COM SCI M148 Final Project Report

READ ME

FINAL PROJECT CODE: https://github.com/krackalackel02/Datafy

Data Set of Choice: Spotify

For this project, we analyzed the spotify-tracks-dataset provided by user maharshipandya. According to the Hugging Face platform, the data was collected and cleaned through Spotify's API and Python (not much further information on the method of collection was provided). The raw data set contains 114,000 entries. Each row corresponds to a song and its measured features. A link to the data can be found here.

The data set contains the following features:

- **track_id:** The Spotify ID for the track
- **artists**: The artists' names who performed the track. If there is more than one artist, they are separated by a ;
- **album_name:** The album name in which the track appears
- **track_name:** Name of the track
- **popularity:** The popularity of a track is a value between 0 and 100, with 100 being the most popular. The popularity is calculated by algorithm and is based, in the most part, on the total number of plays the track has had and how recent those plays are. Generally speaking, songs that are being played a lot now will have a higher popularity than songs that were played a lot in the past. Duplicate tracks (e.g. the same track from a single and an album) are rated independently. Artist and album popularity is derived mathematically from track popularity.
- **duration_ms:** The track length in milliseconds
- **explicit:** Whether or not the track has explicit lyrics (true = yes it does; false = no it does not OR unknown)
- **danceability:** Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable
- **energy:** Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale
- **key:** The key the track is in. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1
- **loudness:** The overall loudness of a track in decibels (dB)
- **mode:** Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0
- **speechiness:** Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values

between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks

- **acousticness:** A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic
- **instrumentalness:** Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content
- **liveness:** Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live
- **valence:** A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry)
- **tempo:** The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration
- **time_signature:** An estimated time signature. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure). The time signature ranges from 3 to 7 indicating time signatures of 3/4, to 7/4.
- **track_genre:** The genre in which the track belongs

Problem of Interest:

Given the data, our team was interested in addressing how the popularity of a song could be predicted by other features. To conduct this analysis, we performed exploratory data analysis to understand relationships between different variables. We then trained multiple models based on the concepts presented in class to determine which models would be able to address our problem.

Key Methodology that Addressed Problems and Why:

This report details our methodological approach and findings in developing predictive models for song popularity. Our analysis began with extensive exploratory data analysis (EDA), which revealed minimal direct correlations between individual features and popularity metrics, with maximum correlations of approximately 0.05. This early finding guided our subsequent modeling decisions, highlighting the need for sophisticated approaches capable of capturing complex feature interactions.

In response to these data characteristics, we implemented and evaluated multiple modeling approaches of increasing complexity. Our baseline K-Nearest Neighbors (KNN) implementation achieved 57.5% test accuracy, demonstrating the limitations of simple distance-based metrics in handling the high-dimensional nature of our dataset. Building upon these initial insights, we explored dimensionality reduction and clustering techniques through Principal Component Analysis (PCA). While these methods provided valuable perspectives on data variance and structure, they did not directly enhance our predictive capabilities due to the absence of clear popularity-related clustering patterns.

Our more advanced modeling efforts yielded significantly improved results. The Random Forest methodology demonstrated robust performance with 66% test accuracy and an AUC of 0.7085, effectively managing feature complexity through its ensemble approach of aggregated decision trees. However, the neural network architecture emerged as our most effective solution, achieving 77% test accuracy. This superior performance can be attributed to its sophisticated architecture, which incorporated ReLU activation functions in hidden layers and sigmoid activation in the output layer, enabling effective capture of non-linear relationships while maintaining strong generalization capabilities.

The comparative performance of these methodologies offers valuable insights into the nature of song popularity prediction. The neural network's success highlights the importance of modeling complex, non-linear feature interactions, while the Random Forest's performance demonstrates the value of ensemble methods in managing feature complexity. The limitations of simpler approaches underscore the inherent complexity of popularity prediction and the necessity for sophisticated modeling techniques. Through the strategic implementation of these varied approaches, complemented by rigorous preprocessing protocols, we successfully developed a robust framework for predicting song popularity. The clear superiority of the neural network methodology suggests that future efforts in this domain should prioritize architectures capable of modeling complex feature relationships while maintaining computational efficiency and generalization capabilities.

Cross-Validation/Evaluation Metrics Used:

Results

In this section, we present the outcomes of the various models used to predict song popularity. These results highlight each method’s strengths and weaknesses and have guided our selection of the most effective approach.

Cross-validation

- For the KNN model, 5-fold cross-validation was used to validate model performance and generalizability.
- For the random forest and neural network models, we split the data into training, validation, and test sets.

Neural network results

- Test accuracy: 78%
  The neural network achieved the highest accuracy among all models tested, and it was able to effectively capture the non-linear relationships between features and popularity. The high accuracy suggests that the model was proficient in distinguishing between popular and non-popular songs.
- Training and validation loss:

The training and validation loss curves both showed a steady decrease over epochs, eventually converging. This shows that the model was learning effectively without overfitting and the model generalizes well to unseen data.
The decreasing loss trends also suggest that the model's learning rate (0.01) was well-tuned, and the network learned at an optimal pace. Had the learning rate been too high, we would have observed erratic loss curves, while a lower learning rate would have slowed convergence significantly.

- Model limitations:
  Neural networks are often considered black-box models, so it is difficult to interpret the exact contributions of each feature to the prediction.

Random forest results

- Validation accuracy: 66.5%
- Test accuracy: 66.0%
  The random forest model provided a solid performance, with an accuracy of 66% on the test set. This performance is notably better than KNN but falls short of the neural network.
- Test AUC: 0.7085
  The AUC score suggests that the model had fair predictive performance- it was reasonably good at distinguishing between popular and non-popular songs. The model's predictions are better than random guessing but there is still room for improvement.
- Model strengths:
  The random forest model was effective in handling the data's complexity and reducing the impact of outliers. It was able to aggregate the results of multiple decision trees, providing robustness against overfitting.
- Model weaknesses:
  The random forest model's accuracy was limited, possibly due to weak correlations between individual features and popularity. Its inability to capture more intricate patterns in the data prevented it from achieving higher accuracy.

KNN results

- Test accuracy: 56.8%
  The K-Nearest Neighbors (KNN) model struggled to accurately classify songs based on popularity, achieving a test accuracy of just 56.8%. This result is only marginally better than random guessing. KNN was not well-suited for the problem.
- 5-fold cross-validation AUC:
  - Scores: [0.5775, 0.5813, 0.5791, 0.5822, 0.5815]
  - Average AUC: 0.5803
    The low AUC scores indicate the model's limited ability to distinguish between popular and non-popular songs. There is poor classification performance.
- Confusion matrix for KNN:
  [[3883 4145]
  [3606 6314]]
  True positive rate: 63.6%
  True negative rate: 48.4%

The confusion matrix shows that KNN had difficulty distinguishing between the two classes- there was a high number of false positives and false negatives. The model struggles to identify non-popular songs accurately since it had a relatively low true negative rate.

