Music Genre Classification 2024

Introduction & Background

- Music genre classification is an essential task in music information retrieval (MIR) that helps in organizing, recommending, and searching for music efficiently. Given the vast amount of music available today, automating the task of genre identification has become crucial for music streaming platforms, researchers, and developers alike.
  - Traditional methods of music classification often relied on manual tagging and expert knowledge, which were time-consuming and subject to human error. With the advent of machine learning, it has become possible to automate this process by analyzing audio features to predict the genre of a song. The challenge lies in accurately capturing the nuances of different genres and ensuring the model generalizes well across diverse datasets. This project leverages advanced machine learning techniques to tackle this problem, aiming for high accuracy and efficiency in genre classification.

Framing the Problem

In the realm of music classification, understanding the intricate relationship between various audio features and musical genres is paramount. Leveraging domain knowledge reveals that genres are influenced by several key characteristics such as tempo, loudness, valence, and more. Each of these features contributes uniquely to the overall sound and identity of a song. Utilizing this domain knowledge, the task at hand is to develop a predictive model capable of accurately classifying songs into their respective genres based on these fundamental audio features. By framing the problem in this context, we aim to explore the relationships between these audio features and musical genres, uncovering patterns that can enhance the ability to categorize music effectively.

By leveraging data analytics and machine learning, we aim to develop a robust model for accurate music genre classification. This will lead to improved music recommendations and discovery tools, enhancing the overall music experience.

Objectives

- **Primary Objective:** Develop a machine learning model capable of accurately classifying music into genres based on available audio features.

  - **Specific Objectives:**

1. **Exploratory Data Analysis (EDA):** Analyze and visualize the provided dataset to gain insights into the feature distribution and relationships.
2. **Data Preprocessing:** Prepare the data for modeling by handling missing values, scaling features to optimize model performance.
3. **Model Selection & Evaluation:**

- Compare different classification algorithms, including CatBoost, XGBoost, LightGBM, and Random Forest.
- Explore and implement ensemble techniques, such as stacking, to improve the final classification model's performance.
- Evaluate model performance using appropriate metrics, focusing on the F1 score as the main measure of success.

Dataset Overview

- **Source:** The dataset for this project comes from a Kaggle competition and contains 14,396 entries.
  - The dataset consists of 17 features, which capture various audio characteristics of songs, and a target column 'Class' indicating the genre of each song.
- Our project revolves around a dataset comprising data on over **14,396** rows. It includes essential audio attributes such as danceability, energy, loudness, acousticness, instrumentalness, and tempo, alongside categorical features like key, mode, and time signature. This real-world dataset serves as the foundation for our machine learning efforts, aiming to extract insights and build a robust music genre classification model.
- A separate test dataset of 3,600 rows with 17 features (excluding the target variable) is used to evaluate the performance of our final model.

Selecting a Performance Measure

To gauge the effectiveness of our music genre classification models, we opt for the F1-score as our primary performance measure.

The F1-score is a powerful metric that balances precision (the proportion of correctly predicted positive classifications) and recall (the proportion of actual positives that were correctly classified). It can be interpreted as the harmonic mean of precision and recall, calculated as follows:

F1 = 2 * (precision * recall) / (precision + recall)

By maximizing the F1-score, we strive to enhance the accuracy of our models across all music genres, ensuring a robust and balanced classification performance. This metric is particularly relevant due to the potential for class imbalance in our dataset, where some genres might be more prevalent than others.

Exploratory Data Analysis (EDA).

Exploring the dataset is a fundamental step in understanding its characteristics and preparing for further analysis.

Here's an overview of our exploratory analysis:

1. **Previewing Data:** We start by examining the first few rows of the dataset to get a glimpse of its structure and content.

```
# Display the first five rows of the dataset
train_data.head()
```

2. **Data Summary:** Utilizing the info() function, we obtain a summary of the dataset, including the data types and non-null counts for each column as in Pic 1:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14396 entries, 0 to 14395
Data columns (total 18 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Id                  14396 non-null  int64
 1   Artist Name         14396 non-null  object
 2   Track Name          14396 non-null  object
 3   Popularity          14063 non-null  float64
 4   danceability        14396 non-null  float64
 5   energy              14396 non-null  float64
 6   key                 12787 non-null  float64
 7   loudness            14396 non-null  float64
 8   mode                14396 non-null  int64
 9   speechiness         14396 non-null  float64
 10  acousticness        14396 non-null  float64
 11  instrumentalness    10855 non-null  float64
 12  liveness            14396 non-null  float64
 13  valence             14396 non-null  float64
 14  tempo               14396 non-null  float64
 15  duration_in min/ms  14396 non-null  float64
 16  time_signature      14396 non-null  int64
 17  Class               14396 non-null  int64
dtypes: float64(12), int64(4), object(2)
memory usage: 2.0+ MB
```

Pic 1: 2. Data Summary: info() function

Here's an overview of our exploratory analysis:

3. **Number of Instances and Features:** We ascertain the total number of instances and features in

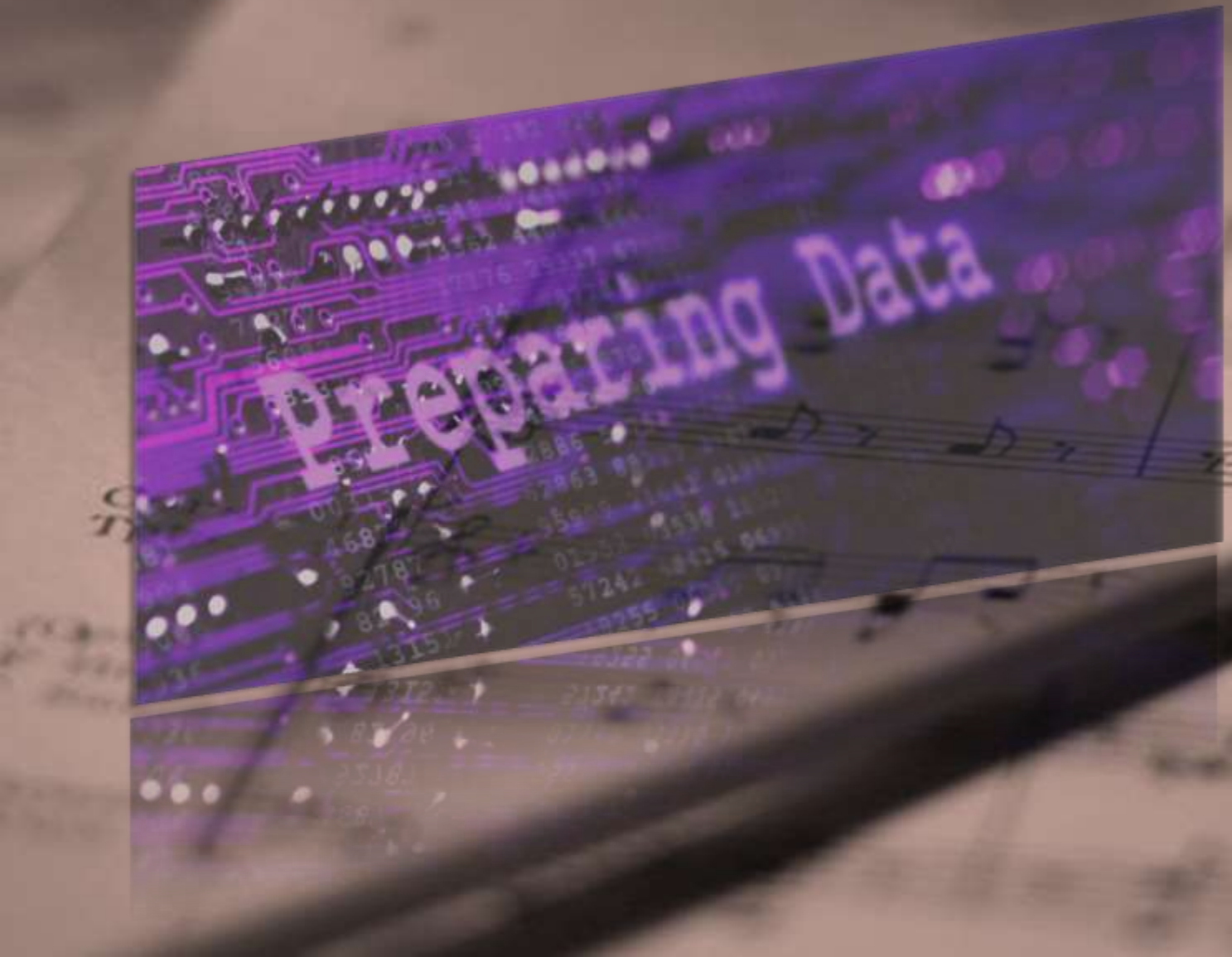
```
Number of instances: 14396
Number of features: 18
```

4. **Identify all categorical variables:**

