Spotify Song Popularity Prediction Report

Problem Statement

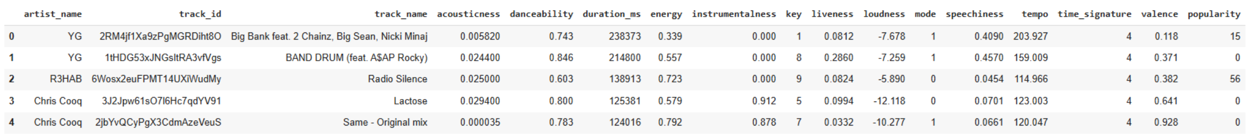
Why do some songs go viral while most remain unnoticed? With millions of tracks flooding streaming platforms, predicting a song’ s success remains a challenge. This project explores the relationship between song attributes and popularity, leveraging machine learning to identify key factors and build predictive models.

Dataset Overview

The dataset consists of **130,664 rows and 17 columns**, containing information about various song attributes. The key columns include:

- **Track Name, Track ID, Artist Name**
- **Audio Features**: Acousticness, Liveness, Loudness, Energy, Mode, Danceability, Tempo, Speechiness, Instrumentalness, Valence, etc.
- **Target Variable**: **Popularity Score (0-100)**

DF



	artist_name	track_id	track_name	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence	popularity
0	YG	2RM4jf1Xa9zPgMGRDiht8O	Big Bank feat. 2 Chainz, Big Sean, Nicki Minaj	0.005820	0.743	238373	0.339	0.000	1	0.0812	-7.678	1	0.4090	203.927	4	0.118	15
1	YG	1tHDG53xJNGsItRA3vfVgs	BAND DRUM (feat. A$AP Rocky)	0.024400	0.846	214800	0.557	0.000	8	0.2860	-7.259	1	0.4570	159.009	4	0.371	0
2	R3HAB	6Wosx2euFPMT14UXiWudMy	Radio Silence	0.025000	0.603	138913	0.723	0.000	9	0.0824	-5.890	0	0.0454	114.966	4	0.382	56
3	Chris Cooq	3J2Jpw61sO7I6Hc7qdYV91	Lactose	0.029400	0.800	125381	0.579	0.912	5	0.0994	-12.118	0	0.0701	123.003	4	0.641	0
4	Chris Cooq	2jbYvQCyPgX3CdmAzeVeuS	Same - Original mix	0.000035	0.783	124016	0.792	0.878	7	0.0332	-10.277	1	0.0661	120.047	4	0.928	0

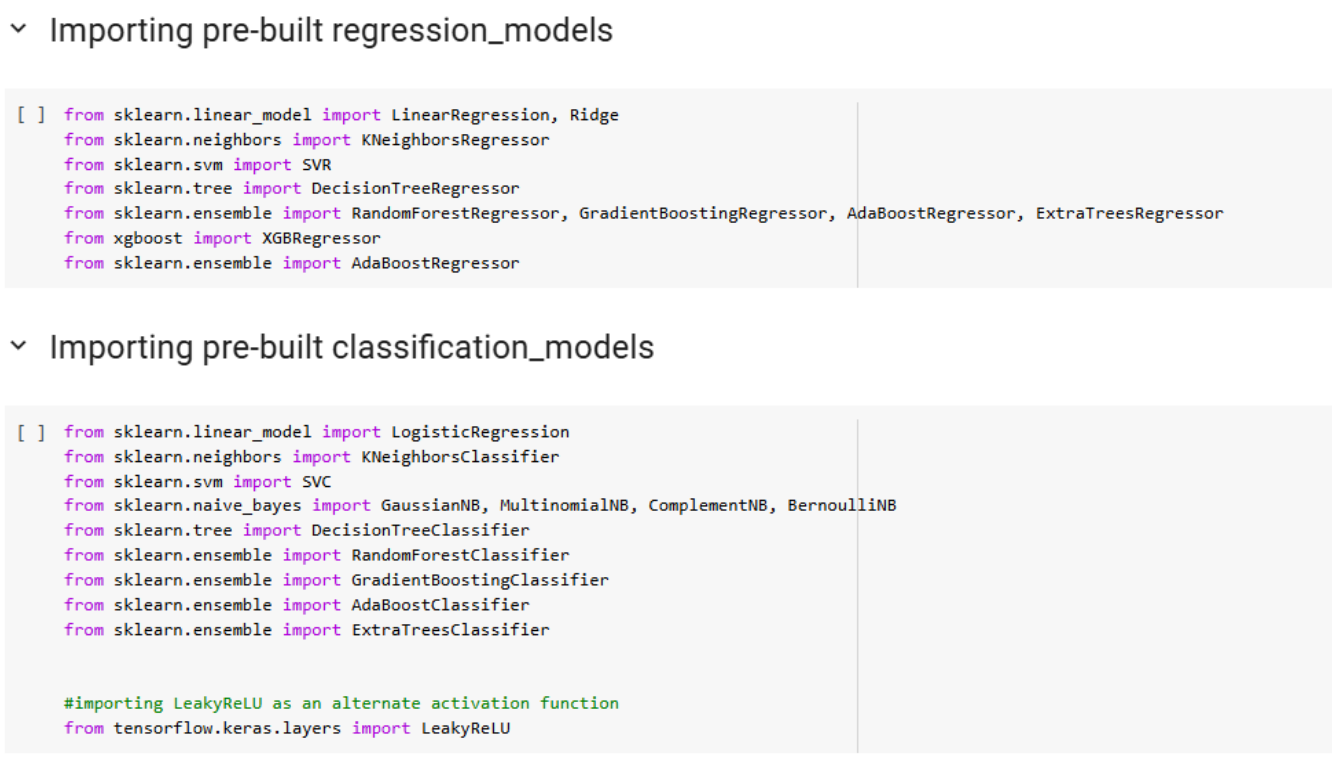
Downloading Python Packages and importing libraries

```
!pip install xgboost
!pip install scikit-learn
! pip install pandas pandas-profiling
! pip install xgboost scikit-learn
```

```
]  !pip install ydata-profiling
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import warnings
warnings.filterwarnings('ignore')
import plotly.express as px
import plotly.graph_objects as go
from ydata_profiling import ProfileReport
```

DF



Importing pre-built regression_models

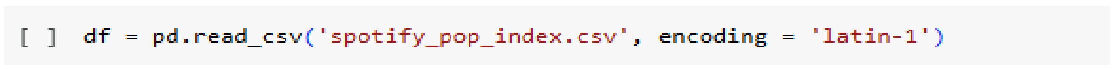
```
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, AdaBoostRegressor, ExtraTreesRegressor
from xgboost import XGBRegressor
from sklearn.ensemble import AdaBoostRegressor
```

Importing pre-built classification_models

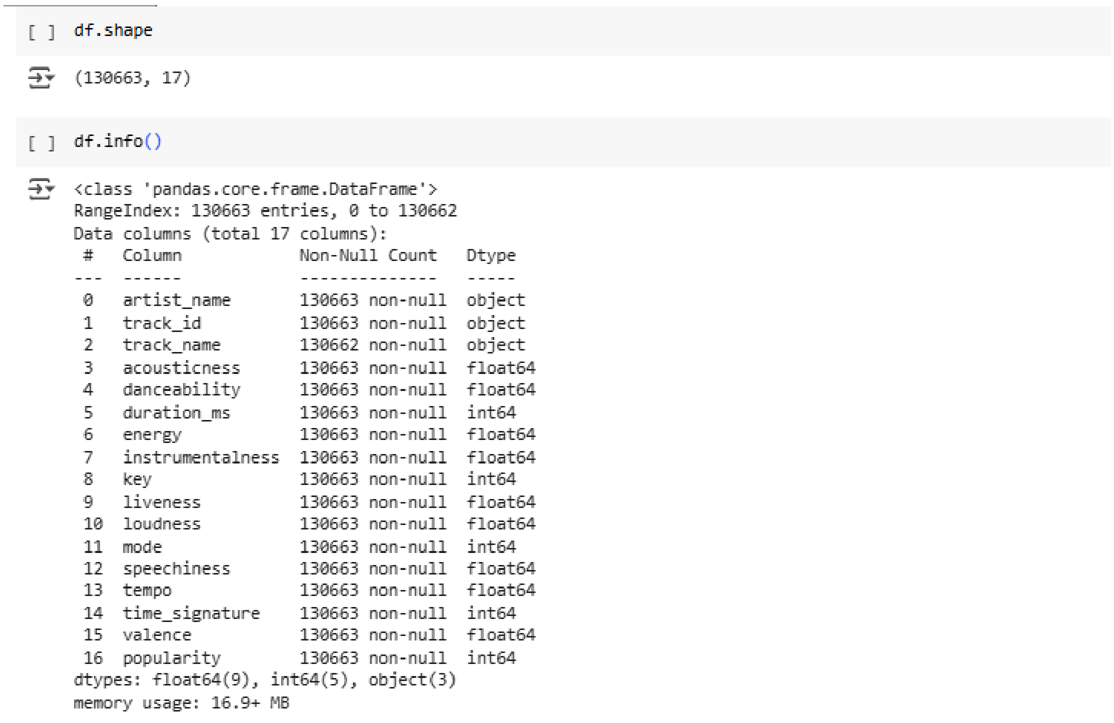
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB, MultinomialNB, ComplementNB, BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import ExtraTreesClassifier


#importing LeakyReLU as an alternate activation function
from tensorflow.keras.layers import LeakyReLU
```

Importing the Dataset and Exploratory Data Analysis (EDA)

DF



```
[ ]  df = pd.read_csv('spotify_pop_index.csv', encoding = 'latin-1')
```

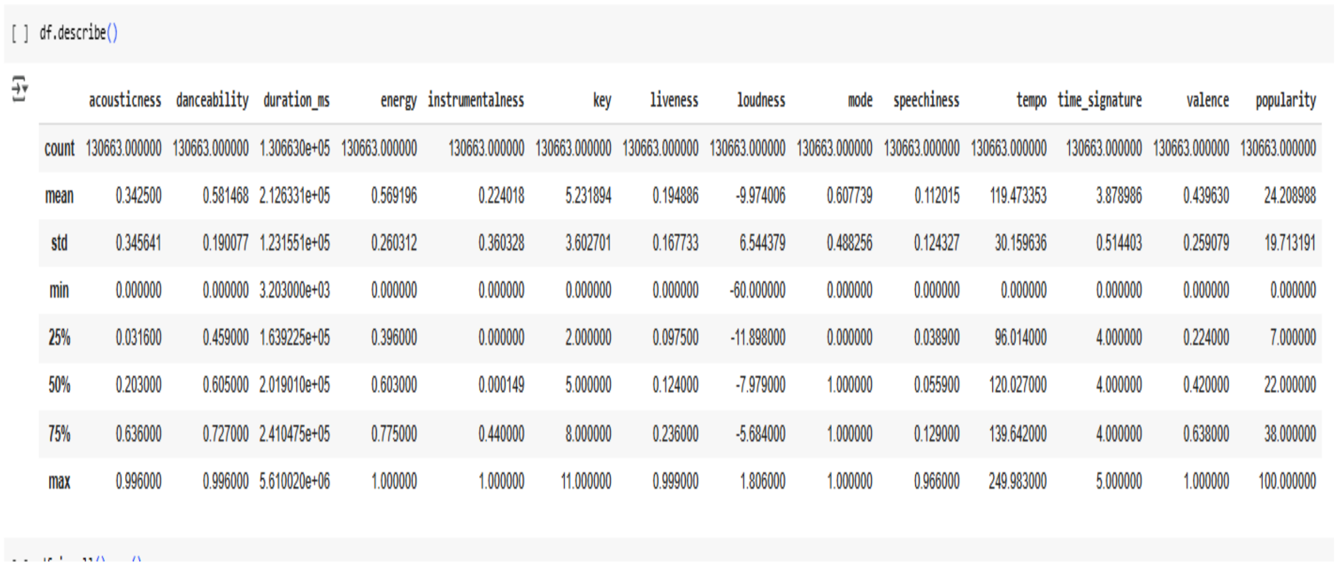


```
[ ]  df.shape
```

```
(130663, 17)
```

```
[ ]  df.info()
```

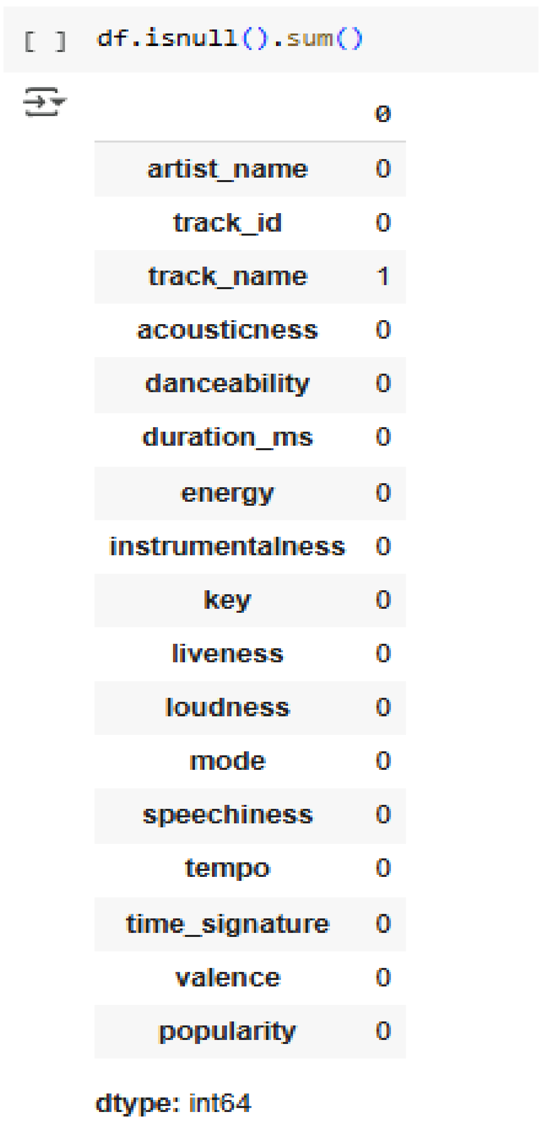
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130663 entries, 0 to 130662
Data columns (total 17 columns):
 #   Column            Non-Null Count   Dtype  
---  ------            --------------   -----  
 0   artist_name       130663 non-null  object 
 1   track_id          130663 non-null  object 
 2   track_name        130662 non-null  object 
 3   acousticness      130663 non-null  float64
 4   danceability      130663 non-null  float64
 5   duration_ms       130663 non-null  int64  
 6   energy            130663 non-null  float64
 7   instrumentalness  130663 non-null  float64
 8   key               130663 non-null  int64  
 9   liveness          130663 non-null  float64
 10  loudness          130663 non-null  float64
 11  mode              130663 non-null  int64  
 12  speechiness       130663 non-null  float64
 13  tempo             130663 non-null  float64
 14  time_signature    130663 non-null  int64  
 15  valence           130663 non-null  float64
 16  popularity        130663 non-null  int64  
dtypes: float64(9), int64(5), object(3)
memory usage: 16.9+ MB
```



```
[ ]  df.describe()
```