- Model limitations:
  The KNN model's poor performance can be attributed to the Curse of Dimensionality and its reliance on distance metrics. It does not perform well when the data is high-dimensional or when features have a low correlation with the target variable.

Conclusions

Our project aims to predict a song's popularity based on its features. We leveraged various machine learning models including neural networks, random forests, and k-nearest neighbors (KNN).

The neural network model outperformed its counterparts, achieving a test accuracy of 77%. It was well-suited for capturing the non-linear relationships inherent in the data, which simpler models like KNN and random forest could not fully exploit. ReLU activation in the hidden layer and the sigmoid activation in the output layer helped the network learn complex patterns and produce reliable probabilistic outputs.

With a test accuracy of 66% and an AUC of 0.7085, the random forest model demonstrated a moderate ability to predict song popularity. While it handled non-linearities better than KNN, its performance was still limited by the weak correlations between individual features and popularity. The model's robustness to overfitting made it a reliable, albeit less accurate, alternative to the neural network.

The KNN model struggled with the task and had a test accuracy of 56.8% and an AUC of 0.5803. Its poor performance can be attributed to the Curse of Dimensionality and its reliance on distance metrics, which are less effective for high-dimensional data with weakly correlated features.

Predicting song popularity is a complex task that requires models capable of capturing non-linear interactions and subtle patterns within the data. The neural network's superior performance underscores the importance of using flexible and adaptive models for such tasks.

How to Use Code: - Neaam

Steps to Run Code

1. Clone the github repository to run on your device locally.
2. Install dependencies.
3. Use the latest version of Python to avoid conflicts.
4. Run the code.

APPENDIX

Data Preprocessing and Cleaning

Our data preprocessing workflow implemented a comprehensive approach to ensure optimal data quality and compatibility with learning algorithms. The process consisted of several critical steps designed to address common data challenges while preserving valuable information.

Missing data handling required a nuanced approach. We identified and removed columns with minimal missingness, including 'artists', 'album_name', and 'track_name', as their contribution to the modeling process was determined to be negligible. For the remaining features, we employed strategic imputation methods—applying mean imputation for numeric variables to maintain their central tendencies, while categorical features received mode imputation to ensure consistent representation across the dataset.
Our duplicate management strategy operated on two distinct levels. The first level focused on removing exact duplicates where all column values matched, effectively eliminating redundant data points. The second level addressed track-specific duplicates by implementing a sophisticated aggregation process. For records sharing the same track_id, we combined genres using logical operations and calculated average values for numeric features such as popularity. This approach ensured the preservation of unique information while maintaining data integrity.

```
Num rows before removing duplicates: 114000
Num rows after removing exact duplicates: 113550
Num rows involved in ID duplicates (including original rows): 23809
Num rows after aggreagting inexact duplicates: 89741
```

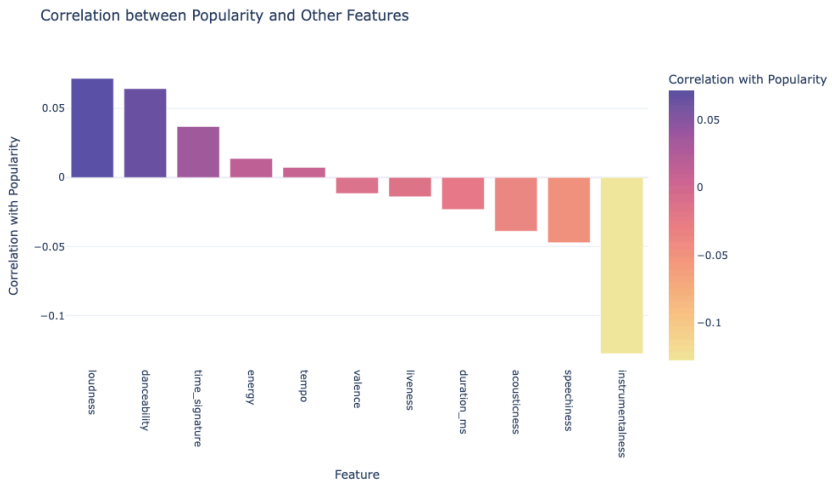
To enhance data quality further, we implemented outlier detection and removal using the Z-score method, establishing a threshold of three standard deviations. This statistical approach effectively identified and addressed extreme values that could potentially compromise the accuracy of our analyses and machine learning models.

The final preprocessing phase focused on feature encoding to optimize our data for machine learning applications. We transformed categorical features (key, mode, track_genre) into binary columns through one-hot encoding, maximizing their utility while preserving interpretability. Additionally, we standardized boolean columns into integer representations (True → 1, False → 0) to ensure consistency across all features.

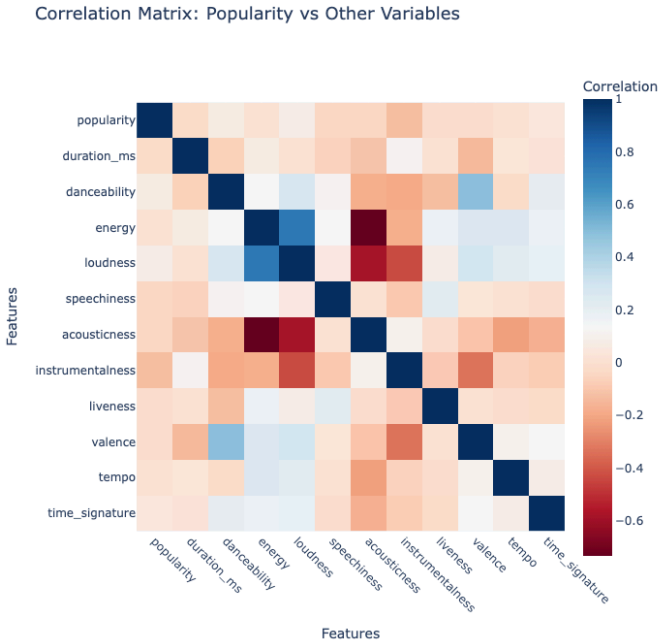
Exploratory Data Analysis

Our exploratory analysis revealed several key insights about relationships between musical features and their distributions within our dataset. Figure 1 presents our initial correlation analysis between track popularity and various audio features, showing notably weak correlations across all variables, with

maximum absolute correlation values of approximately 0.05. This suggests that predicting track popularity may be more complex than simply analyzing audio characteristics.