```
Index(['Artist Name', 'Track Name'], dtype='object')
```

As seen in the output of the code, both 'Artist Name' and 'Track Name' are already identified as categorical variables (object type).

Preparing the Data



Data Preprocessing - Preparing the Data

1. Dropping Irrelevant Columns

The "Id" column is dropped as it's a unique identifier and doesn't contribute to predicting the music genre. Removing it simplifies the dataset and helps the model focus on relevant features.

Getting the number of instances and features after dropping irrelevant columns ‘Id’ :

```
Number of instances: 14396
Number of features: 17
```

Data Preprocessing - Preparing the Data

2. Checking for Duplicate Rows

- **Dealing with Duplicates:** Identifying and removing duplicate entries to ensure data integrity.

Preparing the Data

- **Data Cleaning**
  - **Checking for Duplicate Rows**

```
[ ]  1 # Counting Duplicate Rows
     2 train_data.duplicated().sum()
```

```
0
```

Executed the code above to check for duplicates in the dataset. The result was , confirming that there are no duplicate entries present in the dataset. This ensures the integrity and uniqueness of each data point for subsequent analysis and model training.

Data Preprocessing - Preparing the Data

3. Checking for Missing Values

```
Id                      0
Artist Name             0
Track Name              0
Popularity            333
danceability            0
energy                  0
key                  1609
loudness                0
mode                    0
speechiness             0
acousticness            0
instrumentalness     3541
liveness                0
valence                 0
tempo                   0
duration_in min/ms      0
time_signature          0
Class                   0
dtype: int64
```

Data Preprocessing | Preparing the Data – Data Cleaning

4. Handling Missing Values

Handling missing values for numerical features

```
[19]  1 # Handling missing values for numerical features
      2
      3 # Impute 'Popularity' and 'instrumentalness' with their median values
      4 train_data['Popularity'].fillna(train_data['Popularity'].median(), inplace=True)
      5 train_data['instrumentalness'].fillna(train_data['instrumentalness'].median(), inplace=True)
```

Handling missing values for categorical feature

```
[20]  1 # Handling missing values for categorical feature 'key'
      2
      3 # Impute 'key' with the most frequent value
      4 train_data['key'].fillna(train_data['key'].mode()[0], inplace=True)
```

Final Check for Missing Values

```
      1 # Verify that there are no missing values left
      2 print(train_data.isnull().sum().sum())
```

```
0
```

Preparing the Data

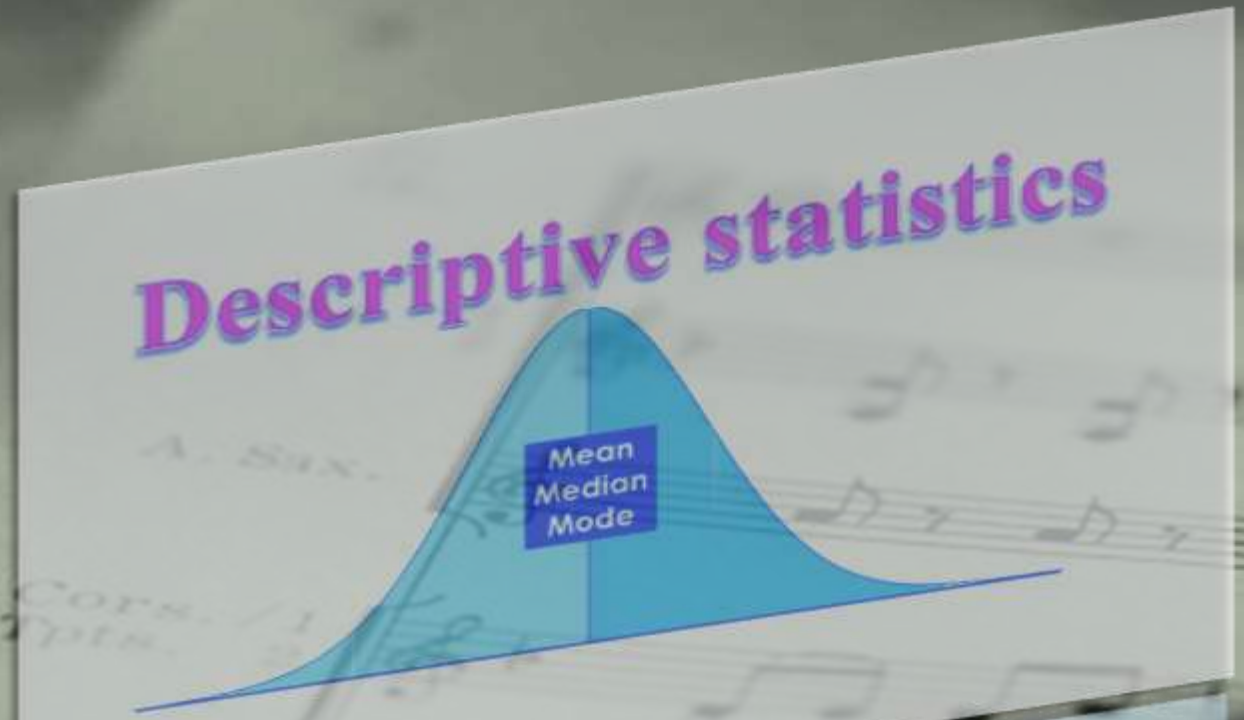
1. Creating Feature Lists: Defining num_cols and cat_cols lists helps organize and distinguish numerical and categorical features.

```
num_cols = ['Popularity', 'danceability', 'energy', 'loudness',
            'speechiness', 'acousticness', 'instrumentalness', 'liveness',
            'valence', 'tempo', 'duration_in min/ms']

cat_cols = ['Artist Name','Track Name','mode', 'key', 'time_signature', 'Class']
```

While the info() method might display certain features as numerical (float), it's crucial to recognize the underlying nature of the data for proper analysis. The following features, despite numerical representation, are fundamentally categorical:

- **'mode':** Represents the modality (major or minor) of a track, a categorical concept with distinct, non-ordinal values.
- **'key':** Utilizes Pitch Class notation, where integers map to distinct musical keys. While numerically represented, the relationship between these values is not continuous or ordinal.
- **'time_signature':** Indicates the rhythmic structure of a musical piece (e.g., 4/4, 3/4). While numerically represented, these values represent distinct categories, not a continuous scale.
- **'Class':** The target variable itself, representing the music genre, is inherently a categorical feature.

Descriptive Statistics



Descriptive Statistics

1. Explore Summary Statistics for Numerical Columns

- **Measures of Central Tendency (mean, median(50%))**
- **Measures of Dispersion (Standard Deviation(std))**
- **Measures of Position (Quartile)**

	Popularity	danceability	energy	loudness	speechiness	acousticness	instrumentalness	liveness	valence	tempo	duration_in min/ms
count	14396.000000	14396.000000	14396.000000	14396.000000	14396.000000	14396.000000	14396.000000	14396.000000	14396.000000	14396.000000	1.439600e+04
mean	44.513059	0.543105	0.662422	-7.900852	0.080181	0.246746	0.135278	0.195782	0.486379	122.695372	2.000942e+05
std	17.216466	0.165517	0.235967	4.057362	0.085157	0.310922	0.274652	0.159258	0.239476	29.538490	1.116891e+05
min	1.000000	0.059600	0.001210	-39.952000	0.022500	0.000000	0.000001	0.011900	0.021500	30.557000	5.016500e-01
25%	33.000000	0.432000	0.508000	-9.538000	0.034800	0.004280	0.000295	0.097275	0.299000	99.799000	1.654458e+05
50%	44.000000	0.545000	0.699000	-7.013500	0.047100	0.081450	0.003920	0.129000	0.480500	120.060000	2.089410e+05
75%	56.000000	0.658000	0.861000	-5.162000	0.083100	0.432250	0.057600	0.256000	0.672000	141.988250	2.522470e+05
max	100.000000	0.989000	1.000000	1.342000	0.955000	0.996000	0.996000	0.992000	0.986000	217.416000	1.477187e+06

✓ 0s completed at 4:56 PM

Descriptive Statistics

2. Explore summary statistics for categorical columns

	0
Artist Name	7913
Track Name	12455
mode	2
key	11
time_signature	4
Class	11

dtype: int64

Descriptive Statistics

2. Explore summary statistics for categorical columns

2.1 ‘time_signature’