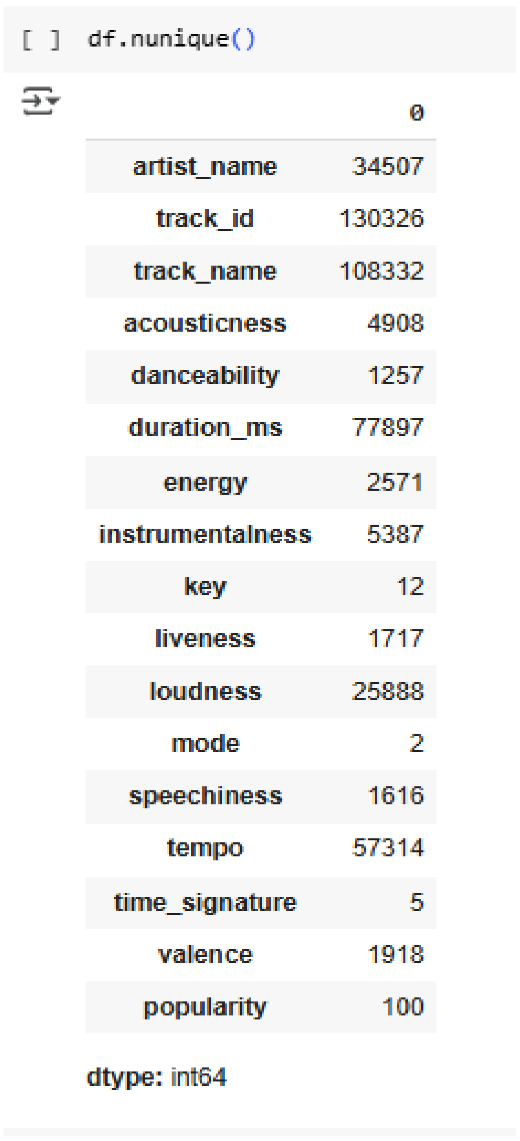
	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence	popularity
count	130663.000000	130663.000000	1.306630e+05	130663.000000	130663.000000	130663.000000	130663.000000	130663.000000	130663.000000	130663.000000	130663.000000	130663.000000	130663.000000	130663.000000
mean	0.342500	0.581468	2.126331e+05	0.569196	0.224018	5.231894	0.194886	-9.974006	0.607739	0.112015	119.473353	3.878986	0.439630	24.208988
std	0.345641	0.190077	1.231551e+05	0.260312	0.360328	3.602701	0.167733	6.544379	0.488256	0.124327	30.159636	0.514403	0.259079	19.713191
min	0.000000	0.000000	3.203000e+03	0.000000	0.000000	0.000000	0.000000	-60.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.031600	0.459000	1.639225e+05	0.396000	0.000000	2.000000	0.097500	-11.898000	0.000000	0.038900	96.014000	4.000000	0.224000	7.000000
50%	0.203000	0.605000	2.019010e+05	0.603000	0.000149	5.000000	0.124000	-7.979000	1.000000	0.055900	120.027000	4.000000	0.420000	22.000000
75%	0.636000	0.727000	2.410475e+05	0.775000	0.440000	8.000000	0.236000	-5.684000	1.000000	0.129000	139.642000	4.000000	0.638000	38.000000
max	0.996000	0.996000	5.610020e+06	1.000000	1.000000	11.000000	0.999000	1.806000	1.000000	0.966000	249.983000	5.000000	1.000000	100.000000

DF



```
[ ] df.isnull().sum()
```

```
                       0
     artist_name       0
        track_id       0
      track_name       1
    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
      popularity       0

dtype: int64
```

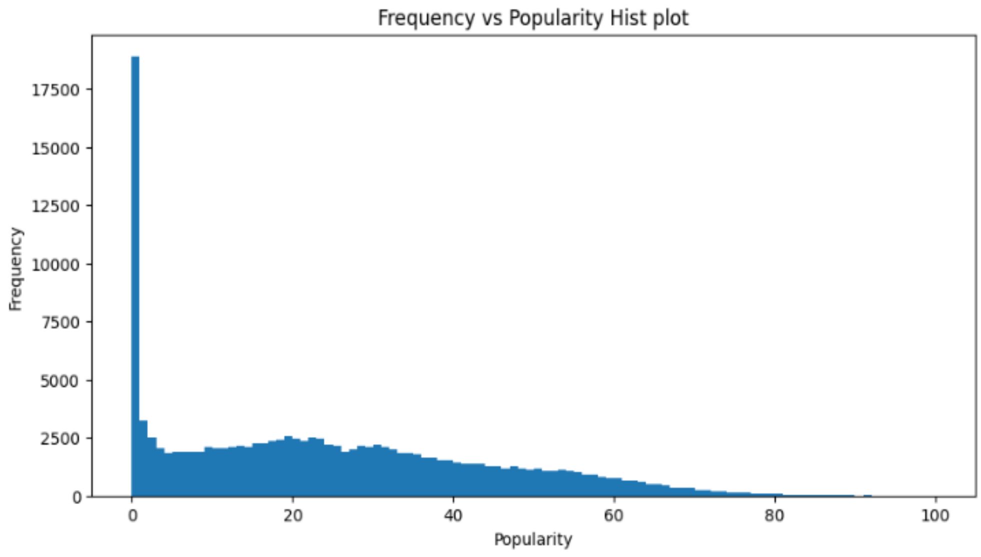


```
[ ] df.nunique()
```

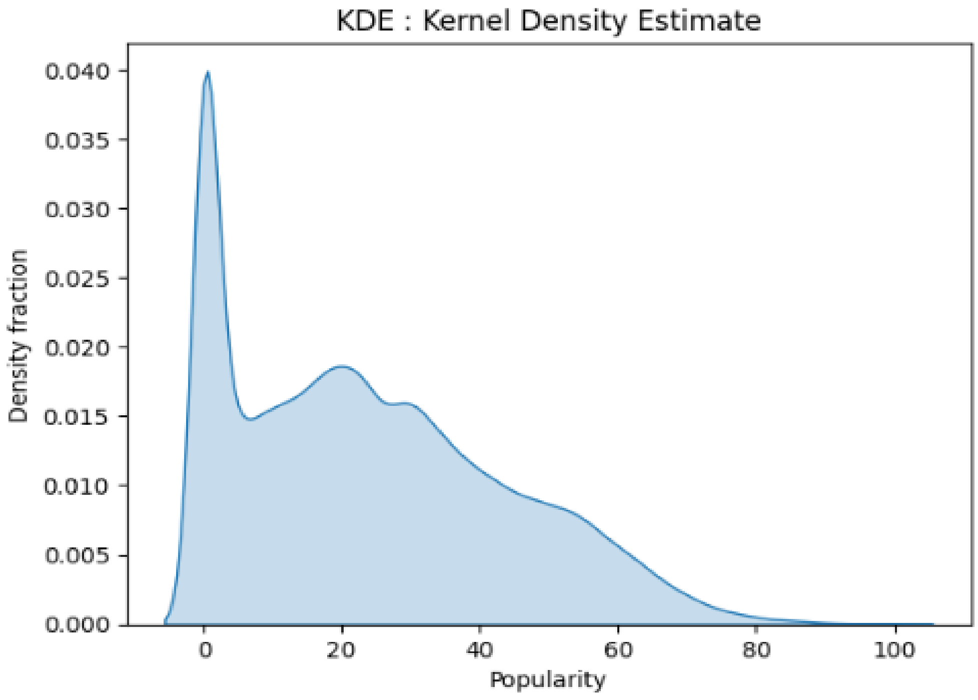
```
                       0
     artist_name   34507
        track_id  130326
      track_name  108332
    acousticness    4908
    danceability    1257
     duration_ms   77897
          energy    2571
instrumentalness    5387
             key      12
        liveness    1717
        loudness   25888
            mode       2
     speechiness    1616
           tempo   57314
  time_signature       5
         valence    1918
      popularity     100

dtype: int64
```

DF

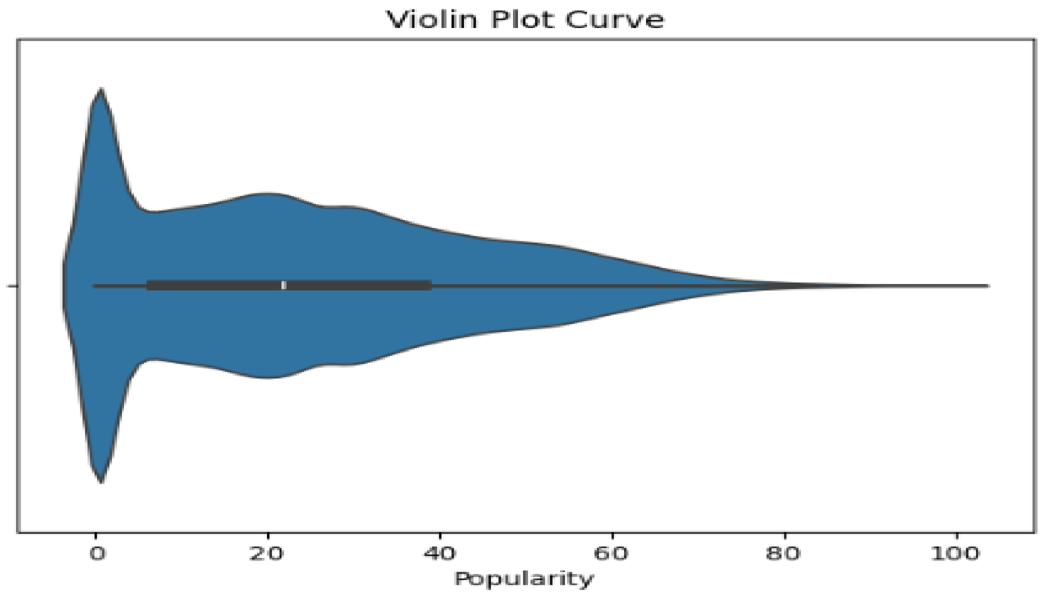
```
Text(0, 0.5, 'Frequency')
```



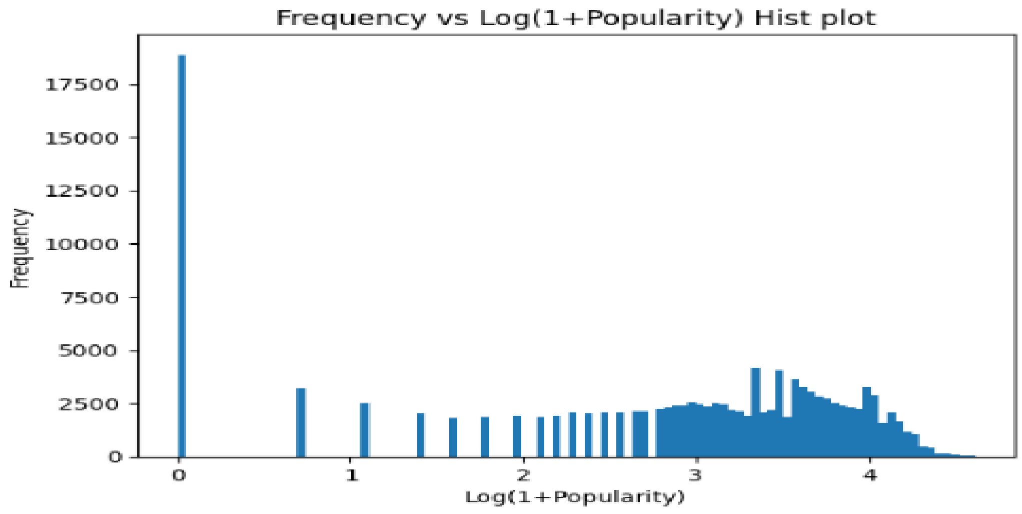
```
Text(0, 0.5, 'Density fraction')
```