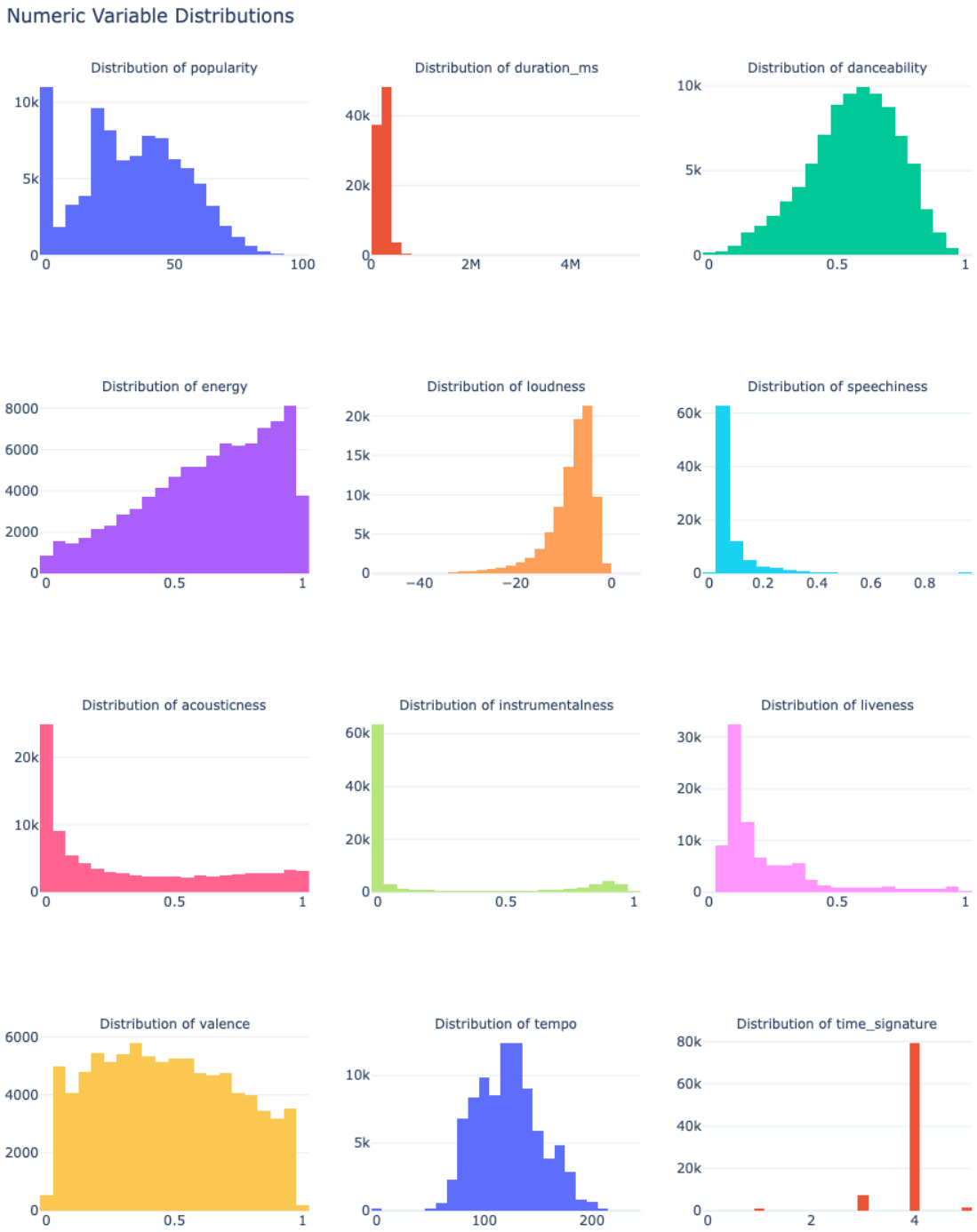


The comprehensive correlation matrix visualization in Figure 2 provides additional insights into feature relationships beyond popularity. We observed strong negative correlations between acousticness and electronic music characteristics (energy, loudness), as well as notable positive correlations between related features such as valence/danceability. These relationships suggest underlying patterns in how musical elements combine in tracks.



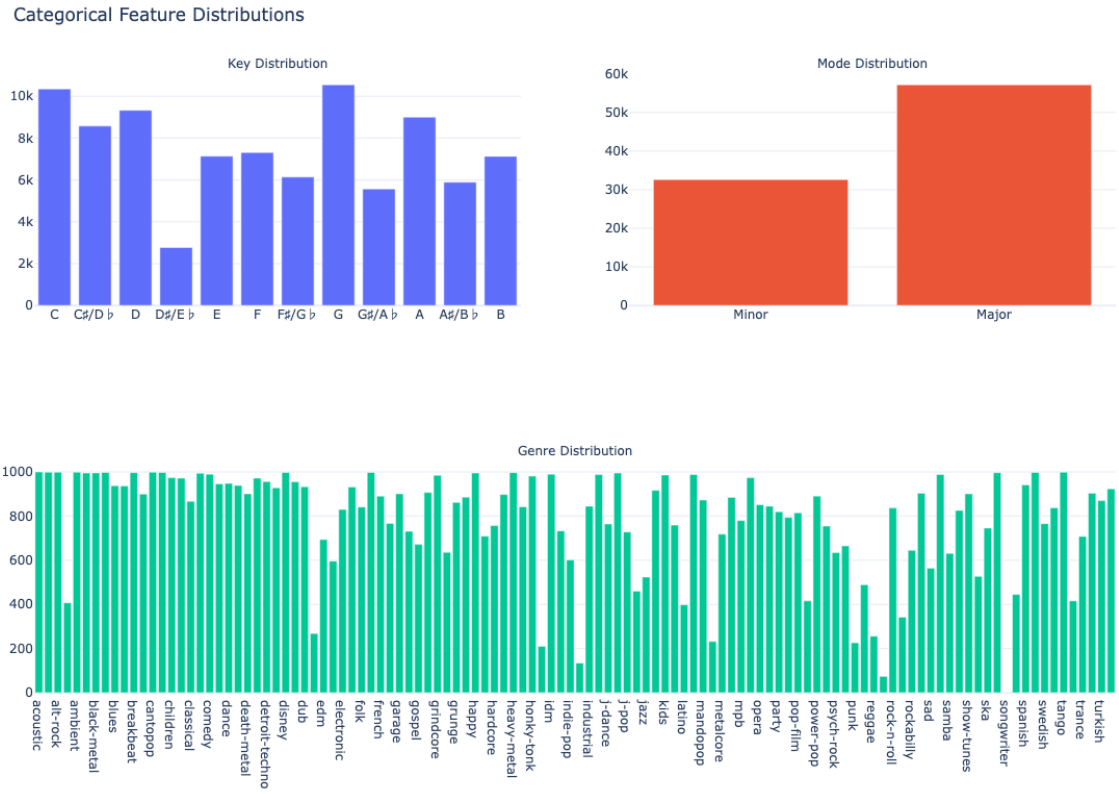
The analysis of numerical features, presented in Figure 3, revealed distinct distribution patterns across our variables. Several features exhibited clear normal distributions, including danceability, tempo, popularity, and loudness, suggesting natural clustering around central values for these characteristics. In contrast,