```
Unique values in categorical columns:
'time_signature' column: [4 3 5 1]
```

- Measures of Frequency

```
time_signature
4    13149
3      994
5      166
1       87
Name: count, dtype: int64
```

Descriptive Statistics

2. Explore summary statistics for categorical columns

2.2 ‘mode’

```
Unique values in categorical columns:

'mode' column: [0 1]
```

- Measures of Frequency

```
mode
1    9217
0    5179
Name: count, dtype: int64
```

Descriptive Statistics

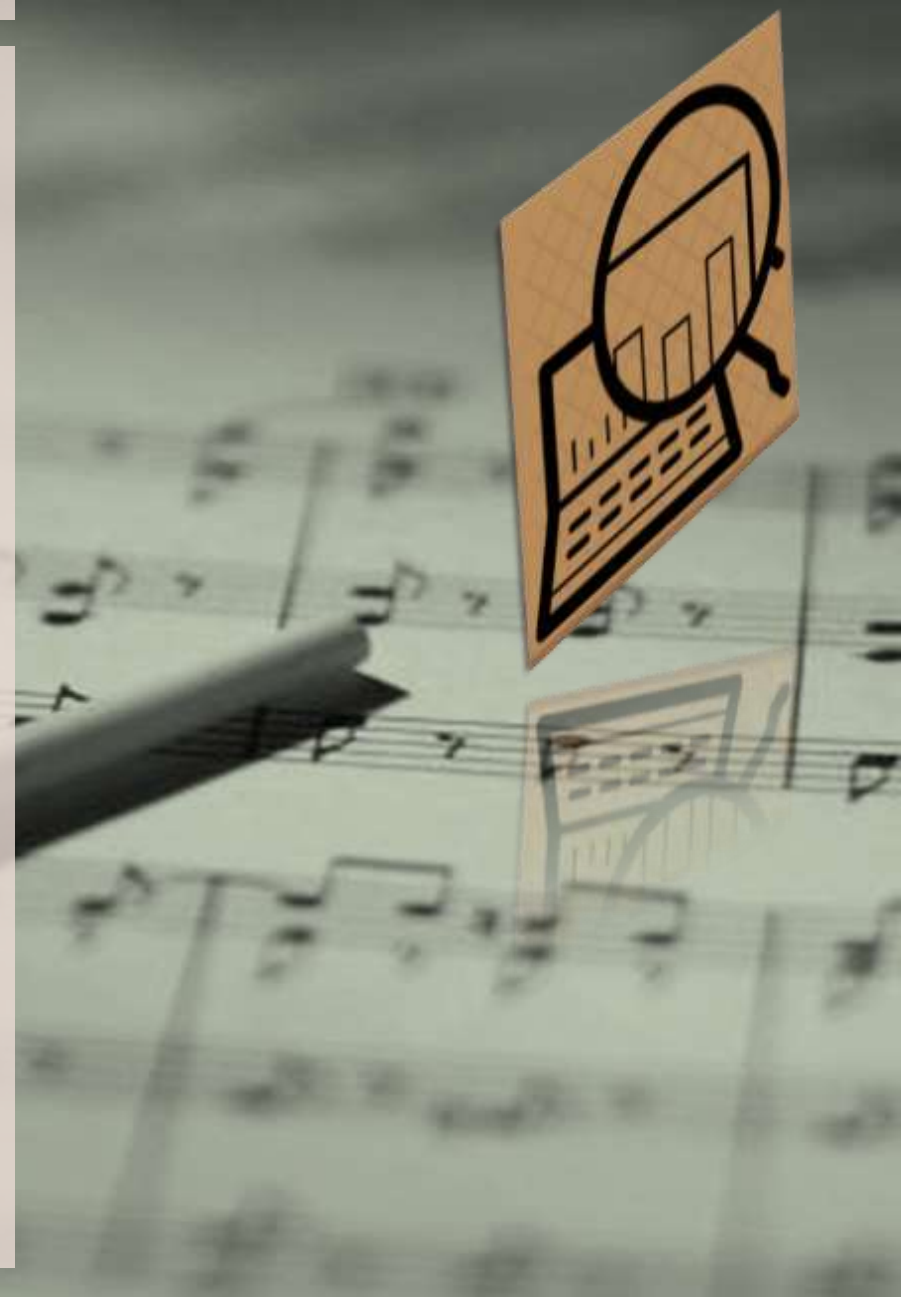
2. Explore summary statistics for categorical columns

2.3 ‘key’

```
Unique values in categorical columns:

'key' column: [ 9. 11.  7.  6.  1.  5. 10.  4.  2.  3.  8.]
```

- Measures of Frequency

```
key
7.0     3259
9.0     1590
2.0     1582
1.0     1351
4.0     1252
11.0    1176
5.0     1115
6.0      963
8.0      872
10.0     825
3.0      411
Name: count, dtype: int64
```



Descriptive Statistics

2. Explore summary statistics for categorical columns

2.4 ‘Class’

for Categorical Features

```
Unique values in categorical columns:

'Class' column: [ 9  6 10  2  5  0  8  4  3  1  7]
```

- Measures of Frequency

```
Class
10      3959
6       2069
9       2019
8       1483
5       1157
1       1098
2       1018
0        500
7        461
3        322
4        310
Name: count, dtype: int64
```

2.5 Measures of Central Tendency (Mode)

