popularity.**

# Insights



```python
#storing mean acousticness scores in dict
mean_acousticness = {'popular': popular_acoustic_clean['acousticness'].mean(),
                     'unpopular': unpopular_acoustic_clean['acousticness'].mean()}

#visualizing mean scores
with sns.axes_style("whitegrid"):
    fig, ax = plt.subplots(figsize=(8,5))
    ax.barh(y=list(mean_acousticness.keys()),
            width=list(mean_acousticness.values()),
            color=[sns.color_palette('viridis')[2],sns.color_palette('viridis')[3]])
    ax.set_xlim(0, 1)
    ax.set_xticks(np.arange(0,1.1,0.1))
    ax.set_ylabel('Popularity of Songs')
    ax.set_xlabel('Mean Acousticness Score')
    ax.set_title('Mean Acousticness Scores for Popular and Unpopular Songs')
    plt.tight_layout()
```




**Similar to the energy and danceability scores, popular songs often have lower acousticness scores. This trend aligns with our observation that acoustic songs typically have lower energy levels and are seldom danceable.**

# Takeaways

In the competitive music streaming market, retaining current subscribers and attracting new ones is crucial for success. Platforms like Spotify use predictive analytics to anticipate the next popular songs, enabling them to curate improved playlists and secure exclusive deals with both established and emerging artists more effectively. Our analysis of around 176,000 songs revealed the following trends:

- Popular songs typically fall under genres like Pop, Rap, Rock, Hip-Hop, and Dance.
- Less mainstream genres such as Children's Music, Comedy, Soundtracks, Classical, and Jazz tend to be less popular.
- Generally, popular songs are high-energy, danceable, and less acoustic compared to less popular tracks.

# Recommendations using our model

- Secure exclusive deals with upcoming artists
- Curate playlists with potential hits
- Provide personalized song recommendations
- Use targeted ads
- Collaborate with record labels
- Analyze music trends
- Find commonality between less popular genres and features of popular genres and give a boost to those songs.

# Thank You!!

***A project by : Nimai (D001), Senorita (D004), Raizel (D007), Jai (D018), Aamir(D020)***