variables such as duration, speechiness, and instrumentalness showed heavily skewed distributions with significant outliers, indicating the presence of specialty tracks that deviate from typical patterns.



Our examination of categorical features in Figure 4 provided valuable context about the musical composition of our dataset. The mode distribution revealed a clear preference for major keys over minor

keys among tracks. The key distribution showed a relatively balanced representation across different musical keys, suggesting no strong bias toward particular tonalities. The genre distribution demonstrated remarkable diversity, with representation across numerous categories without overwhelming dominance by any single genre. This balance suggests our dataset provides a comprehensive view of the musical landscape.

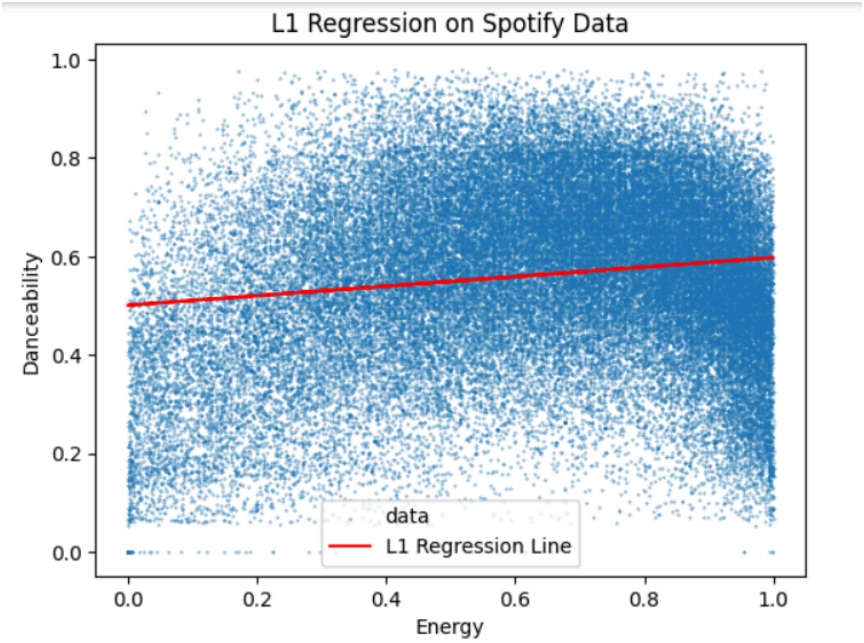


These analytical findings establish a strong foundation for subsequent modeling efforts, particularly highlighting the complexity of popularity prediction and the need to consider multiple feature interactions rather than individual characteristics in isolation. The balanced representation across musical categories and clear distribution patterns provide confidence in the dataset's ability to support robust analysis and modeling.

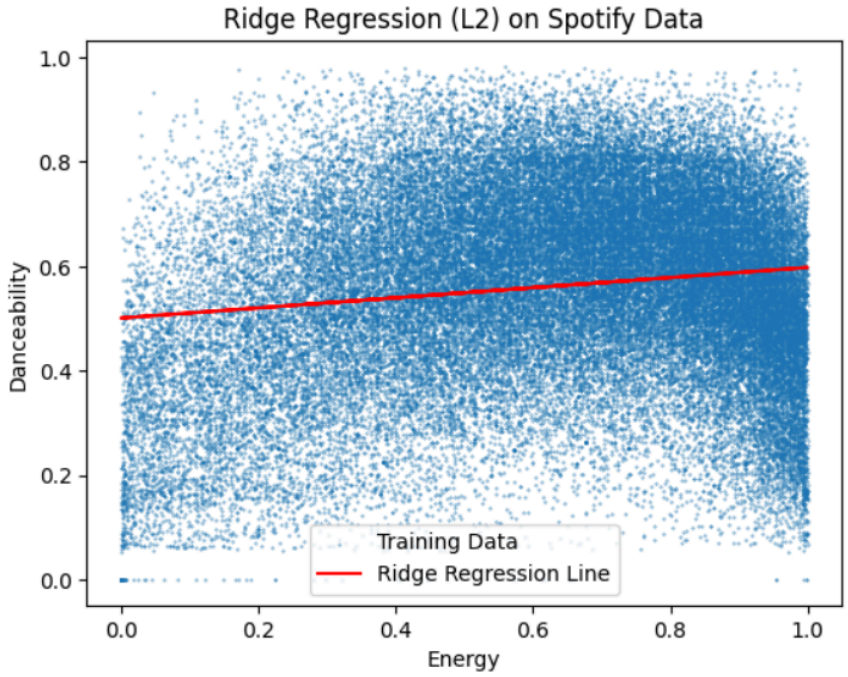
Regression Analysis

In the early stage of our project, we applied linear regression to model the relationship between energy and danceability. Our response and predictor variables were danceability and energy respectively. Our exploratory data analysis revealed that popularity has a weak correlation across different variables. So, we chose energy and danceability since they have a higher correlation than other variable pairs. Choosing popularity would most likely show a nonlinear relationship. We first trained an L1 Regression Model and an L2 Regression Model without using any regularization technique.

Our L1 model:



Our L2 model:



Both the L1 and L2 regression models with no regularization appear to be underfitting the data. The regression lines are almost flat. There seems to be a more complex, non-linear relationship between energy and danceability, and the regression lines fail to capture this. We didn't use this analysis to asssess feature importance.