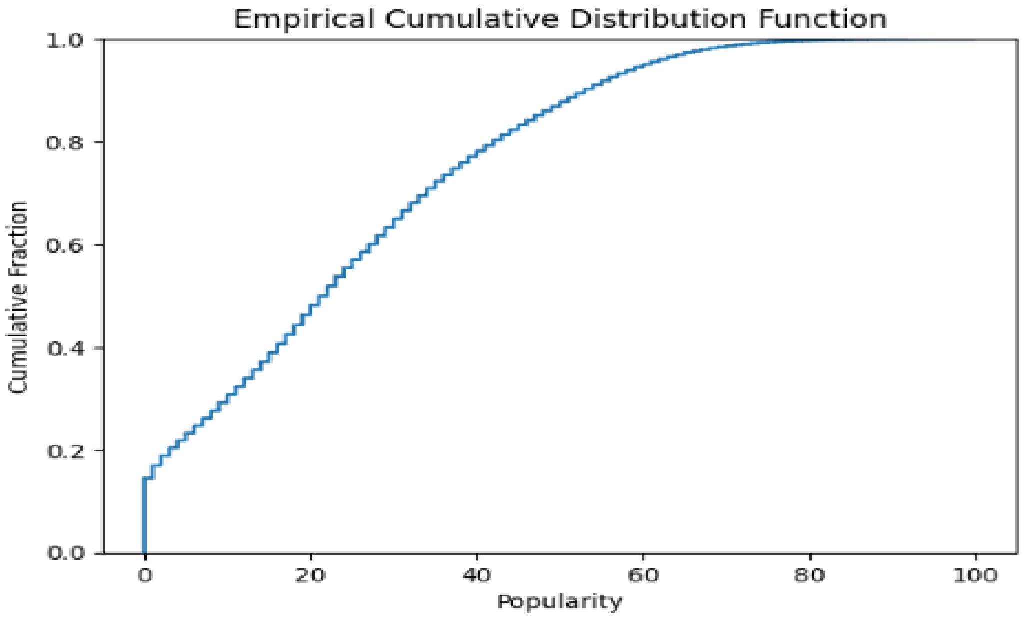
DF



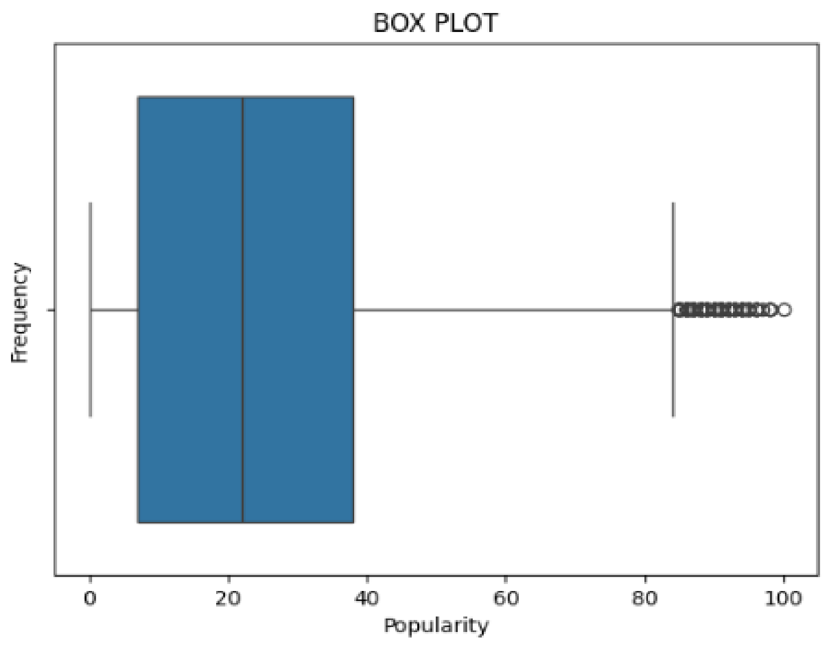
Violin Plot Curve
0
20
40
60
80
100
Popularity

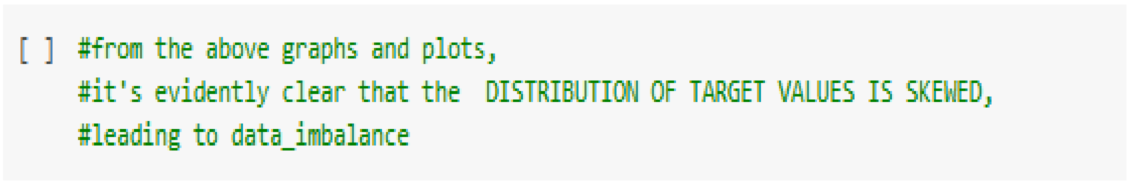


Frequency vs Log(1+Popularity) Hist plot
Frequency
17500
15000
12500
10000
7500
5000
2500
0
0
1
2
3
4
Log(1+Popularity)



Empirical Cumulative Distribution Function
Cumulative Fraction
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Popularity





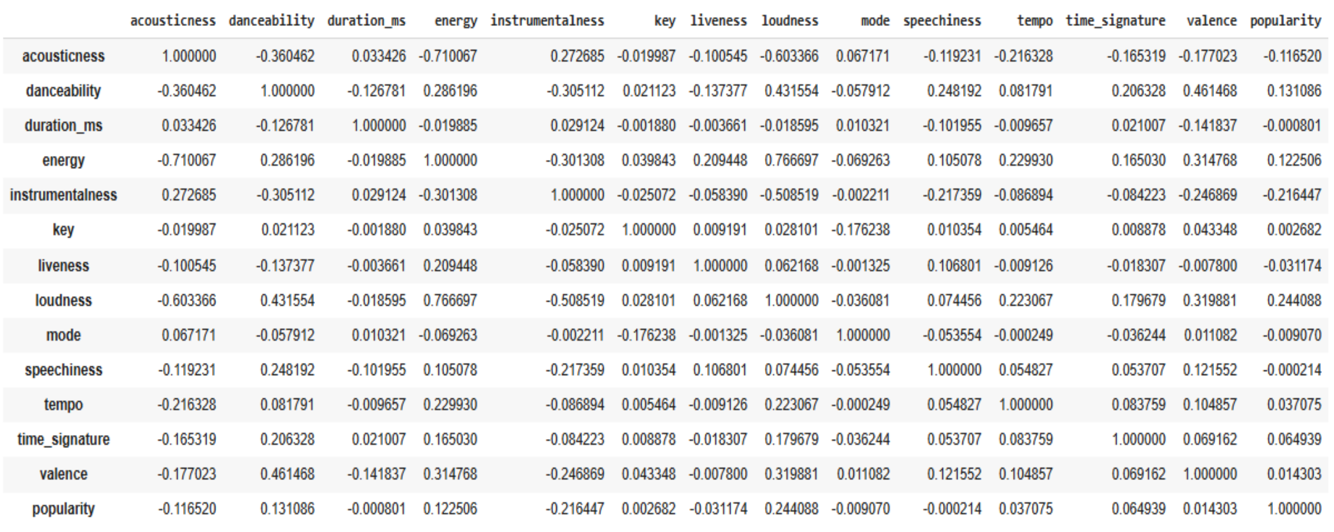
```
[ ]  #from the above graphs and plots,
     #it's evidently clear that the  DISTRIBUTION OF TARGET VALUES IS SKEWED,
     #leading to data_imbalance
```

- The target_col = Popularity_index is heavily imbalanced towards the lower values, with its peak at pop_index = 0.

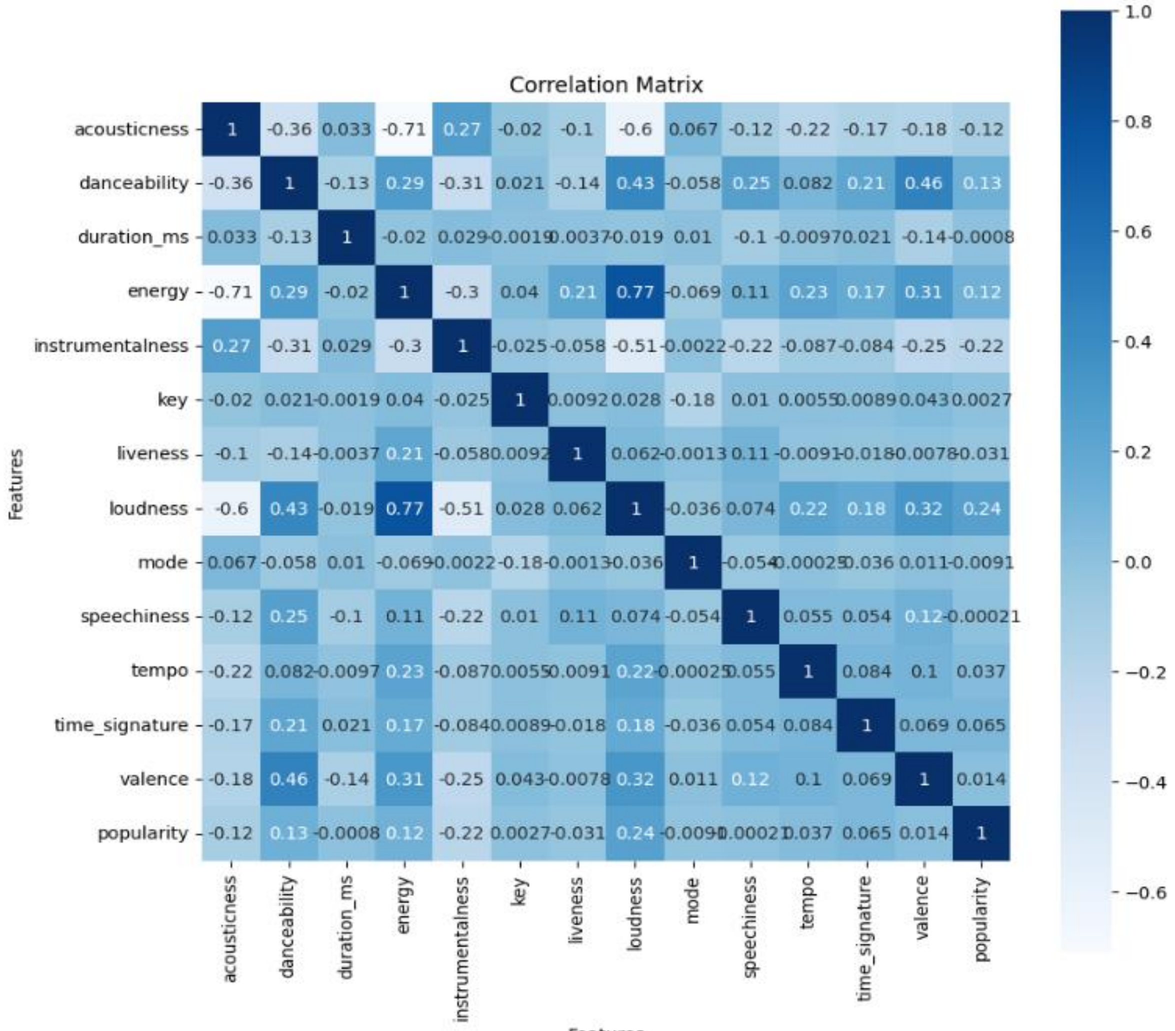
- As we go up the popularity_index, the frequency of the observations decrease/ fall rapidly. This may create an inherent bias for the regressor / classifier model while training as it MAY IGNORE THE MINORITY CLASS correlations.
- This may lead to predictions being made excessvily in favour of the majority class (towards 0)