```
Mode of 'Artist Name': Backstreet Boys
Mode of 'Track Name': Fire
Mode of 'mode': 1
Mode of 'key': 7.0
Mode of 'time_signature': 4
Mode of 'Class': 10
```

Descriptive Statistics

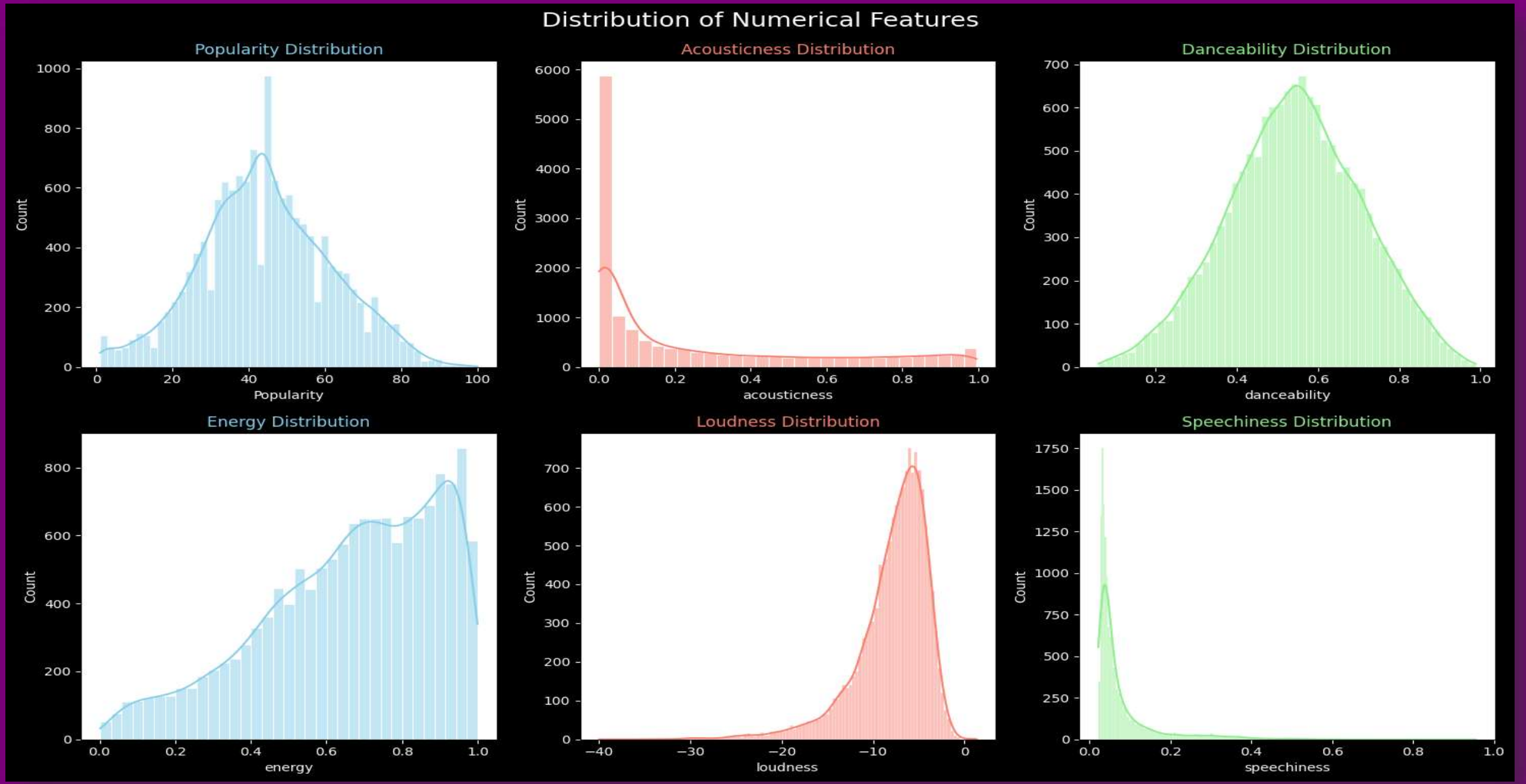
2.6 Measures of Association for Categorical Features

	Artist Name	Track Name	mode	key	time_signature
Chi-Squared Statistic	122864.112817	129513.466669	334.178039	580.185196	524.372274
P-value	{0.0}	{3.901223630615114e-23}	{9.040655067533138e-66}	{9.431195570749923e-69}	{1.1963814955329537e-91}

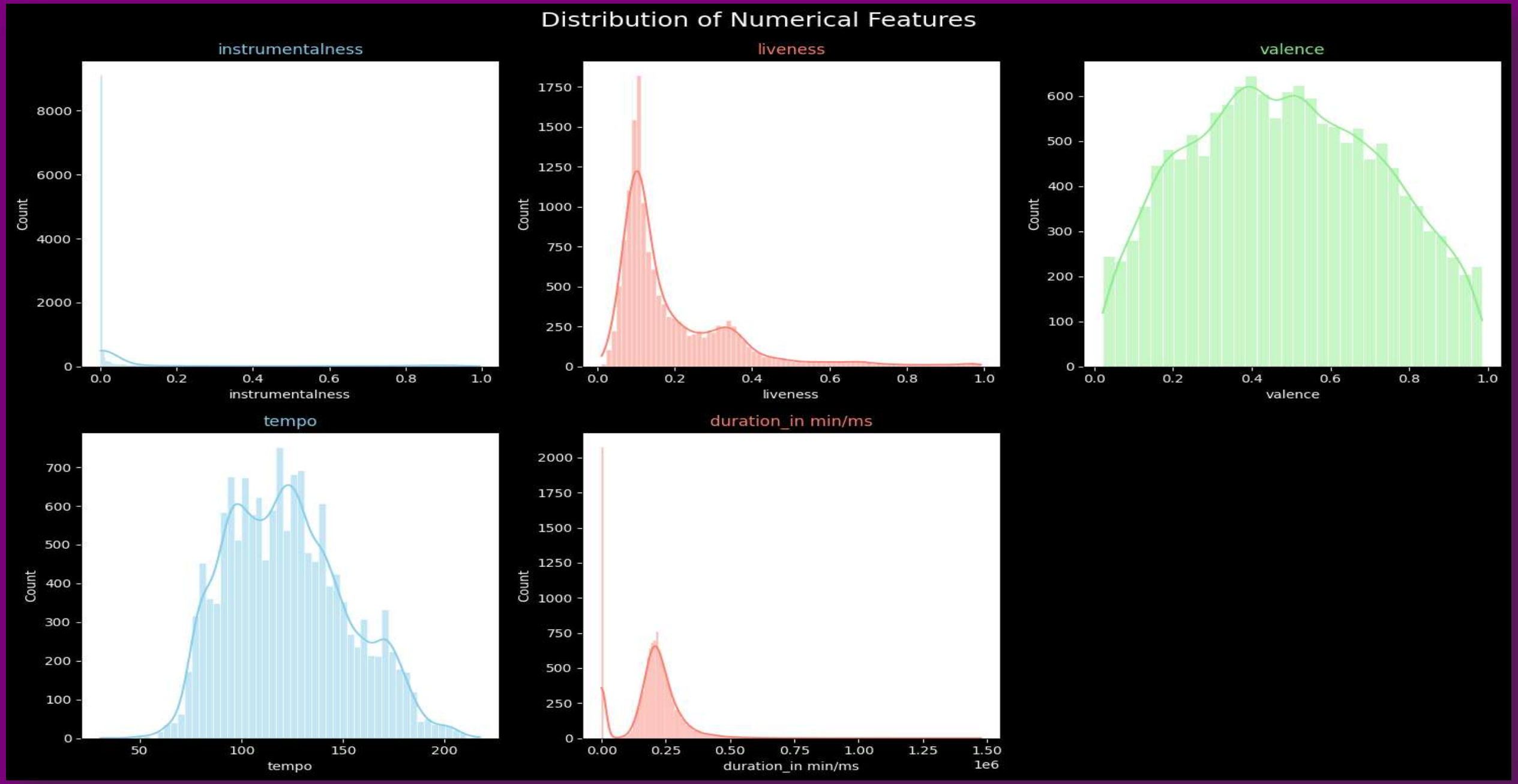
- Artist Name & Track Name: Exhibit extremely high Chi-Squared values and near-zero p-values, indicating a strong but likely overfit association with genre. These features might not generalize well for prediction.
- Mode, Key & Time Signature: Show significant Chi-Squared values and extremely low p-values, suggesting a strong and statistically significant association with genre. These features are likely to be valuable predictors in a genre classification model.
- In conclusion, while all features show a statistically significant association with genre, "mode", "key", and "time_signature" are more promising for building a generalizable genre classification model due to their musical relevance and less specific nature compared to "Artist Name" and "Track Name".