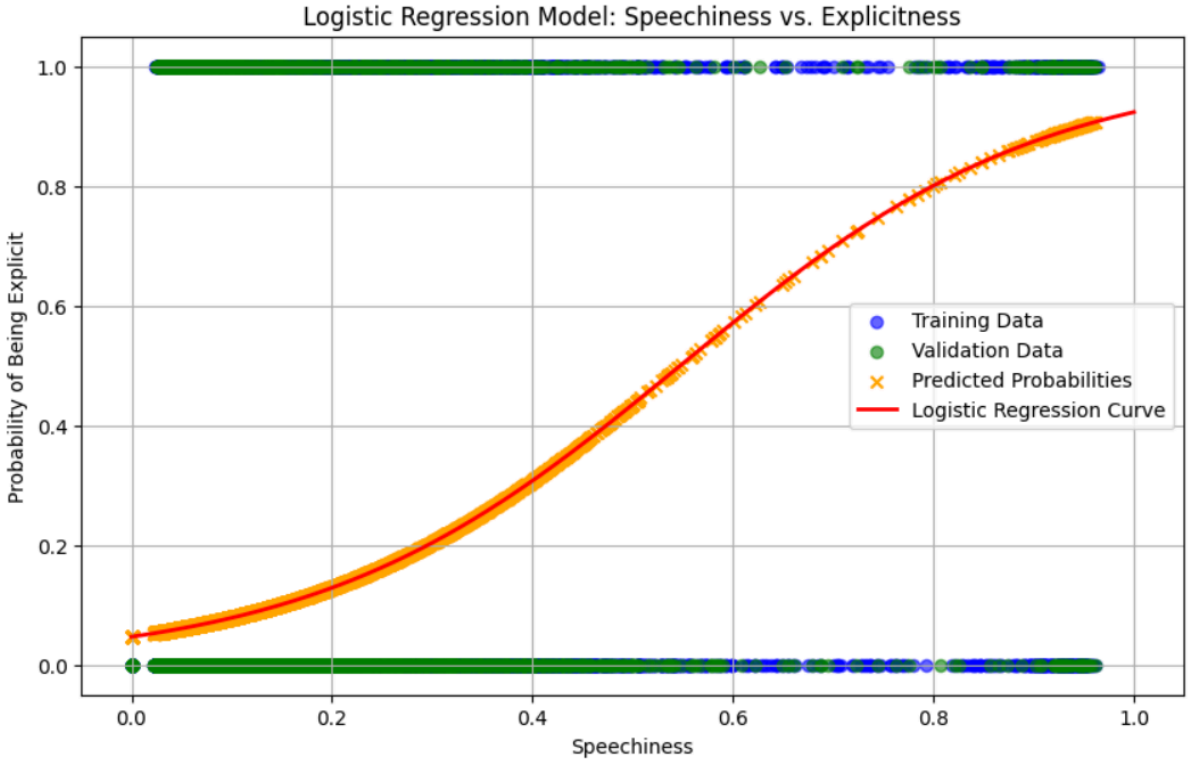
Since both L1 and L2 regression models appear to be underfitting the data and scatter plots suggest a non-linear relationship between energy and danceability, regularization is not necessary. Regularization is primarily a technique to prevent overfitting, not underfitting.

Logistic Regression Analysis

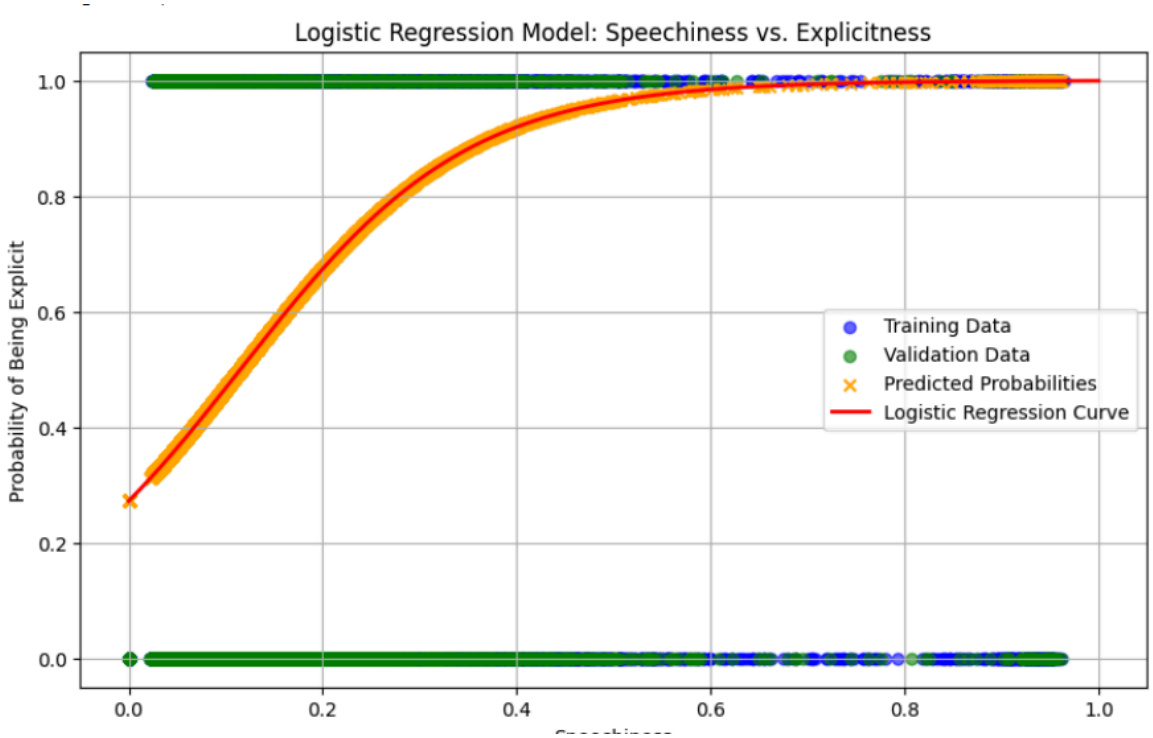
Logistic regression was used to predict whether a song is explicit based on the feature speechiness. The response variable, explicitness, was binary (1 for explicit, 0 for non-explicit).

The following steps were performed. We first split the dataset into training, validation, and test sets. Then, we used logistic regression with and without class weighting to account for class imbalance. We assessed performance through metrics such as accuracy, sensitivity, specificity, and the ROC curve. Our last step was optimizing the decision threshold to balance sensitivity and specificity.

Logistic regression model without class weighting:



Logistic regression model with class weighting:



We found out that the dataset contained significantly more non-explicit songs (91.4%) than explicit ones (8.6%). This imbalance affected model performance, making it biased toward predicting non-explicit songs.