DF

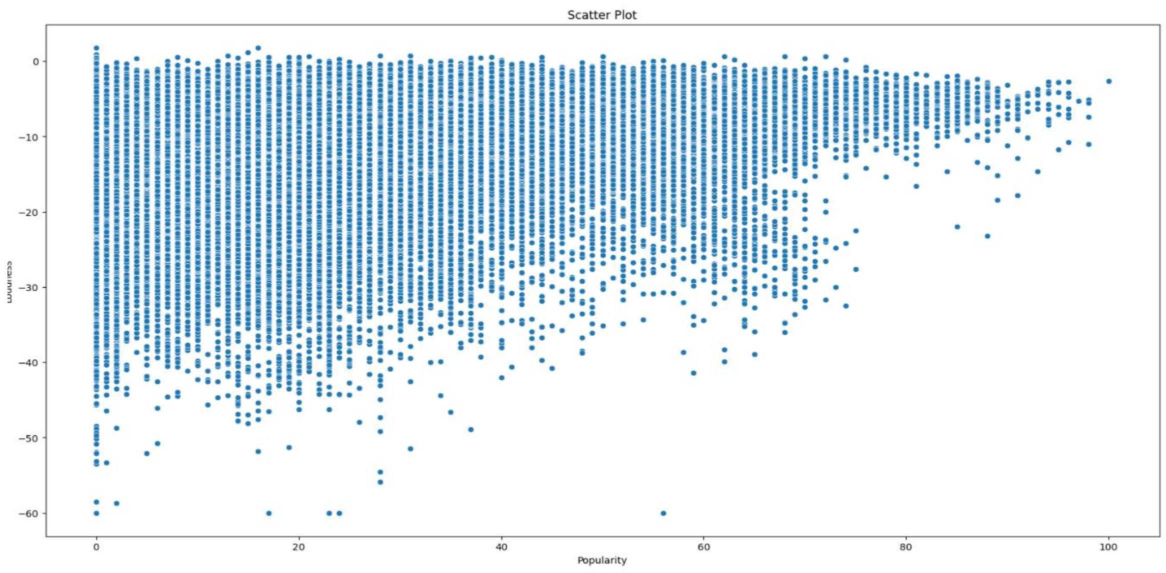
Correlation_Matrix & HeatMap

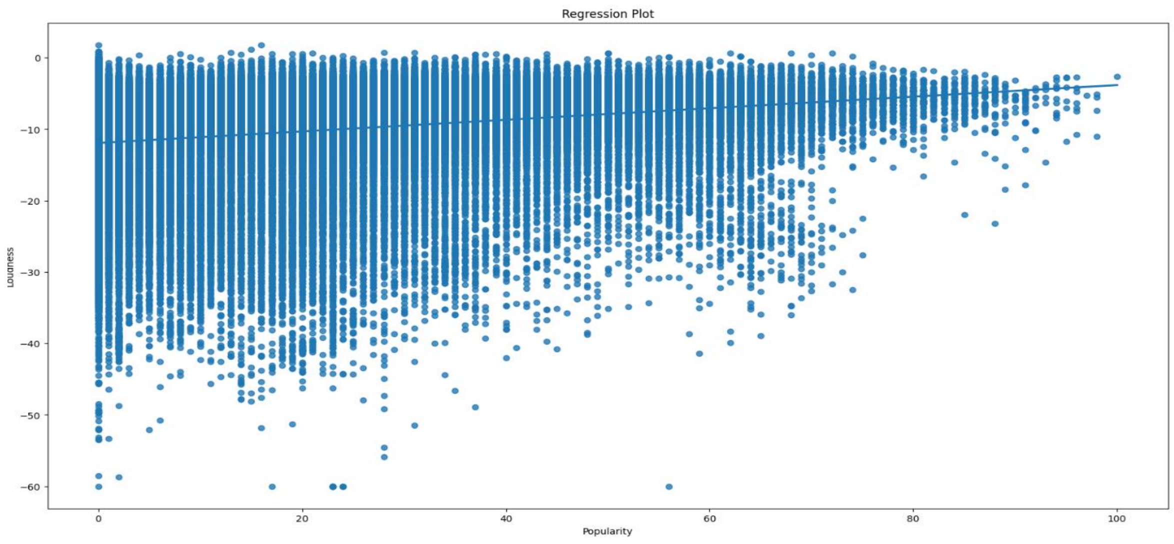


	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence	popularity
acousticness	1.000000	-0.360462	0.033426	-0.710067	0.272685	-0.019987	-0.100545	-0.603366	0.067171	-0.119231	-0.216328	-0.165319	-0.177023	-0.116520
danceability	-0.360462	1.000000	-0.126781	0.286196	-0.305112	0.021123	-0.137377	0.431554	-0.057912	0.248192	0.081791	0.206328	0.461468	0.131086
duration_ms	0.033426	-0.126781	1.000000	-0.019885	0.029124	-0.001880	-0.003661	-0.018595	0.010321	-0.101955	-0.009657	0.021007	-0.141837	-0.000801
energy	-0.710067	0.286196	-0.019885	1.000000	-0.301308	0.039843	0.209448	0.766697	-0.069263	0.105078	0.229930	0.165030	0.314768	0.122506
instrumentalness	0.272685	-0.305112	0.029124	-0.301308	1.000000	-0.025072	-0.058390	-0.508519	-0.002211	-0.217359	-0.086894	-0.084223	-0.246869	-0.216447
key	-0.019987	0.021123	-0.001880	0.039843	-0.025072	1.000000	0.009191	0.028101	-0.176238	0.010354	0.005464	0.008878	0.043348	0.002682
liveness	-0.100545	-0.137377	-0.003661	0.209448	-0.058390	0.009191	1.000000	0.062168	-0.001325	0.106801	-0.009126	-0.018307	-0.007800	-0.031174
loudness	-0.603366	0.431554	-0.018595	0.766697	-0.508519	0.028101	0.062168	1.000000	-0.036081	0.074456	0.223067	0.179679	0.319881	0.244088
mode	0.067171	-0.057912	0.010321	-0.069263	-0.002211	-0.176238	-0.001325	-0.036081	1.000000	-0.053554	-0.000249	-0.036244	0.011082	-0.009070
speechiness	-0.119231	0.248192	-0.101955	0.105078	-0.217359	0.010354	0.106801	0.074456	-0.053554	1.000000	0.054827	0.053707	0.121552	-0.000214
tempo	-0.216328	0.081791	-0.009657	0.229930	-0.086894	0.005464	-0.009126	0.223067	-0.000249	0.054827	1.000000	0.083759	0.104857	0.037075
time_signature	-0.165319	0.206328	0.021007	0.165030	-0.084223	0.008878	-0.018307	0.179679	-0.036244	0.053707	0.083759	1.000000	0.069162	0.064939
valence	-0.177023	0.461468	-0.141837	0.314768	-0.246869	0.043348	-0.007800	0.319881	0.011082	0.121552	0.104857	0.069162	1.000000	0.014303
popularity	-0.116520	0.131086	-0.000801	0.122506	-0.216447	0.002682	-0.031174	0.244088	-0.009070	-0.000214	0.037075	0.064939	0.014303	1.000000



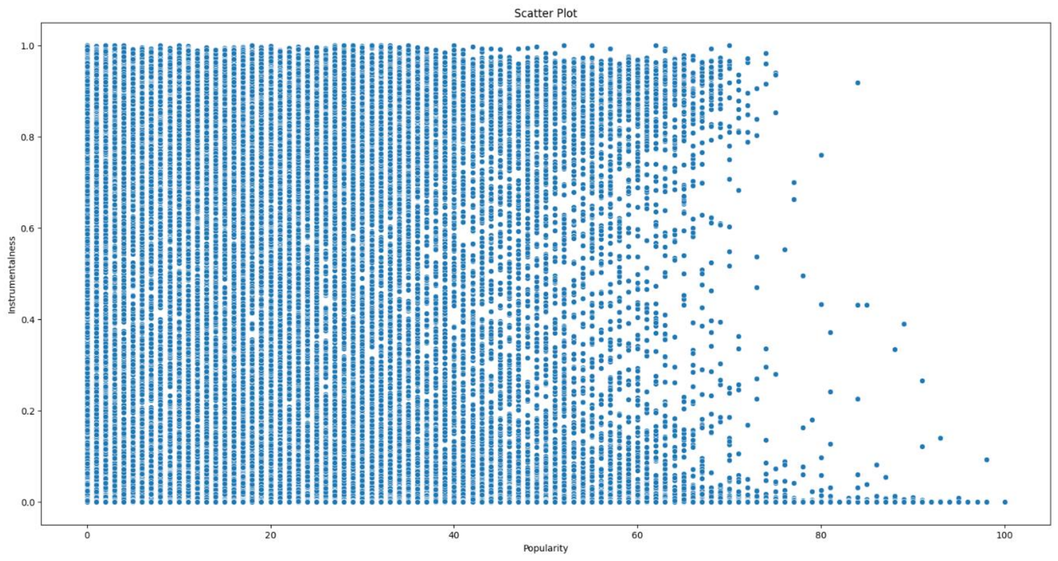
Analysis of Density Variation and Correlation of Loudness with Popularity

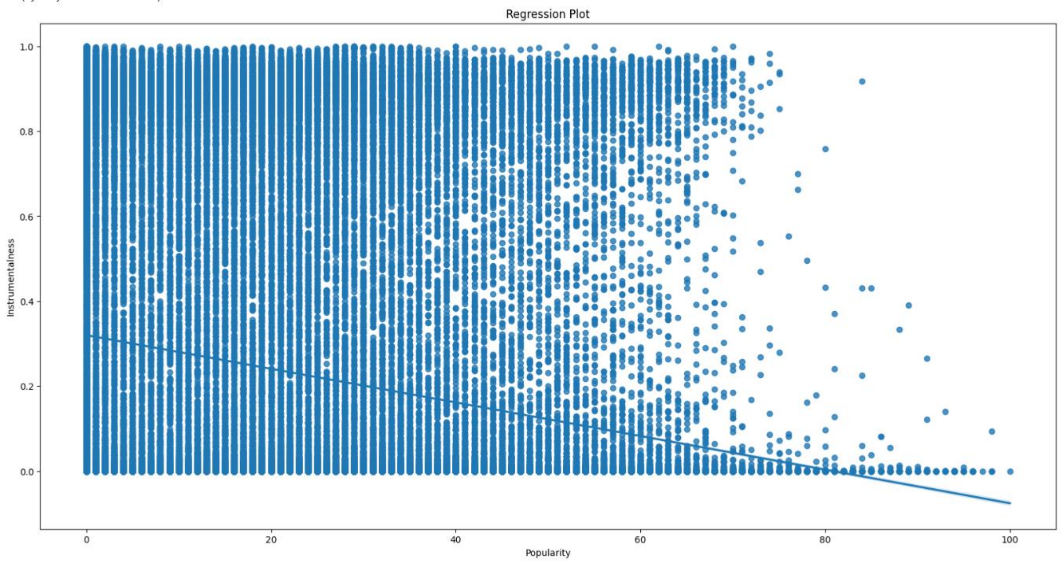




- **Loudness shows the highest correlation with Popularity**, justifying this focused analysis.
- **Severe imbalance in Popularity**: Most songs lie in the 0-50 range, making trend identification challenging.
- **Popular songs (60+) tend to have higher Loudness**, but variance remains high at lower Popularity levels.
- **Regression plot suggests a weak linear relationship**, indicating possible non-linear dependencies.
- Further feature interactions might refine insights into song popularity trends.

Discussing its impact on the target variable (Popularity Score)

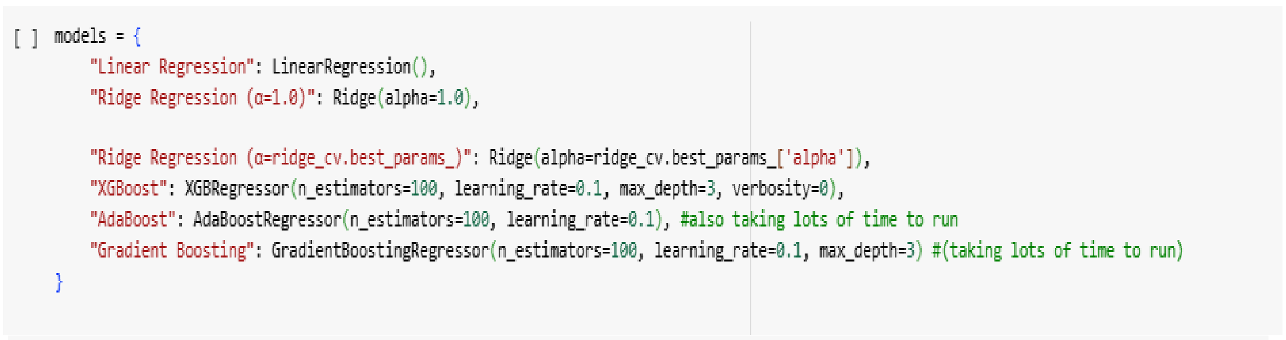




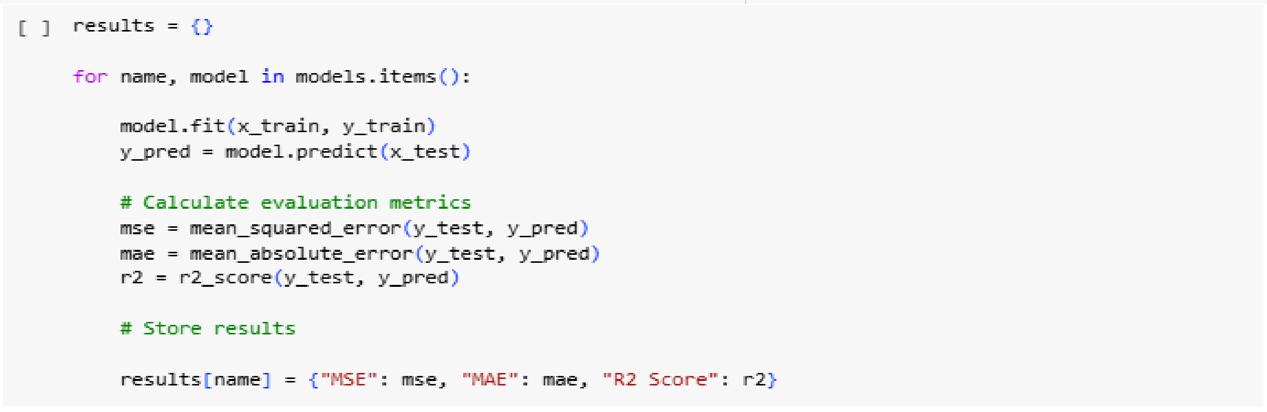
- **Instrumentalness has the second-highest correlation with Popularity**, making it a key feature for analysis.
- **Severe Popularity imbalance persists**, with most songs clustered in the 0-50 range, making patterns harder to interpret.
- **Higher Instrumentalness values (closer to 1) are more frequent in less popular songs**, suggesting that instrumental tracks are generally less popular.
- **Negative trend observed in the regression plot**, reinforcing that as Popularity increases, Instrumentalness decreases.
- Despite correlation, **variance remains high at lower Popularity levels**, indicating other influential factors at play.
- Further analysis with interactions or non-linear methods may provide deeper insights.

DF

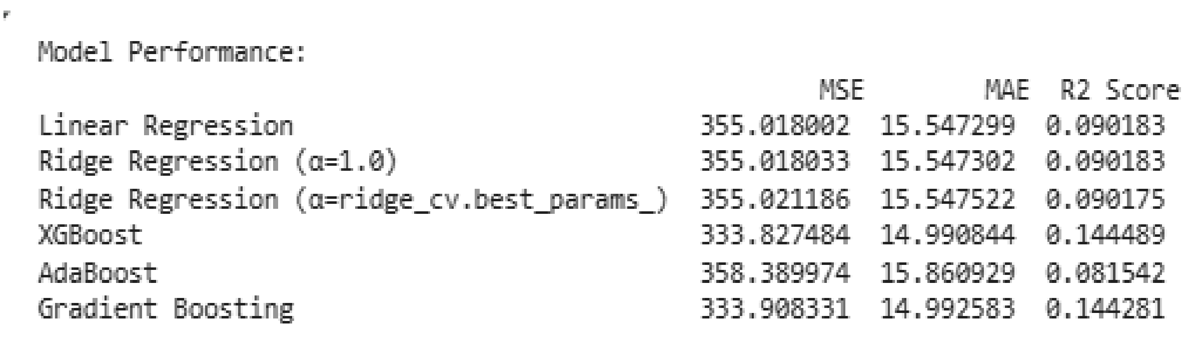
Regression using Classical Models



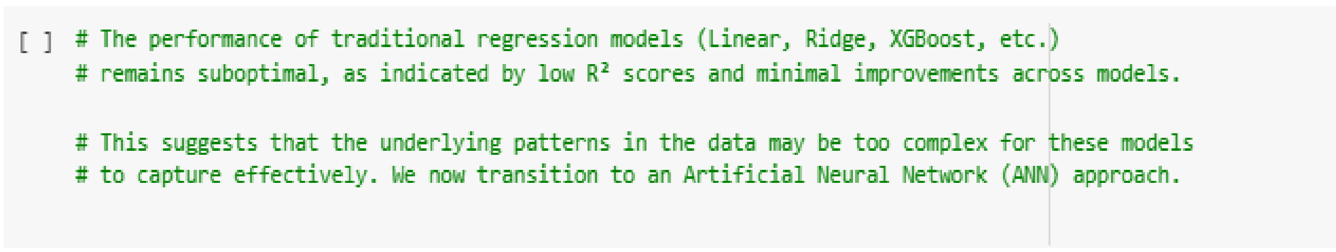
```
[ ] models = {
        "Linear Regression": LinearRegression(),
        "Ridge Regression (α=1.0)": Ridge(alpha=1.0),

        "Ridge Regression (α=ridge_cv.best_params_)": Ridge(alpha=ridge_cv.best_params_['alpha']),
        "XGBoost": XGBRegressor(n_estimators=100, learning_rate=0.1, max_depth=3, verbosity=0),
        "AdaBoost": AdaBoostRegressor(n_estimators=100, learning_rate=0.1), #also taking lots of time to run
        "Gradient Boosting": GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=3) #(taking lots of time to run)
    }
```