Visualization

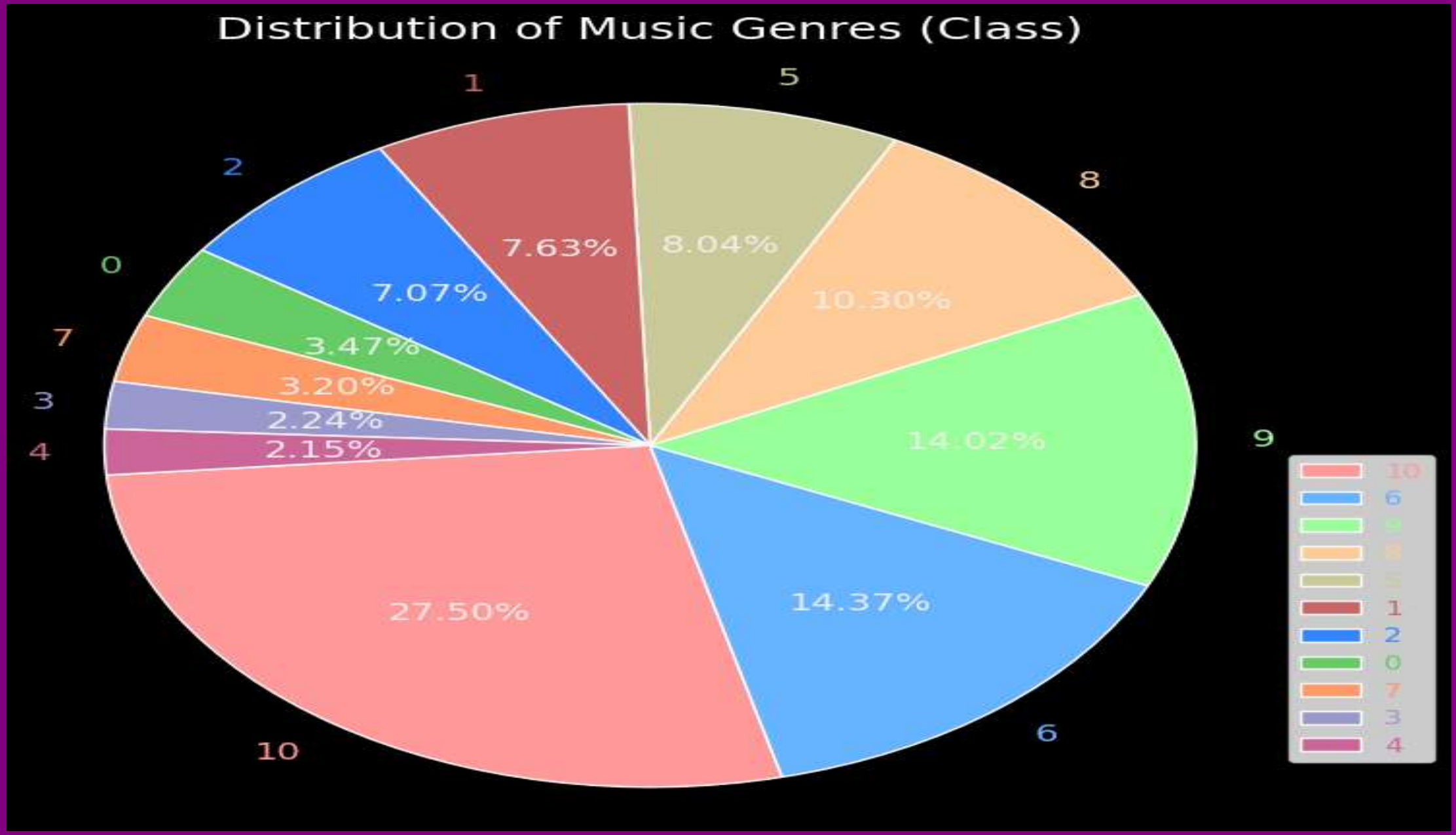
1.1 Exploring Numerical Feature Distributions



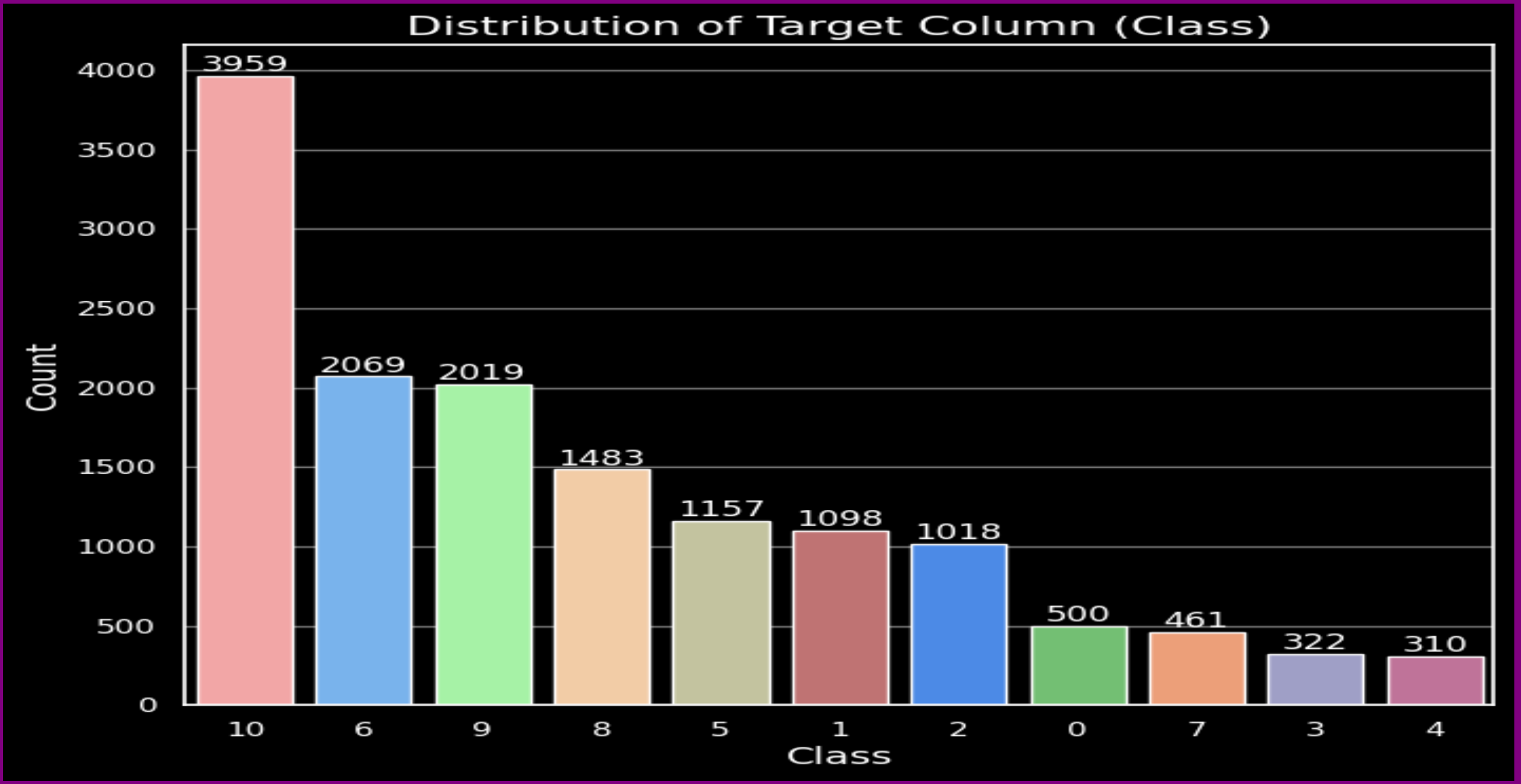
1.2 Exploring Numerical Feature Distributions



2.1 Distribution of Music Genres



2.2 Distribution of Music Genres



Insights: Distribution of Music Genres 'Class’

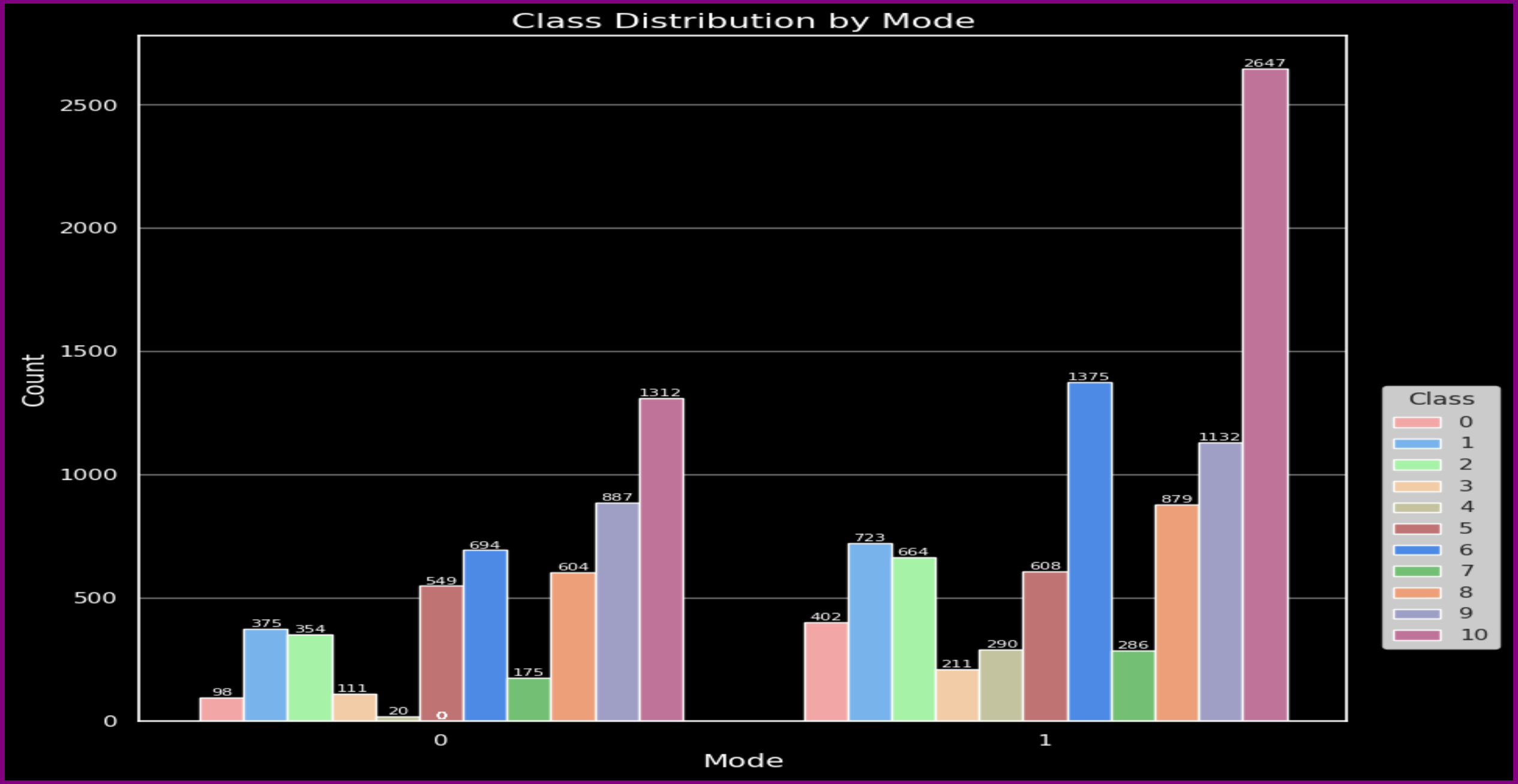
1- Insight: Class Distribution and Imbalance

A bar chart visualization of the target variable ('Class') reveals a significant class imbalance in the dataset. Some music genres are heavily represented (e.g., Class 10, Class 6), while others are underrepresented (e.g., Class 4, Class 3). This imbalance will be addressed during model training using techniques like resampling or cost-sensitive learning to ensure accurate predictions across all genres.

2- Insight: Some Music Genres Are More Common Than Others

We found that some music genres appear much more often in our data than others. This could be a problem when we build our music genre prediction model, as it might not learn to recognize the less common genres as well. We'll need to use special techniques to make sure the model learns from all genres, even the rare ones.

3. Visualization for Class Distribution by Mode