The feature speechiness showed some ability to distinguish explicit from non-explicit songs, but it was not highly predictive on its own.

The initial sensitivity was low (~6%), indicating the model struggled to identify explicit songs when we didn't account for class imbalance. After using class_weight='balanced' and tuning the threshold, the sensitivity improved to ~68% with a reasonable specificity (~71%). In both models, we have an AUC score of 0.763 indicating that the models had fair predictive performance.

We did not use this analysis to assess feature importance.

Regularization was not applied in our logistic regression analysis. Since logistic regression was performed with only one predictor (speechiness), regularization wasn't critical. If more features were included, regularization could help prevent overfitting, improve generalization of the model and identify and reduce the influence of less important features.

KNN/Decision Trees/Random Forest

For the Spotify data, we wanted to classify whether a class was popular or not based on its features. We performed K-nearest neighbors and random forest as classification algorithms to determine whether songs could be accurately predicted to be popular or not. Songs were categorized as popular depending on a

popularity threshold value of 30. If a song had a popularity score of 30 or greater, then the song was considered popular (1). This number was preferred over the threshold of 50 because it produced a more even distribution between popular and non-popular songs.

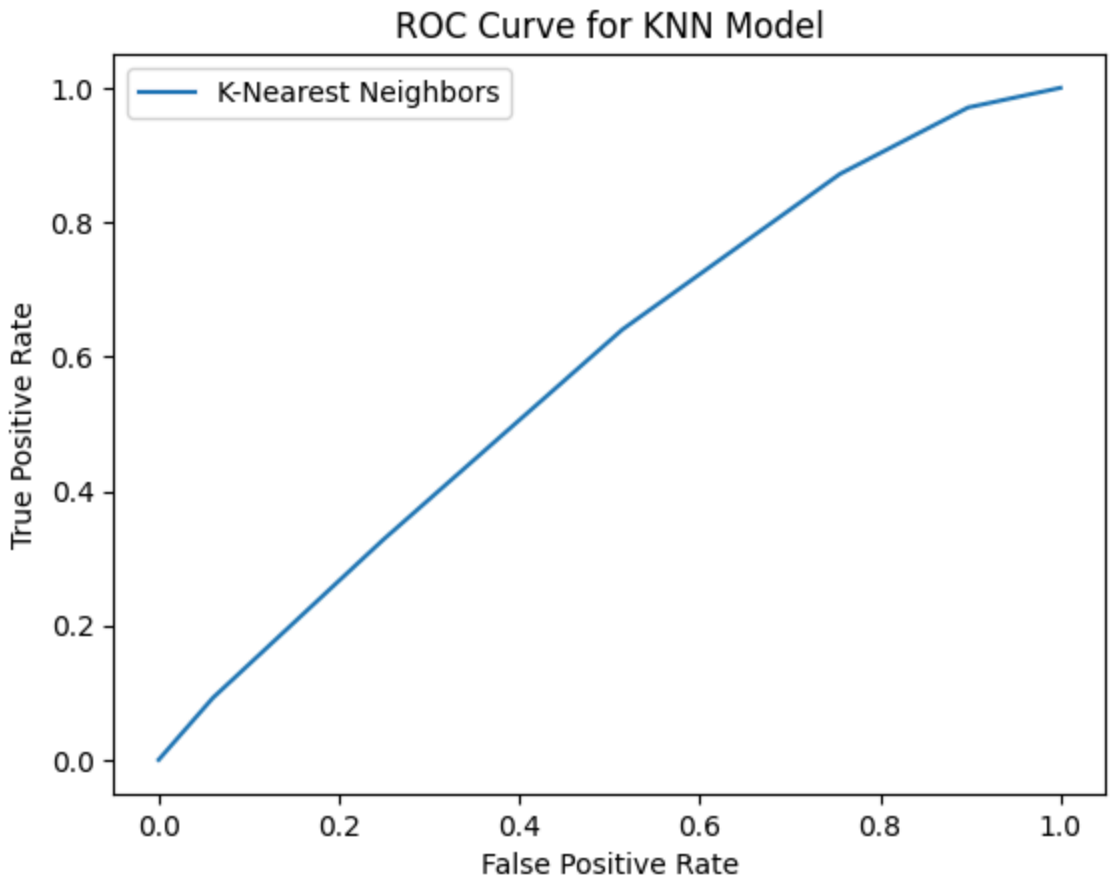
We wanted to determine whether popularity could be classified with KNN based on a point's danceability and speechiness. These features were chosen because based on domain knowledge, popular songs are often danced to. Also, danceability and speechiness had a low correlation when performing the correlation matrix so we wanted to study features that did not exhibit collinearity. The algorithm classifies a song as popular depending on its distance from danceability and speechiness.

To perform KNN, we tested whether popularity could be predicted based on its danceability and speechiness. We chose these metrics because they are relevant to popularity based on domain knowledge and because these two properties are not collinear themselves. We also chose not to include more than these two features to avoid the Curse of Dimensionality. However, the resulting test accuracy of 0.575 and average area under the curve (AUC) based on 5-fold cross-validation demonstrated that the model was not much better than random guessing.

Confusion Matrix:
[[3949 4180]
[3532 6288]]
Accuracy: 0.5703381804000223
Error: 0.42966181959997773
True Positive Rate: 0.640325865580448
True Negative Rate: 0.4857916102841678

AUC Score: 0.5850068172610746
5-Fold Cross-Validation AUC Scores: [0.57505427 0.57381161 0.57402936 0.57362206 0.57831236]
Average AUC Score: 0.5749659329743568