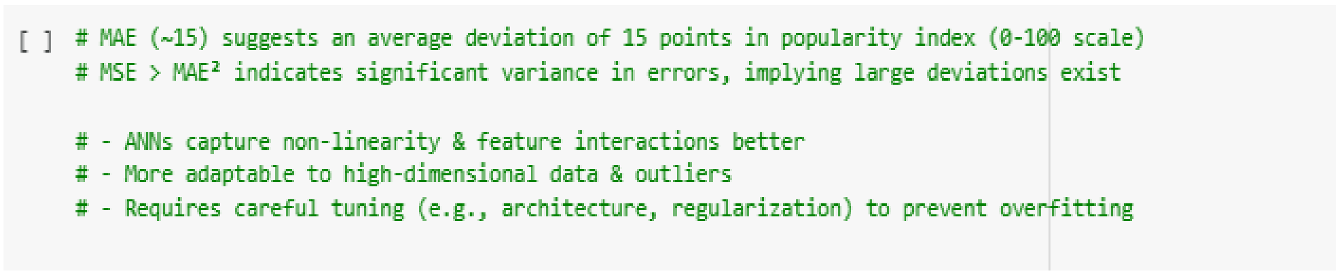
```
[ ] results = {}

    for name, model in models.items():

        model.fit(x_train, y_train)
        y_pred = model.predict(x_test)

        # Calculate evaluation metrics
        mse = mean_squared_error(y_test, y_pred)
        mae = mean_absolute_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)

        # Store results

        results[name] = {"MSE": mse, "MAE": mae, "R2 Score": r2}
```



```
Model Performance:
                                                   MSE        MAE  R2 Score
Linear Regression                           355.018002  15.547299  0.090183
Ridge Regression (α=1.0)                    355.018033  15.547302  0.090183
Ridge Regression (α=ridge_cv.best_params_)  355.021186  15.547522  0.090175
XGBoost                                     333.827484  14.990844  0.144489
AdaBoost                                    358.389974  15.860929  0.081542
Gradient Boosting                           333.908331  14.992583  0.144281
```



```
[ ] # The performance of traditional regression models (Linear, Ridge, XGBoost, etc.)
    # remains suboptimal, as indicated by low R² scores and minimal improvements across models.

    # This suggests that the underlying patterns in the data may be too complex for these models
    # to capture effectively. We now transition to an Artificial Neural Network (ANN) approach.
```



```
[ ] # MAE (~15) suggests an average deviation of 15 points in popularity index (0-100 scale)
    # MSE > MAE² indicates significant variance in errors, implying large deviations exist

    # - ANNs capture non-linearity & feature interactions better
    # - More adaptable to high-dimensional data & outliers
    # - Requires careful tuning (e.g., architecture, regularization) to prevent overfitting
```

MODEL COMPLEXITIES

Excluding Certain Models Due to High Computational Complexity

Certain models were excluded due to their **high computational cost**, making them impractical for large datasets.

1. k-Nearest Neighbors (KNN) Regression

- **Complexity:** O(n2)O(n^2)
- **Reason:** Requires computing pairwise distances for all points, making it **slow** and **inefficient** as data size grows.

2. Support Vector Regression (SVR)

- **RBF Kernel:** O(n3)O(n^3) → **High training time** due to kernel computation and quadratic optimization.
- **Linear Kernel:** O(n2)O(n^2) → **Still costly**, solving a large optimization problem.

3. Decision Tree Regression (DTR)

- **Complexity:** O(nlog⁡n)O(n \log n)
- **Reason:** Recursively splits data, which can be **computationally heavy** for large datasets.

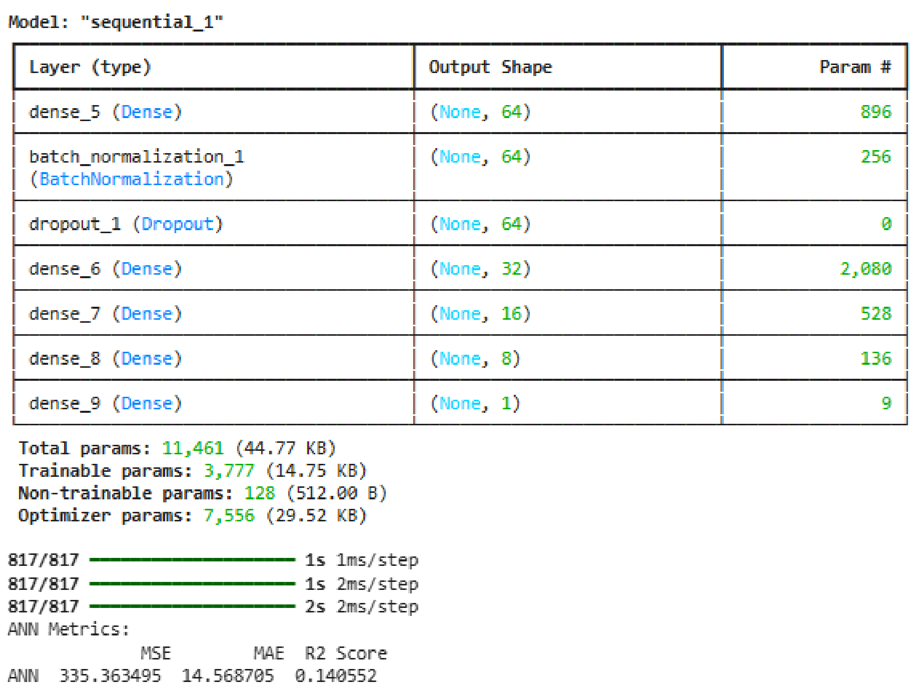
4. Random Forest Regression (RFR)

- **Complexity:** O(t⋅f⋅nlog⁡n)O(t \cdot f \cdot n \log n) (where tt = trees, ff = features)
- **Reason:** Training multiple trees increases computational cost, making it **resource-intensive**.

These models were avoided to prioritize **scalability and efficiency** for the dataset.

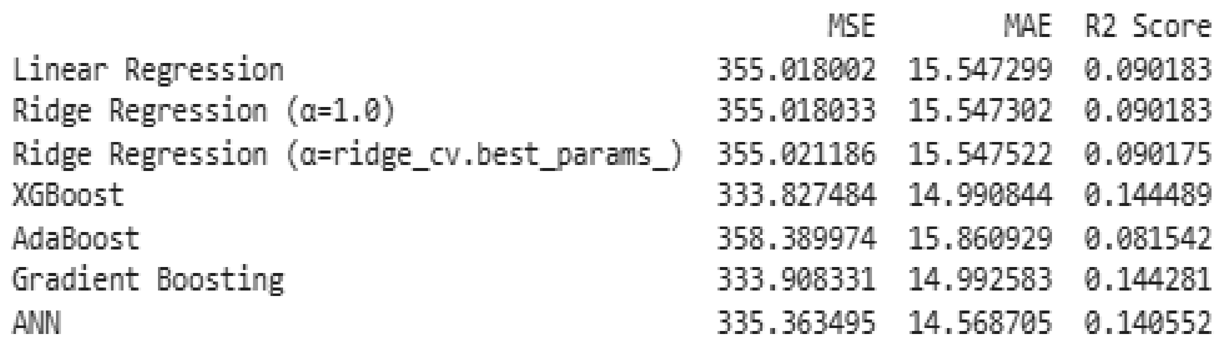
Regression using Artificial Neural Networks (ANNs)

DF

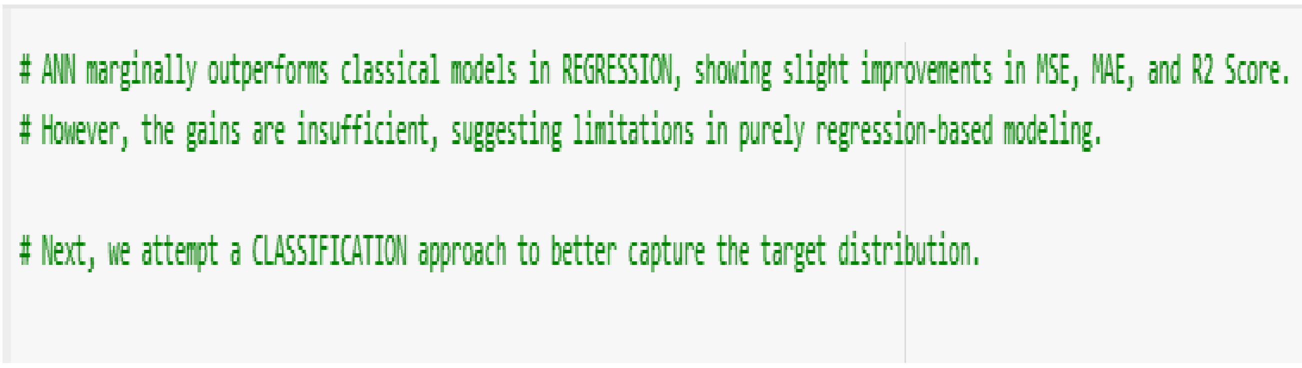


```
Model: "sequential_1"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃             Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━┩
│ dense_5 (Dense)                      │ (None, 64)                  │                 896 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────────┤
│ batch_normalization_1                │ (None, 64)                  │                 256 │
│ (BatchNormalization)                 │                             │                     │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────────┤
│ dropout_1 (Dropout)                  │ (None, 64)                  │                   0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────────┤
│ dense_6 (Dense)                      │ (None, 32)                  │               2,080 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────────┤
│ dense_7 (Dense)                      │ (None, 16)                  │                 528 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────────┤
│ dense_8 (Dense)                      │ (None, 8)                   │                 136 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────────┤
│ dense_9 (Dense)                      │ (None, 1)                   │                   9 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────────┘
 Total params: 11,461 (44.77 KB)
 Trainable params: 3,777 (14.75 KB)
 Non-trainable params: 128 (512.00 B)
 Optimizer params: 7,556 (29.52 KB)

817/817 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
817/817 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
817/817 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step
ANN Metrics:
             MSE        MAE  R2 Score
ANN  335.363495  14.568705  0.140552
```

Refining Our Approach - Classification



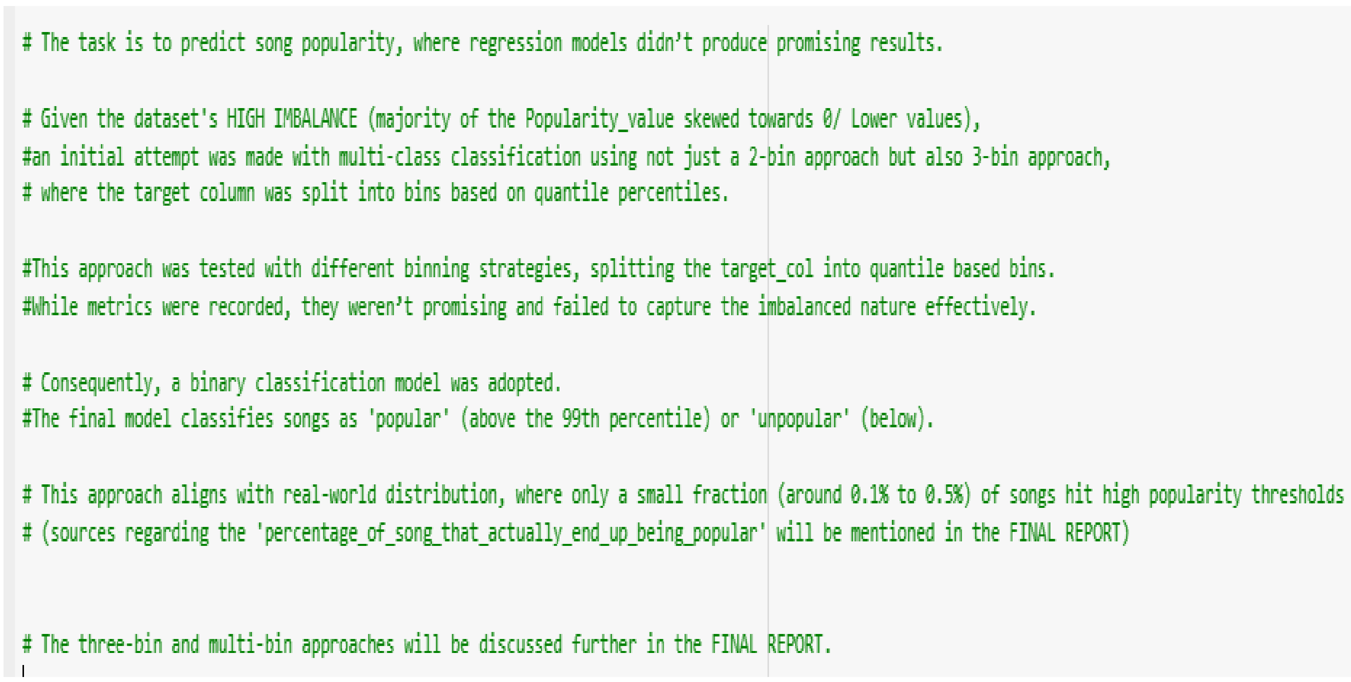
```
                                                 MSE        MAE  R2 Score
Linear Regression                         355.018002  15.547299  0.090183
Ridge Regression (α=1.0)                  355.018033  15.547302  0.090183
Ridge Regression (α=ridge_cv.best_params_) 355.021186  15.547522  0.090175
XGBoost                                   333.827484  14.990844  0.144489
AdaBoost                                  358.389974  15.860929  0.081542
Gradient Boosting                         333.908331  14.992583  0.144281
ANN                                       335.363495  14.568705  0.140552
```



```
# ANN marginally outperforms classical models in REGRESSION, showing slight improvements in MSE, MAE, and R2 Score.
# However, the gains are insufficient, suggesting limitations in purely regression-based modeling.

# Next, we attempt a CLASSIFICATION approach to better capture the target distribution.
```