Insights: 'Class' Distribution by 'Mode'

Analyzing the count plot of 'Class' distribution by 'mode' reveals interesting patterns in how different music genres are represented across these two modes (likely major and minor):

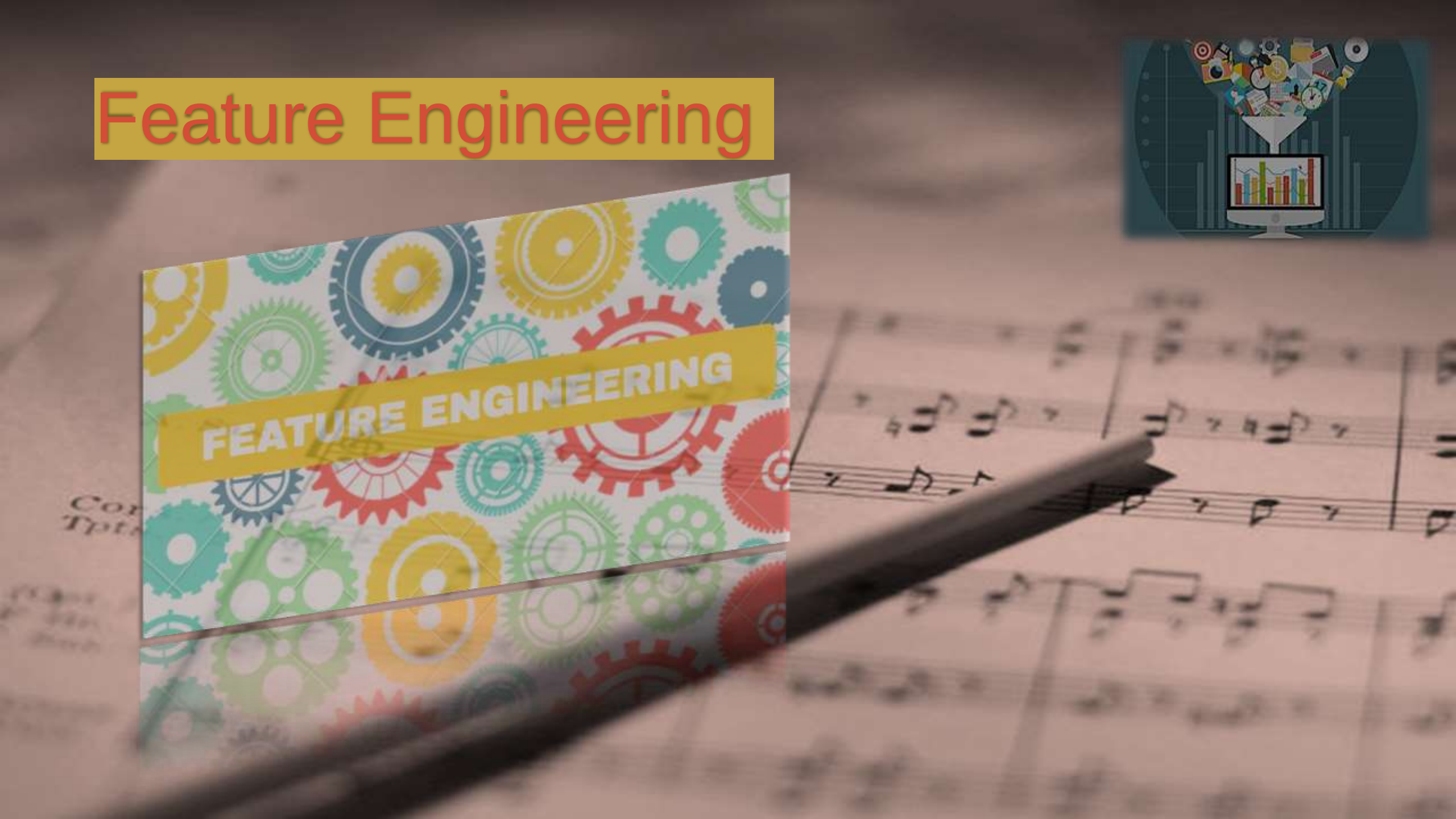
1. **Mode 0 (Minor):**

- Dominance of Certain Genres: Classes 10, 9, and 8 are the most prevalent in this mode, suggesting these genres might favor a minor tonality.
- Underrepresentation of Others: Classes 3, 4, and 7 appear less frequently, indicating a potential dislike for minor keys within these genres.

2. **Mode 1 (Major):**

- Shift in Genre Distribution: While Class 10 remains dominant, Class 6, which was prominent in Mode 0, becomes significantly less frequent. This suggests a strong preference for major keys in Class 6.
- Increased Representation: Genres like Classes 1, 2, and 4 show a notable increase in representation compared to Mode 0, indicating a potential affinity for major keys.

Feature Engineering



Feature Engineering

1. Dropping Irrelevant Columns (“Artist Name”, “Track Name”)

2. Checking for Duplicate Rows after dropping irrelevant columns

- We have 24 Duplicate Rows so:

3. Dropping Duplicate Rows

4. Getting the number of instances and features