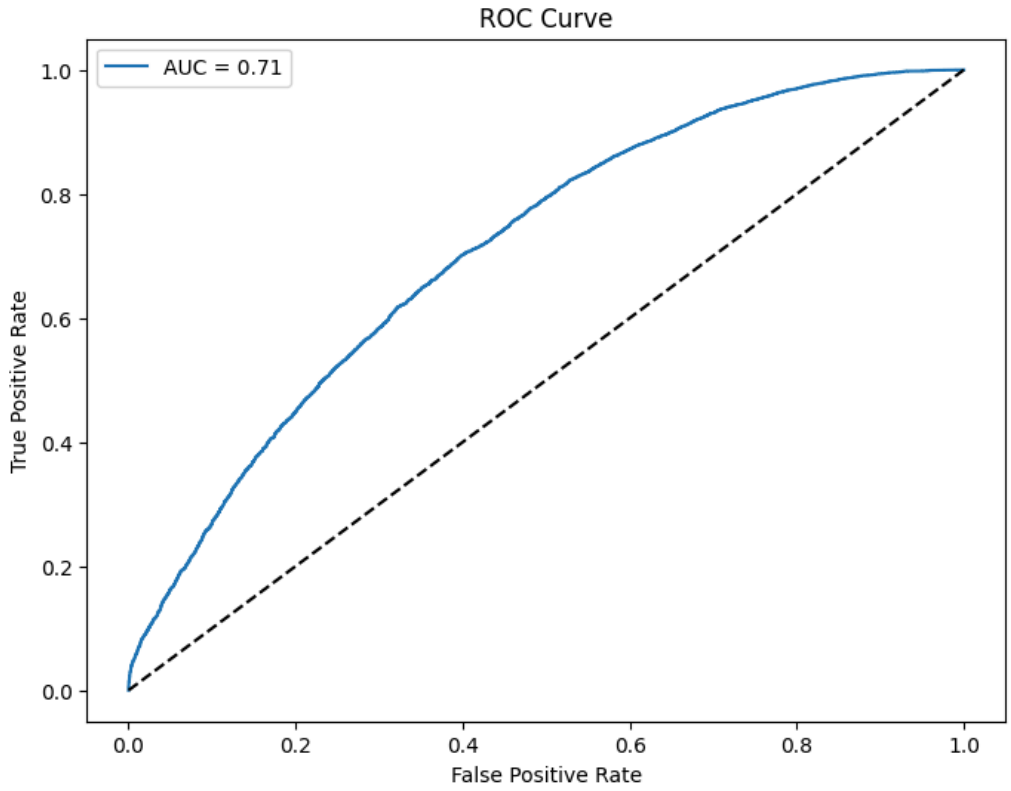
Figure 1: AUC curve of KNN



We then performed classification using the random forest algorithm. The random forest takes a majority vote from the classifications of the individual decision trees. The decision trees split the data based on the CART algorithm to classify songs as popular or not. The resulting test area under the curve for this model was 0.7085, demonstrating an improvement from KNN on classification of popularity.

Validation MSE: 0.3346891018497883
Test MSE: 0.3396478716291509
Validation Accuracy: 0.6653108981502117
Test Accuracy: 0.6603521283708491
Test AUC: 0.7085016344667884

Figure 2: AUC curve of random forest

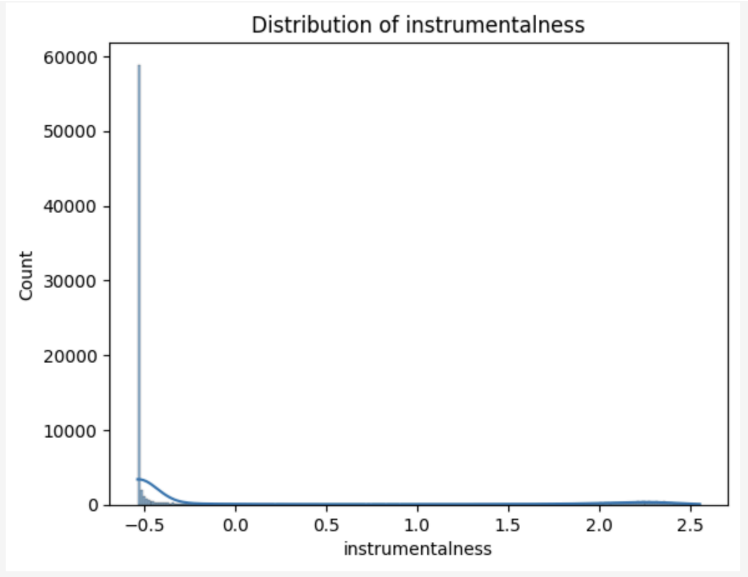


PCA & Clustering

Principal component analysis (PCA) is a dimensionality reduction technique which transforms correlated variables into uncorrelated variables to explain variance in the data. PCA and clustering did not directly address the problem of predicting a song's popularity based on its features. However, we were able to use PCA and clustering to see if there were certain songs that might have clustered together and if there was any relationship to its popularity. Based on our analysis, there were no clear clusters that depicted patterns in popularity.

For the Spotify dataset, we performed PCA on numerical variables including "speechiness", "danceability", "energy", "valence", "tempo", "loudness", and "acousticness." We excluded categorical variables such as "mode" and "time signature" because categorical variables do not have meaningful variance structure for PCA to analyze. We also excluded instrumentalness because it appeared to be very poorly distributed (Fig 1.). Prior to PCA, we standardized our variables because they were on different scales. To determine if a song is popular or not, we used a threshold of 50. If a song has a popularity score over 50, it is encoded as a 1 to indicate popularity. If it is below 50, then it is labeled with a 0, describing it as not popular.

Figure 1: Distribution of Instrumentalness



Once PCA was performed, we viewed the scree plot (Fig 2.) to determine which principal components to focus our attention on. We decided to choose the first two principal components because cumulatively, they describe 0.58 of the variance of the data. We then plotted PC2 against PC1 (Fig 3.).

Figure 2: Scree plot of PCA on standardized Spotify data

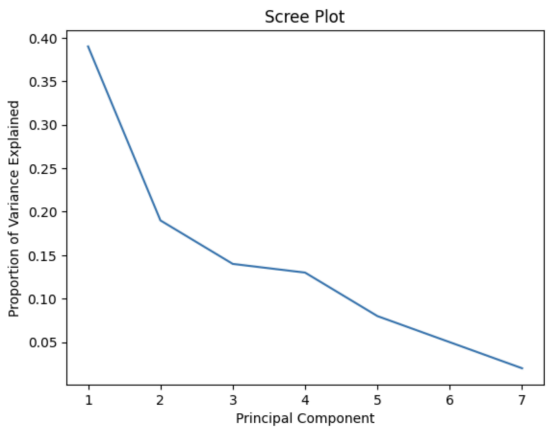
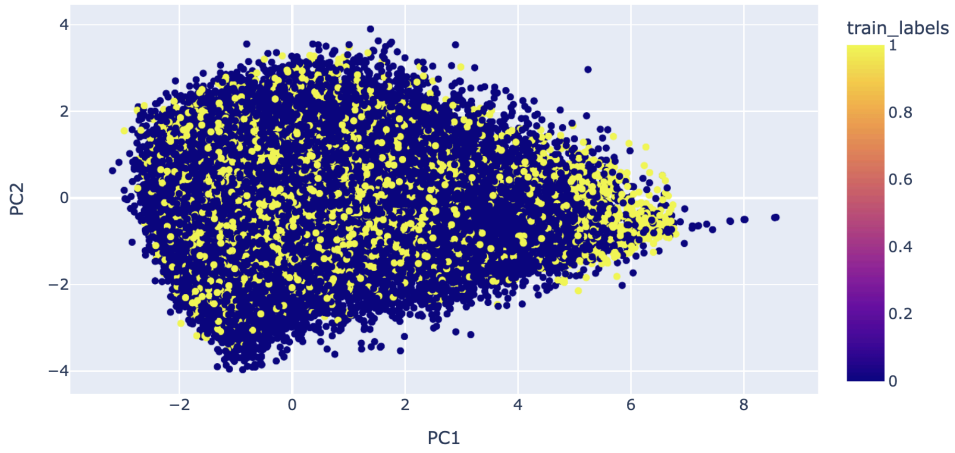