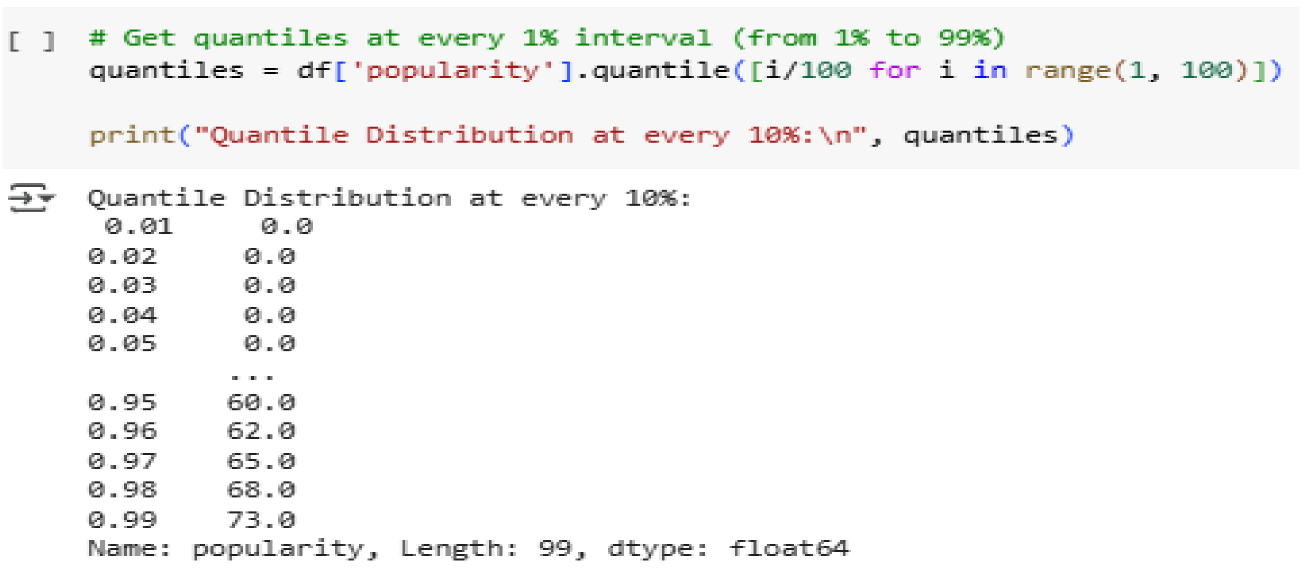
Feature Engineering: Quantile Binning Approach

DF



```
# The task is to predict song popularity, where regression models didn't produce promising results.

# Given the dataset's HIGH IMBALANCE (majority of the Popularity_value skewed towards 0/ Lower values),
#an initial attempt was made with multi-class classification using not just a 2-bin approach but also 3-bin approach,
# where the target column was split into bins based on quantile percentiles.

#This approach was tested with different binning strategies, splitting the target_col into quantile based bins.
#While metrics were recorded, they weren't promising and failed to capture the imbalanced nature effectively.

# Consequently, a binary classification model was adopted.
#The final model classifies songs as 'popular' (above the 99th percentile) or 'unpopular' (below).

# This approach aligns with real-world distribution, where only a small fraction (around 0.1% to 0.5%) of songs hit high popularity thresholds
# (sources regarding the 'percentage_of_song_that_actually_end_up_being_popular' will be mentioned in the FINAL REPORT)



# The three-bin and multi-bin approaches will be discussed further in the FINAL REPORT.
```

Getting the QUANTILES at every 1% interval



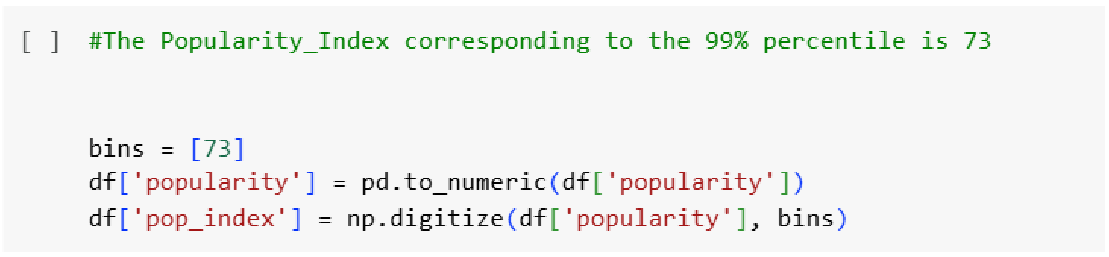
```
[ ]  # Get quantiles at every 1% interval (from 1% to 99%)
     quantiles = df['popularity'].quantile([i/100 for i in range(1, 100)])

     print("Quantile Distribution at every 10%:\n", quantiles)
```

```
Quantile Distribution at every 10%:
  0.01      0.0
0.02      0.0
0.03      0.0
0.04      0.0
0.05      0.0
         ...
0.95     60.0
0.96     62.0
0.97     65.0
0.98     68.0
0.99     73.0
Name: popularity, Length: 99, dtype: float64
```

Popularity_index = 73 corresponds to the 99% (TOP 1% OF SONGS)

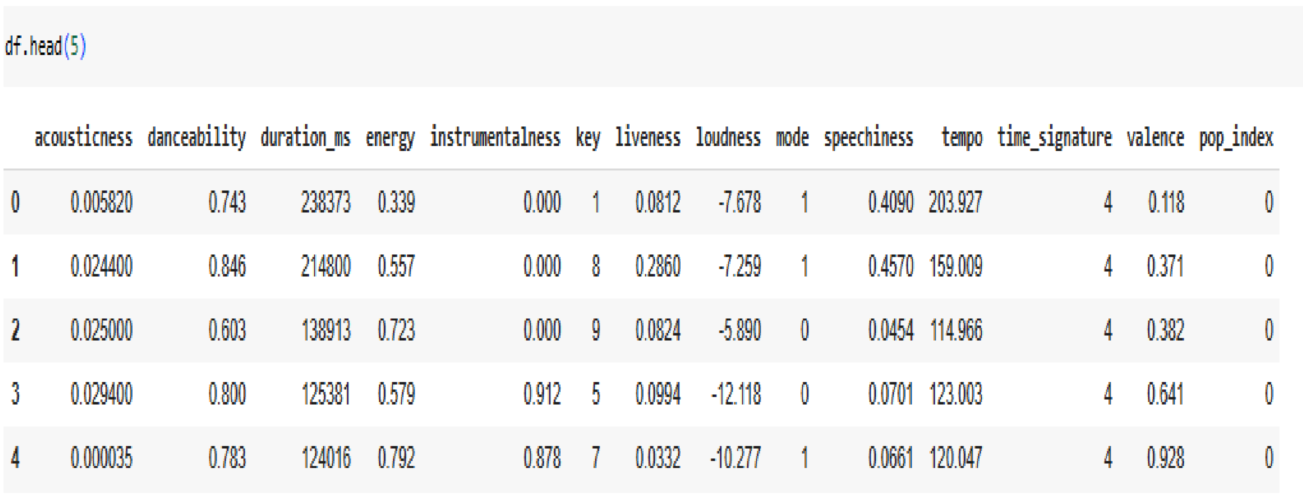
Given that global streaming data suggests that fewer than 1% of songs achieve significant popularity, a quantile binning approach was adopted with the top 1% (popularity index ≥ 73) classified as 'popular'. This aligns with real-world music consumption patterns, ensuring a realistic and industry-relevant classification.

DF



```
[ ]  #The Popularity_Index corresponding to the 99% percentile is 73


     bins = [73]
     df['popularity'] = pd.to_numeric(df['popularity'])
     df['pop_index'] = np.digitize(df['popularity'], bins)
```

Final dataset used in Classification purposes

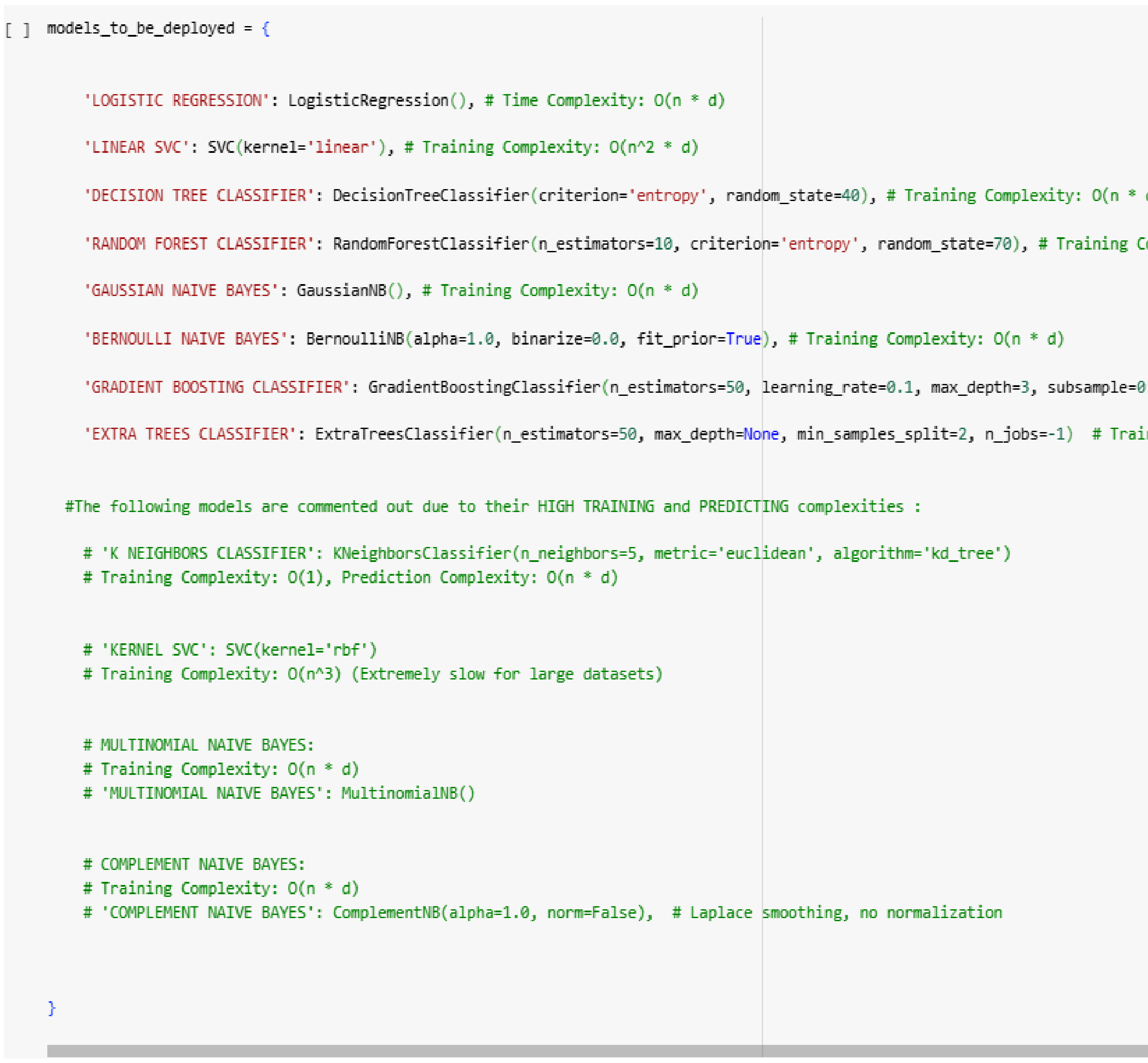


```
df.head(5)
```

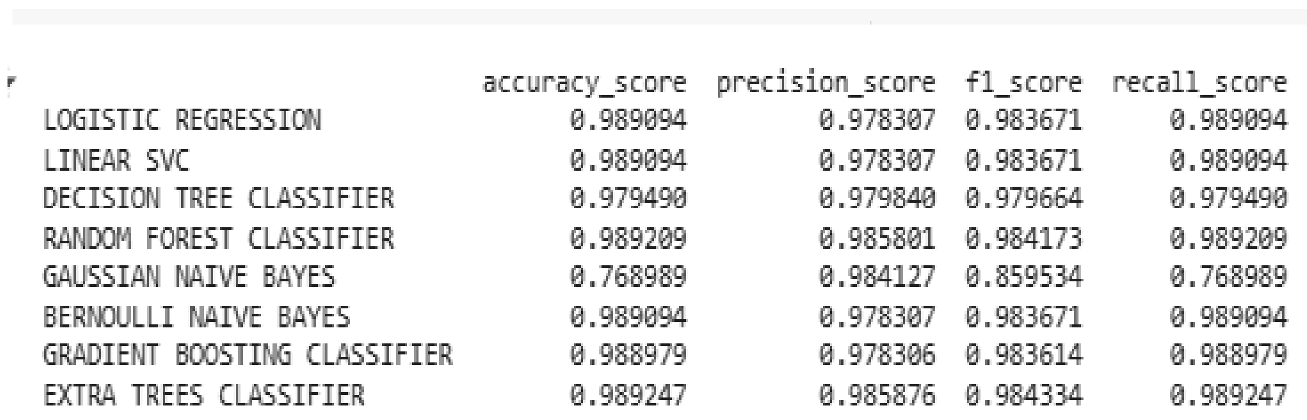
	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudness	mode	speechiness	tempo	time_signature	valence	pop_index
0	0.005820	0.743	238373	0.339	0.000	1	0.0812	-7.678	1	0.4090	203.927	4	0.118	0
1	0.024400	0.846	214800	0.557	0.000	8	0.2860	-7.259	1	0.4570	159.009	4	0.371	0
2	0.025000	0.603	138913	0.723	0.000	9	0.0824	-5.890	0	0.0454	114.966	4	0.382	0
3	0.029400	0.800	125381	0.579	0.912	5	0.0994	-12.118	0	0.0701	123.003	4	0.641	0
4	0.000035	0.783	124016	0.792	0.878	7	0.0332	-10.277	1	0.0661	120.047	4	0.928	0