```
(14372, 15)
```

Feature Engineering

2. Handling Outliers

```
# Replace outliers with upper/lower bounds using IQR
```

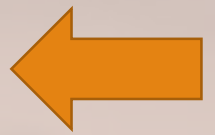
- Replace Outliers with upper/lower outlier thresholds using IQR
- Checking for Duplicate Rows after handling outliers
- We have 2 Duplicate Rows so we dropping them.

Feature Engineering

3. List of categorical features with low cardinality (few unique values)

```python
# List of categorical features with high cardinality (many unique values)
cat_cols_high_cardinality = []  #  'Artist Name', 'Track Name'
```

```python
# List of categorical features with low cardinality (few unique values)
cat_cols_low_cardinality = ['key', 'mode', 'time_signature']  # 'mode', 'key' & 'time_signature' are not high-cardinality feature
```



```python
from sklearn.base import TransformerMixin

class FrequencyEncodeTransformer(TransformerMixin):

    # Define frequency encoding function
    def frequency_encoding(self,X, column):
        # Calculate the frequency of each unique value in the column (using value_counts())
        freq = X[column].value_counts().to_dict()
        # Replace the original values with their corresponding frequencies (using Mapping)
        return X[column].map(freq)

    # Frequency encoding for high cardinality features
    def encode_high_cardinality(self,X):
        for col in X.columns:
            X[col] = self.frequency_encoding(X, col)
        return X

    # Fit method
    def fit(self, X, y=None):
        return self  # Returns the names of the features after transformation.
    # Transform method
    def transform(self, X):
            return self.encode_high_cardinality(X)
    # Get feature names
    def get_feature_names_out(self, input_features=None): # input_features: Optional list of input feature names.
        return input_features  # Returns: The original feature names as frequency encoding doesn't change them.
```

Feature Engineering – Pipeline – PowerTransformer()

4. Define List of numerical features , Define Numerical Pipeline and Handling Skewed Features

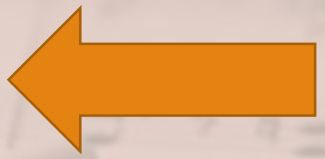
```
[60]  1 # List of numerical features with right skew distribution
      2 num_cols_right_skew = ['acousticness', 'speechiness', 'liveness', 'instrumentalness']
      3 # List of numerical features with lefft skew distribution
      4 num_cols_left_skew = ['loudness', 'energy', 'duration_in min/ms']
      5 # List of numerical features
      6 num_cols_P = ['Popularity', 'danceability', 'energy', 'loudness', 'speechiness', 'acousticness', 'liveness', 'valence', 'tempo', 'instrumentalness', 'duration_in min/ms']
```

```
[61]  1 # Define PowerTransformer for handling skewed features
      2 power_transformer = PowerTransformer(method='yeo-johnson', standardize=False)
      3
      4 # Define numerical pipeline
      5 num_pipeline = Pipeline(steps=[
      6     ('imputer', SimpleImputer(strategy='median')),  # Impute missing values with median
      7     ('scaler', RobustScaler(with_centering=False)),
      8     #('scaler', StandardScaler(with_mean=False))     # Standardize numerical features
      9 ])
     10
     11 # Combine transformation and numerical pipeline
     12 log_num_pipeline = Pipeline(steps=[
     13     ('power_transform', ColumnTransformer(
     14         transformers=[
     15             ('power', power_transformer, ['acousticness', 'speechiness', 'liveness', 'instrumentalness', 'loudness', 'energy', 'duration_in min/ms', 'Popularity'])
     16         ],
     17         remainder='passthrough'
     18     )),
     19     ('num_pipeline', num_pipeline)
     20 ])
```

Feature Engineering – Pipeline – ColumnTransformer()

5. Define Categorical Pipeline

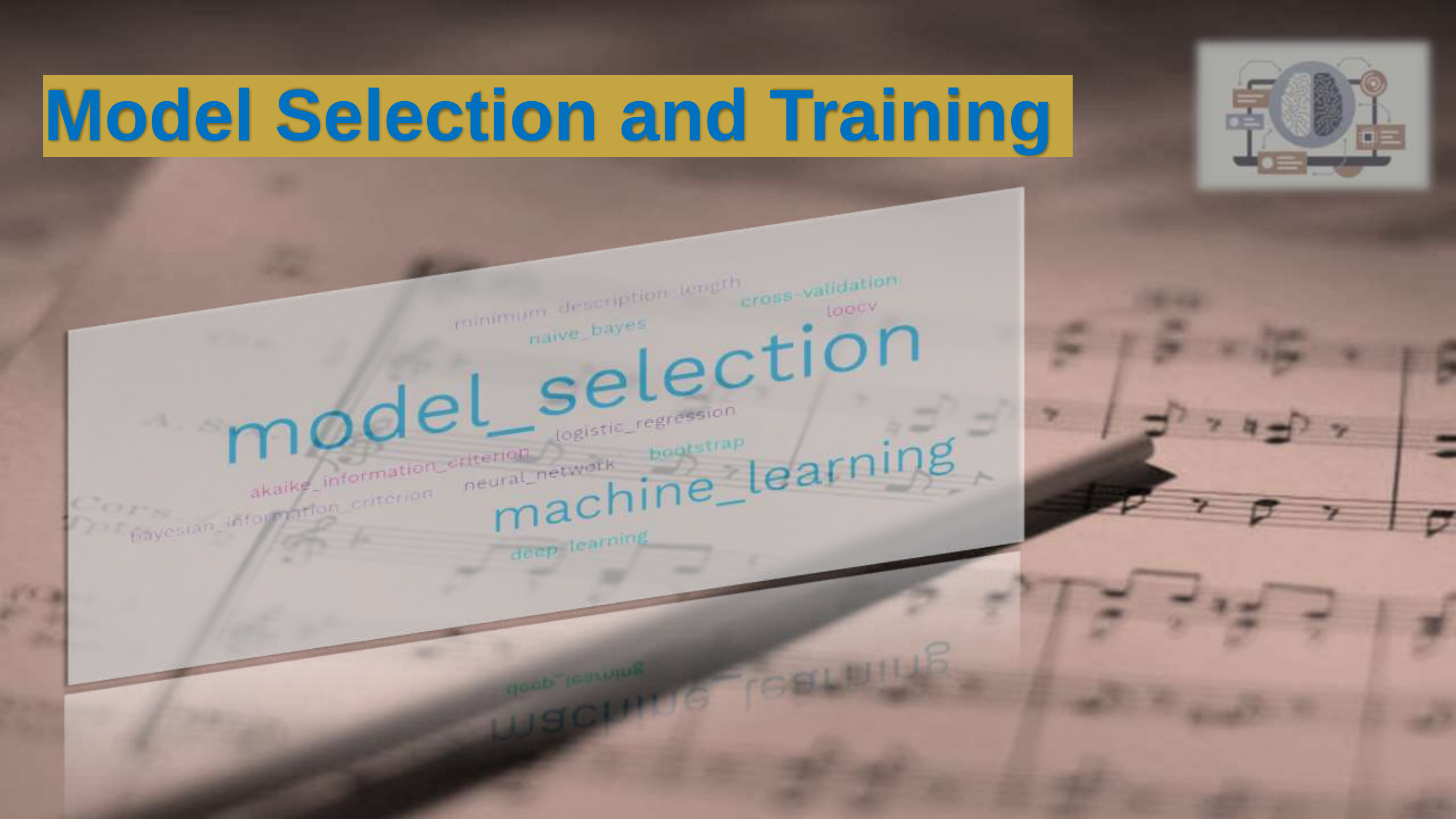
```
# Define categorical pipelines
# Pipeline - Frequency encoding for high-cardinality features
#freq_pipeline = Pipeline(steps=[
    #('encode', FrequencyEncodeTransformer()),
    #('scaler', RobustScaler(with_centering=False))
    #('scaler', StandardScaler(with_mean=False))
#])