DF

Classification using Classical Models

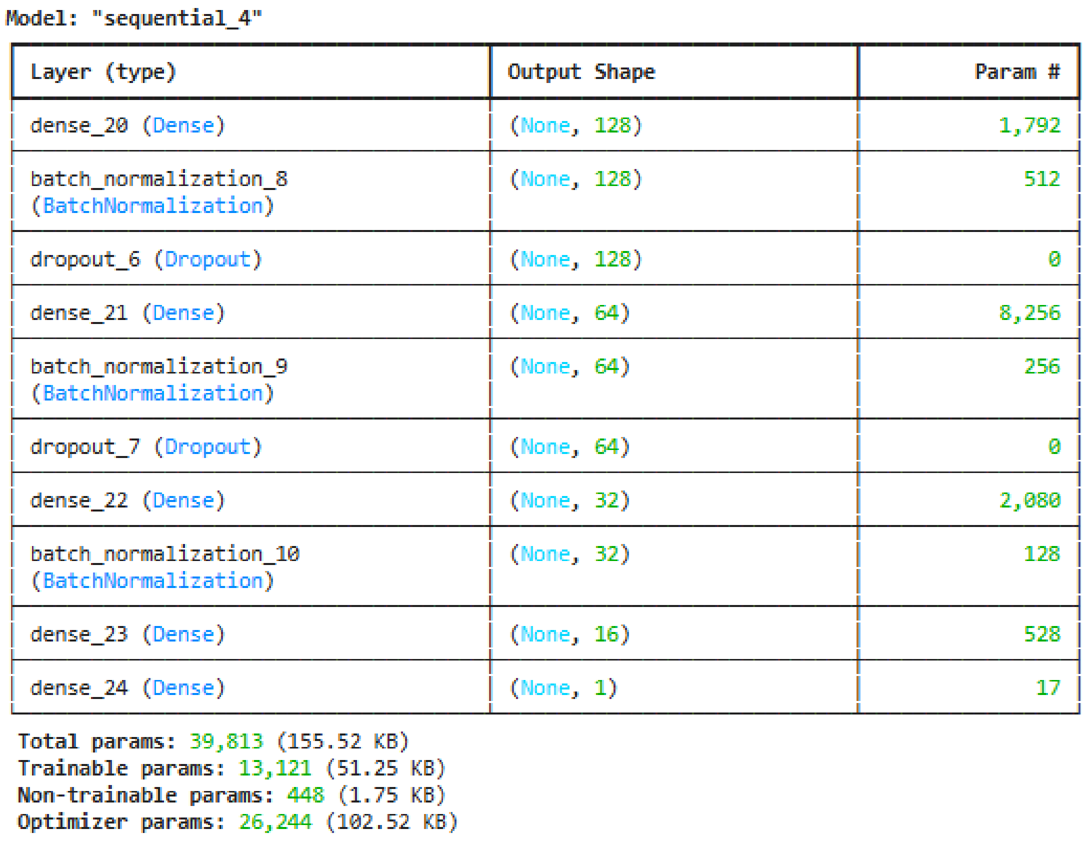


```
[ ]  models_to_be_deployed = {


         'LOGISTIC REGRESSION': LogisticRegression(), # Time Complexity: O(n * d)

         'LINEAR SVC': SVC(kernel='linear'), # Training Complexity: O(n^2 * d)

         'DECISION TREE CLASSIFIER': DecisionTreeClassifier(criterion='entropy', random_state=40), # Training Complexity: O(n * 

         'RANDOM FOREST CLASSIFIER': RandomForestClassifier(n_estimators=10, criterion='entropy', random_state=70), # Training C

         'GAUSSIAN NAIVE BAYES': GaussianNB(), # Training Complexity: O(n * d)

         'BERNOULLI NAIVE BAYES': BernoulliNB(alpha=1.0, binarize=0.0, fit_prior=True), # Training Complexity: O(n * d)

         'GRADIENT BOOSTING CLASSIFIER': GradientBoostingClassifier(n_estimators=50, learning_rate=0.1, max_depth=3, subsample=0

         'EXTRA TREES CLASSIFIER': ExtraTreesClassifier(n_estimators=50, max_depth=None, min_samples_split=2, n_jobs=-1)  # Trai


       #The following models are commented out due to their HIGH TRAINING and PREDICTING complexities :

         # 'K NEIGHBORS CLASSIFIER': KNeighborsClassifier(n_neighbors=5, metric='euclidean', algorithm='kd_tree')
         # Training Complexity: O(1), Prediction Complexity: O(n * d)


         # 'KERNEL SVC': SVC(kernel='rbf')
         # Training Complexity: O(n^3) (Extremely slow for large datasets)


         # MULTINOMIAL NAIVE BAYES:
         # Training Complexity: O(n * d)
         # 'MULTINOMIAL NAIVE BAYES': MultinomialNB()


         # COMPLEMENT NAIVE BAYES:
         # Training Complexity: O(n * d)
         # 'COMPLEMENT NAIVE BAYES': ComplementNB(alpha=1.0, norm=False),  # Laplace smoothing, no normalization



     }
```

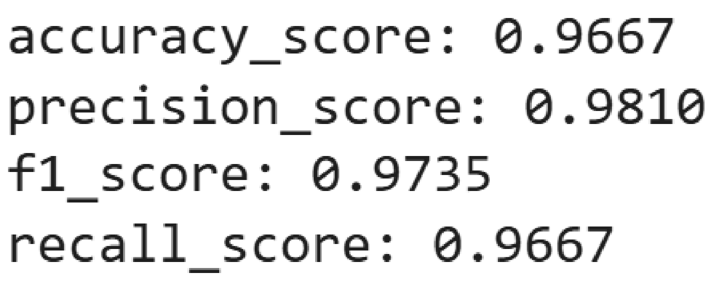


```
                              accuracy_score  precision_score  f1_score  recall_score
LOGISTIC REGRESSION                 0.989094         0.978307  0.983671      0.989094
LINEAR SVC                          0.989094         0.978307  0.983671      0.989094
DECISION TREE CLASSIFIER            0.979490         0.979840  0.979664      0.979490
RANDOM FOREST CLASSIFIER            0.989209         0.985801  0.984173      0.989209
GAUSSIAN NAIVE BAYES                0.768989         0.984127  0.859534      0.768989
BERNOULLI NAIVE BAYES               0.989094         0.978307  0.983671      0.989094
GRADIENT BOOSTING CLASSIFIER        0.988979         0.978306  0.983614      0.988979
EXTRA TREES CLASSIFIER              0.989247         0.985876  0.984334      0.989247
```

Classification using ANNs



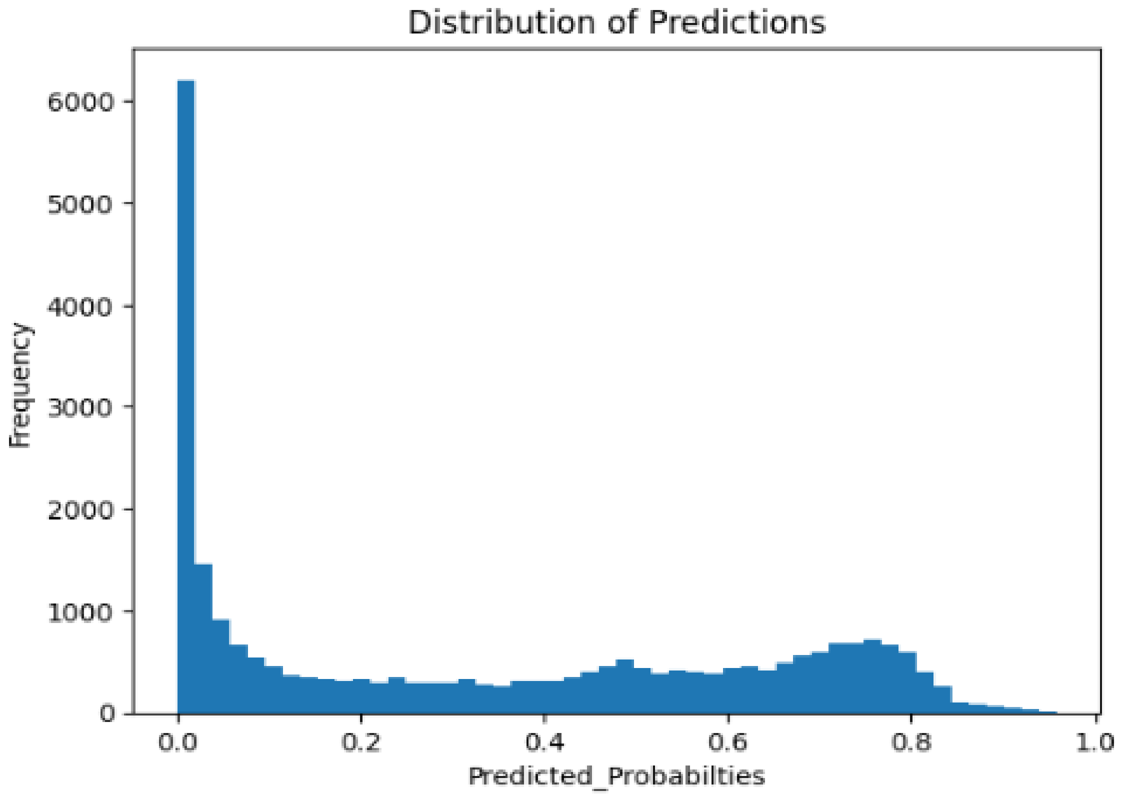
```
Model: "sequential_4"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense_20 (Dense)                │ (None, 128)            │         1,792 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_8           │ (None, 128)            │           512 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_6 (Dropout)             │ (None, 128)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_21 (Dense)                │ (None, 64)             │         8,256 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_9           │ (None, 64)             │           256 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_7 (Dropout)             │ (None, 64)             │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_22 (Dense)                │ (None, 32)             │         2,080 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_10          │ (None, 32)             │           128 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_23 (Dense)                │ (None, 16)             │           528 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_24 (Dense)                │ (None, 1)              │            17 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 39,813 (155.52 KB)
 Trainable params: 13,121 (51.25 KB)
 Non-trainable params: 448 (1.75 KB)
 Optimizer params: 26,244 (102.52 KB)
```



```
accuracy_score: 0.9667
precision_score: 0.9810
f1_score: 0.9735
recall_score: 0.9667
```

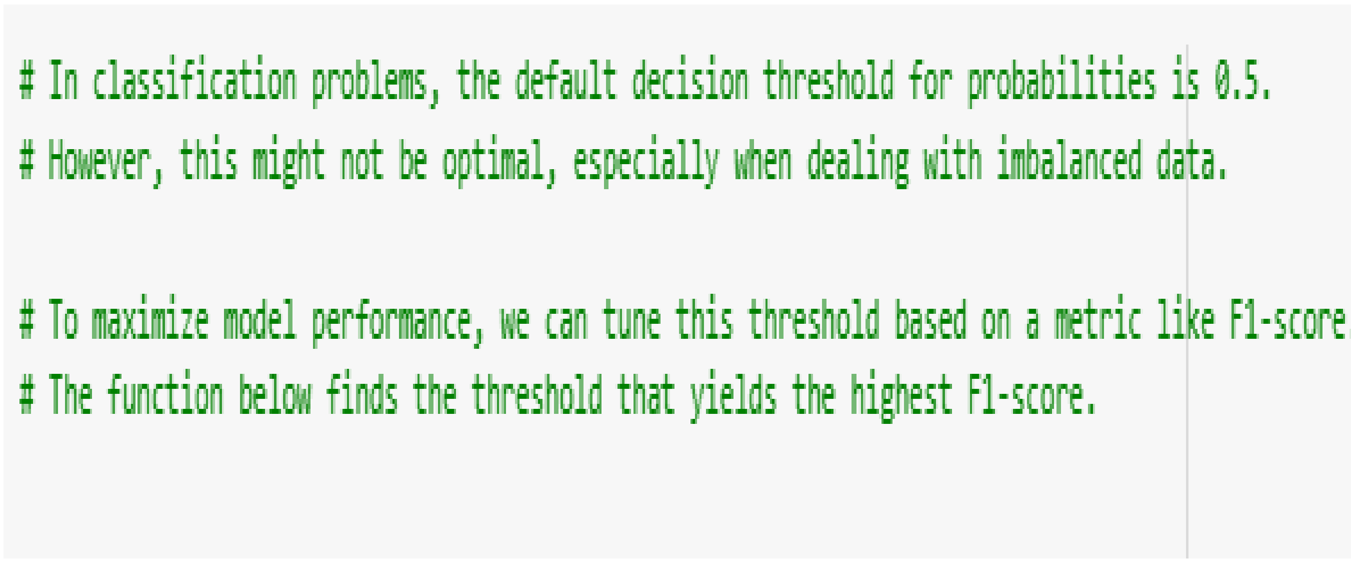
The above are the metrics of ANN model applied on the dataset for the purpose of classification

Below is the distribution of Predicted_Probabilites against their frequency

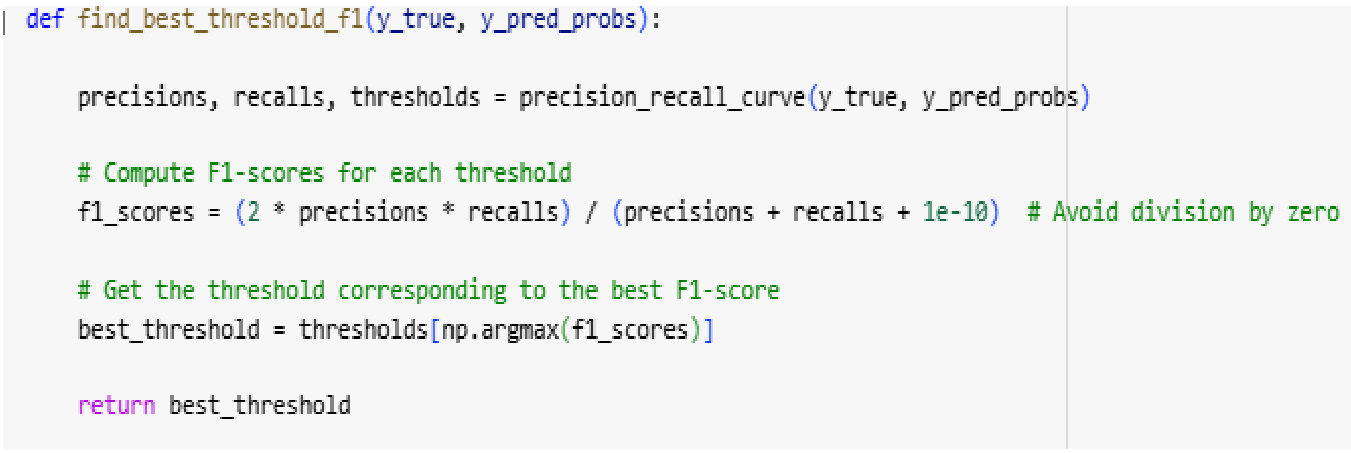


DF

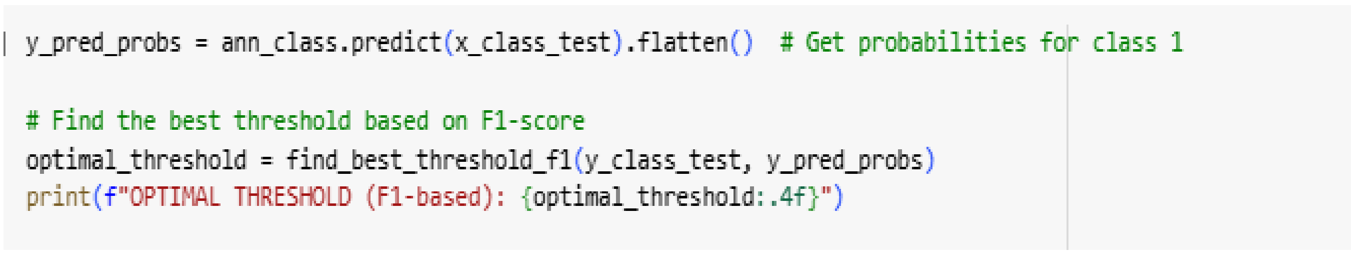
F1-Based Threshold Optimizer



```
# In classification problems, the default decision threshold for probabilities is 0.5.
# However, this might not be optimal, especially when dealing with imbalanced data.

# To maximize model performance, we can tune this threshold based on a metric like F1-score
# The function below finds the threshold that yields the highest F1-score.
```