# Pipeline - One-hot encoding for low-cardinality features
onehot_pipeline = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore')),
    ('scaler', RobustScaler(with_centering=False))
    #('scaler', StandardScaler(with_mean=False))
])
```



6. Combine pipelines into a ColumnTransformer

```
# Combine pipelines into a ColumnTransformer
full_pipeline = ColumnTransformer(
    transformers=[
        ('log_num', log_num_pipeline, num_cols_P),
        #('freq', freq_pipeline, cat_cols_high_cardinality),
        ('onehot', onehot_pipeline, cat_cols_low_cardinality)
    ]
)
```

Model Selection and Training



Model Selection and Training

1. Splitting the Data into Training and Testing Sets

2- Apply Processing Pipeline

3. Select & Train Model

3. Select & Train Model – Part 1

```python
# Select and Train a Model

fitted_models = {}  # Initialize a dictionary to store fitted models

def train_model(model,model_name):

  # Calculate class weights - calculating weights based on class frequencies
  class_weights = compute_class_weight('balanced', classes=np.unique(Y_train), y=Y_train)
  class_weight_dict = dict(enumerate(class_weights))

  # Train the model with selected features AND class weights IF SUPPORTED
  if hasattr(model, 'class_weight'):
    try: # Attempt to fit with class weights
      model.fit(X_train_prepared, Y_train, class_weight=class_weight_dict)
      print(f"{model_name} supports class weights and has been trained accordingly.")
    except TypeError as e:  # If TypeError (likely due to unsupported class_weight), train without it
      if "class_weight" in str(e): # Check if the error is specifically about class_weight
        model.fit(X_train_prepared, Y_train)
        print(f"{model_name} does not support class weights. Trained without them.")
      else:
        # If a different TypeError, raise it for further investigation
        raise e
  else:
    model.fit(X_train_prepared, Y_train)
```

- Class_Weight :

Our dataset exhibited a significant class imbalance, with certain music genres being much more represented than others. This imbalance can bias the model towards predicting the majority classes. To address this, we employed class weights during training. By assigning higher weights to the minority classes, we encouraged the model to pay more attention to learning patterns from these underrepresented genres, ultimately leading to a more accurate and fair classification across all music genres.

3. Select & Train Model – Part 2

```
    # Perform cross-validation
    cv_scores = cross_val_score(model, X_train_prepared, Y_train, cv=5, scoring='f1_weighted')
    # Make predictions
    predictions = model.predict(X_test_prepared)
    # Evaluate the model
    f1 = f1_score(Y_test,predictions, average='weighted')  # , average='weighted'
    recall = recall_score(Y_test,predictions, average='weighted')  # , average='weighted'
    accuracy = accuracy_score(Y_test,predictions)
    precision = precision_score(Y_test,predictions, average='weighted')  # , average='weighted'

    # Store the metrics in a dictionary
    metrics = {
        'Algorithm': model_name,
        'F1-Score': f1,
        'Recall': recall,
        'Accuracy': accuracy,
        'Precision': precision,
        'Cross-Val F1 Scores': cv_scores,
        'Avg Cross-Val F1': cv_scores.mean()
        }

    fitted_models[model_name] = model  # Store the fitted model
    return model, metrics

# Initialize a dictionary to store the results
results = {}
```

- Perform Cross-Validation:

"To ensure the robustness and generalizability of our model, we employed 5-fold cross-validation. This technique involves splitting the training data into five subsets, training the model on four subsets, and evaluating its performance on the remaining subset. This process is repeated five times, with each subset serving as the evaluation set once. By averaging the performance metrics across these folds, we obtained a more reliable estimate of the model's ability to generalize to unseen data, reducing the risk of overfitting."

4- Evaluating Model

```
{'Algorithm': 'XGBClassifier',
 'F1-Score': 0.4955485115918306,
 'Recall': 0.5086986778009742,
 'Accuracy': 0.5086986778009742,
 'Precision': 0.4911874672825021,
 'Cross-Val F1 Scores': array([0.48638804, 0.48922144, 0.48956536, 0.48984317, 0.48411124]),
 'Avg Cross-Val F1': 0.4878258490409113}
```

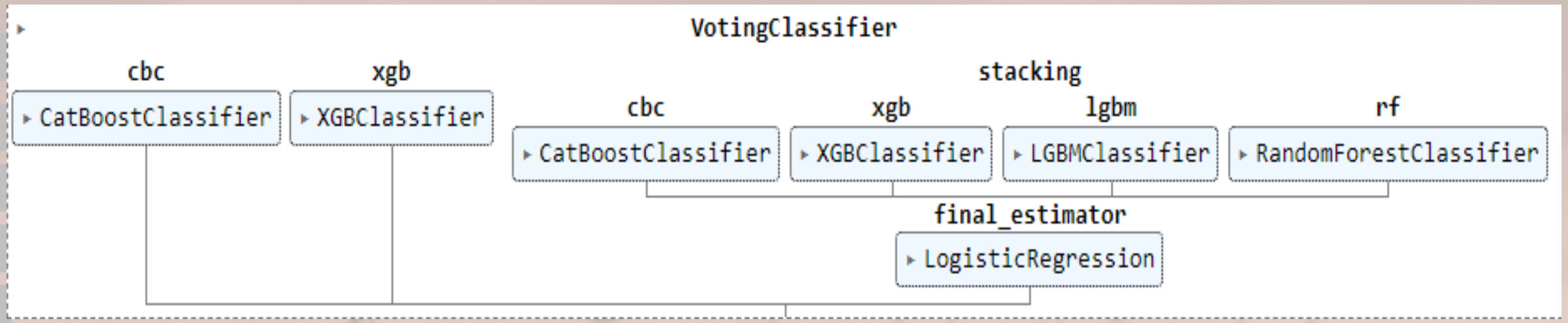
```
{'Algorithm': 'CatBoostClassifier',
 'F1-Score': 0.5144465102098724,
 'Recall': 0.5313152400835073,
 'Accuracy': 0.5313152400835073,
 'Precision': 0.5112942547316827,
 'Cross-Val F1 Scores': array([0.49173936, 0.50002951, 0.49848408, 0.50302227, 0.48787341]),
 'Avg Cross-Val F1': 0.4962297252525293}
```

- **Base Models for Stacking** : [CatBoostClassifier, XGBClassifier, LGBMClassifier, RandomForestClassifier]

- **Meta Model is** LogisticRegression

```
{'Algorithm': 'StackingClassifier',
 'F1-Score': 0.5529504917375768,
 'Recall': 0.5608907446068198,
 'Accuracy': 0.5608907446068198,
 'Precision': 0.5701347571022503,
 'Cross-Val F1 Scores': array([0.54105941, 0.53312999, 0.52763273, 0.54546098, 0.54409012]),
 'Avg Cross-Val F1': 0.538274644980641}
```

5- Ensemble method using VotingClassifier




```
Ensemble Voting Classifier F1 Score: 0.5154158896471976
```

6- Selecting the Best Model

```
Best Model: StackingClassifier
F1-Score: 0.5529504917375768
```

Evaluate Your System on the Test Set

1- Load the test dataset

2- Drop [“Id”, “Artist Name”, “Track Name”]

3- Apply full_pipeline on test dataset

4- Make predictions using the best model

5- Ensure test predictions is a 1D array by using flatten() method

Generating Submission File

1- Create a submission Data Frame

2- Save predictions to a CSV file

Thank You!

Team Names:

- Mays Al-Fasfous
- Sedra Merkhan
- Hesham Alsaadi
- Ahmad Sadik