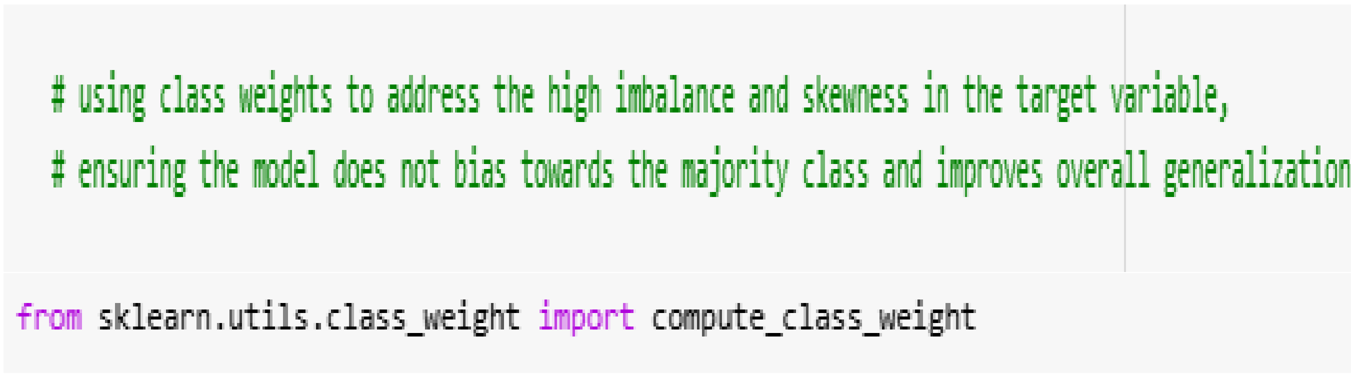
```
def find_best_threshold_f1(y_true, y_pred_probs):

    precisions, recalls, thresholds = precision_recall_curve(y_true, y_pred_probs)

    # Compute F1-scores for each threshold
    f1_scores = (2 * precisions * recalls) / (precisions + recalls + 1e-10)  # Avoid division by zero

    # Get the threshold corresponding to the best F1-score
    best_threshold = thresholds[np.argmax(f1_scores)]

    return best_threshold
```



```
y_pred_probs = ann_class.predict(x_class_test).flatten()  # Get probabilities for class 1

# Find the best threshold based on F1-score
optimal_threshold = find_best_threshold_f1(y_class_test, y_pred_probs)
print(f"OPTIMAL THRESHOLD (F1-based): {optimal_threshold:.4f}")
```

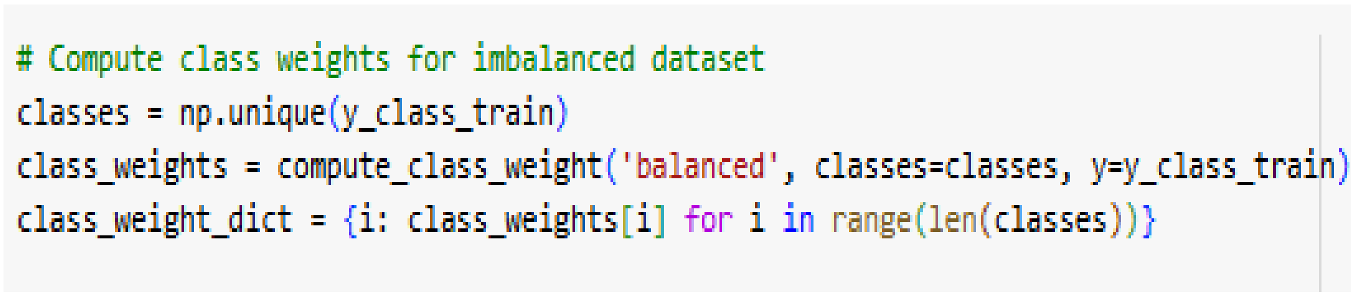
DF

Refining Our Classical Approach: Using Class Weights to Counter Class Imbalance



```
# using class weights to address the high imbalance and skewness in the target variable,
# ensuring the model does not bias towards the majority class and improves overall generalization
```

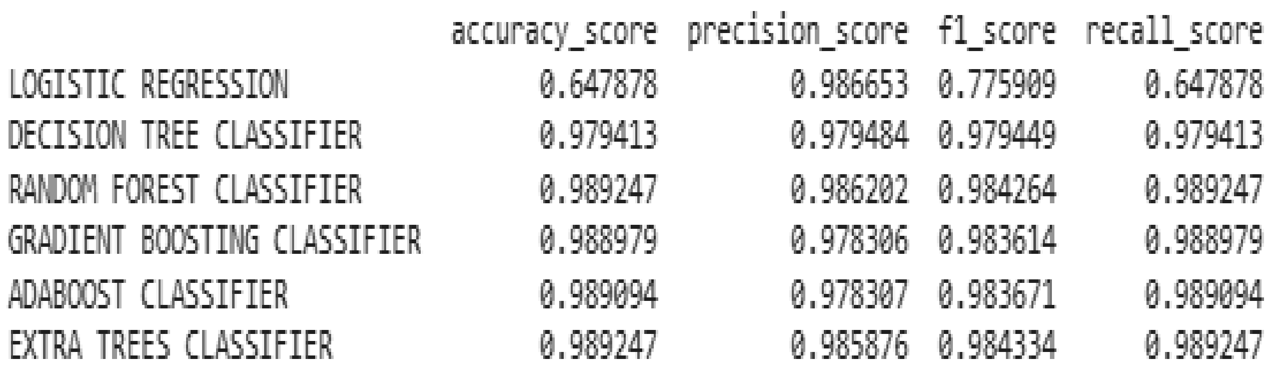
```
from sklearn.utils.class_weight import compute_class_weight
```



```
# Compute class weights for imbalanced dataset
classes = np.unique(y_class_train)
class_weights = compute_class_weight('balanced', classes=classes, y=y_class_train)
class_weight_dict = {i: class_weights[i] for i in range(len(classes))}
```

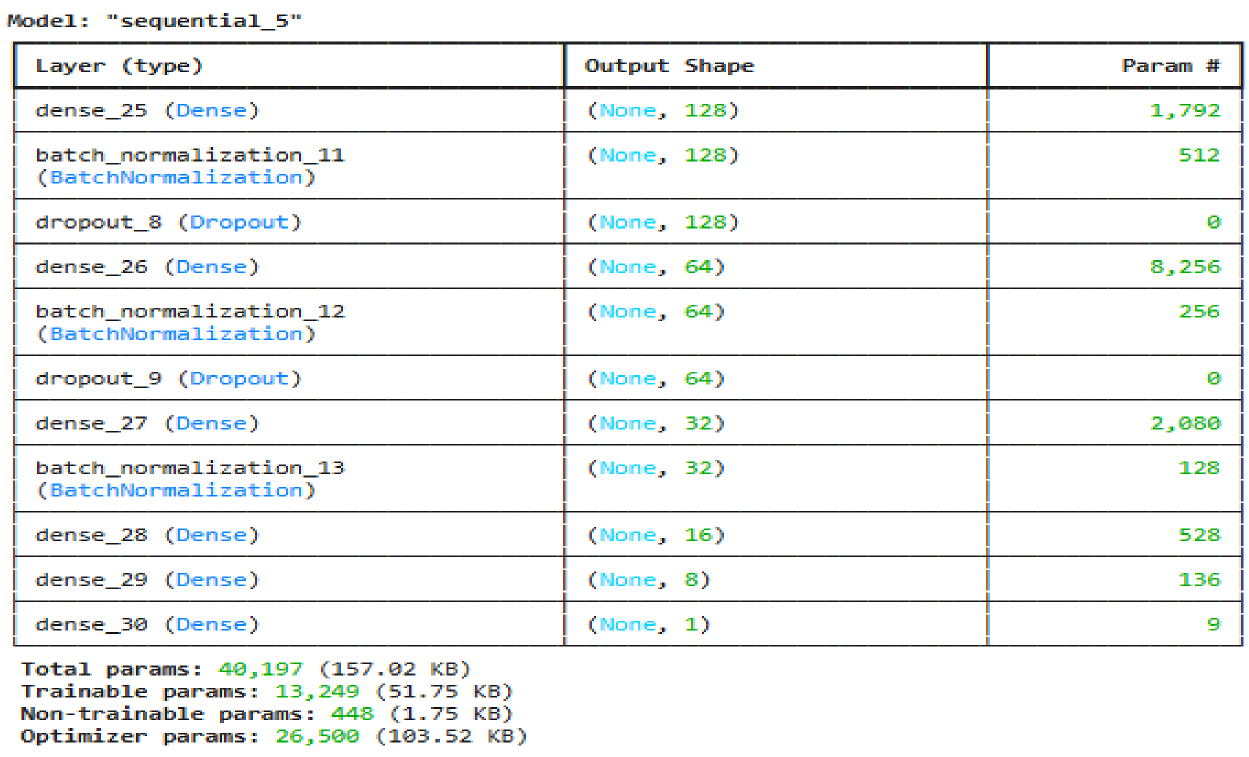
Classification using Classical Models (Revised Approach)

Classifiers used : *Logistic_Regression, Decision_Tree_Classifier, Random_Forest_Classifier, Gradient_Boosting_Classifier, ADABoost_Classifier, Extra_Trees_Classifier*



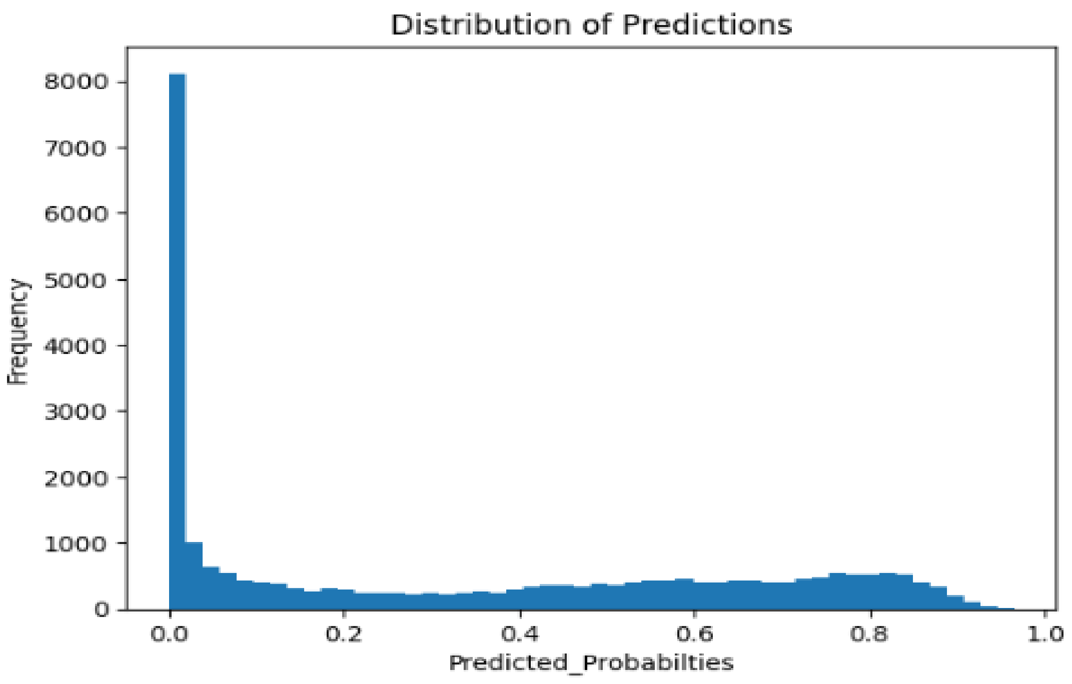
```
                            accuracy_score  precision_score  f1_score  recall_score
LOGISTIC REGRESSION               0.647878         0.986653  0.775909      0.647878
DECISION TREE CLASSIFIER          0.979413         0.979484  0.979449      0.979413
RANDOM FOREST CLASSIFIER          0.989247         0.986202  0.984264      0.989247
GRADIENT BOOSTING CLASSIFIER      0.988979         0.978306  0.983614      0.988979
ADABOOST CLASSIFIER               0.989094         0.978307  0.983671      0.989094
EXTRA TREES CLASSIFIER            0.989247         0.985876  0.984334      0.989247
```

Classification using ANNs (Revised Approach)



```
Model: "sequential_5"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ dense_25 (Dense)                     │ (None, 128)                 │           1,792 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_11               │ (None, 128)                 │             512 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_8 (Dropout)                  │ (None, 128)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_26 (Dense)                     │ (None, 64)                  │           8,256 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_12               │ (None, 64)                  │             256 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dropout_9 (Dropout)                  │ (None, 64)                  │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_27 (Dense)                     │ (None, 32)                  │           2,080 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization_13               │ (None, 32)                  │             128 │
│ (BatchNormalization)                 │                             │                 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_28 (Dense)                     │ (None, 16)                  │             528 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_29 (Dense)                     │ (None, 8)                   │             136 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_30 (Dense)                     │ (None, 1)                   │               9 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 40,197 (157.02 KB)
 Trainable params: 13,249 (51.75 KB)
 Non-trainable params: 448 (1.75 KB)
 Optimizer params: 26,500 (103.52 KB)
```

```
accuracy_score: 0.9687
precision_score: 0.9808
f1_score: 0.9745
recall_score: 0.9687
```


Distribution of Predictions
Frequency
Predicted_Probabilties