Figure 3: Plot of PC1 vs PC2 with popularity labels



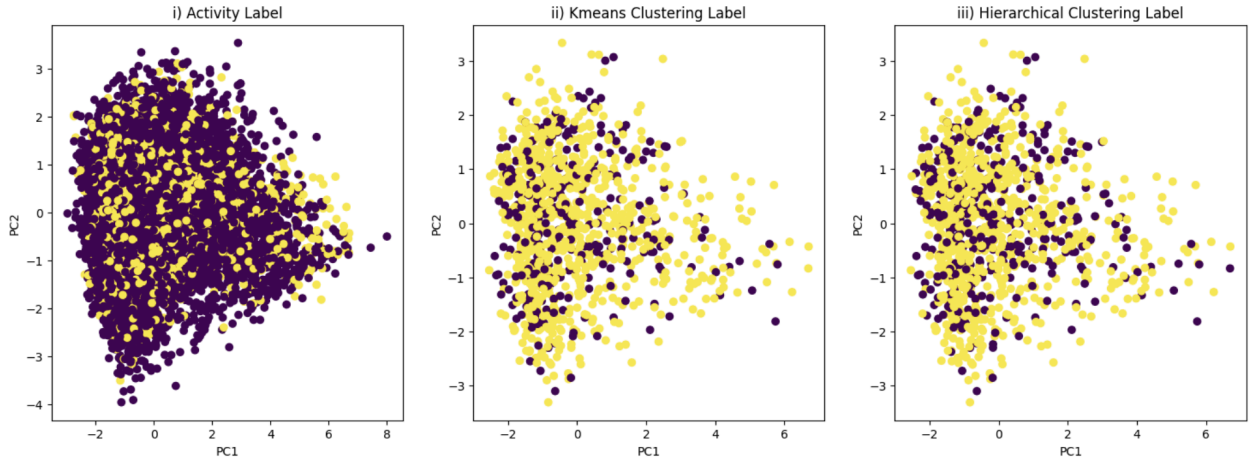
Because we standardized the data, we were able to gain meaningful insight into the variable loadings in both PC1 and PC2 to understand the data better. In PC1, energy contributed to the most variance with a loading of -0.539455 and in PC2, speechiness explained the most variance with a value of -0.958678. Both of these values were negative, suggesting that an increase in the aforementioned variables decreased the value of the PCs.

After performing PCA, we attempted both hierarchical and K-means clustering on the PCA transformed data. Due to the size of the data set, it was too computationally expensive to run clustering on the entire dataset.

To resolve this issue, we randomly sampled 10,000 data points from the original 89,740 point data set. This required an assumption that the random sample would be representative of the general trend in the data. Then, we performed hierarchical clustering and K-means clustering on this subsample to estimate how many clusters each method creates. When the number of clusters is not specified, hierarchical clustering creates two clusters while K-means clustering creates seven clusters.

Because we were looking at popularity, a binary variable, we chose to create two clusters. We performed both K-means clustering and hierarchical clustering on the subset of the data (n=10,000), generated silhouette scores for each method, and calculated the rand index to compare the two clustering methods. The silhouette-score for hierarchical clustering was 0.242 and the silhouette score for K-means was 0.294. The rand index was 0.784. This suggests that the clustering was not very descriptive of the patterns of the data across the clustering methods. However, the higher rand index indicates that both clustering methods were able to place the data points into similar clusters, suggesting that there may be another pattern we were not able to identify. Finally, we plotted the activity labels, K-means clustering labels, and hierarchical clustering to determine if there were any obvious clusters (Fig. 4). There were no obvious patterns in any of the clusters, showing that these methods were not particularly useful in addressing our overall goal.

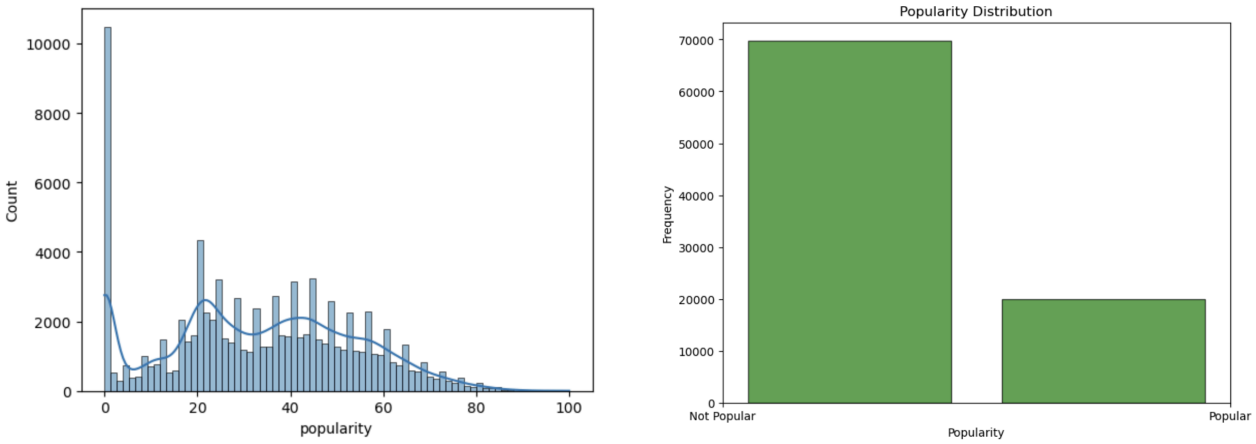
Figure 4: Subset of original data set plotted with their original labels (popular or not), kmeans labels, and hierarchical labels.



Neural Networks

In this section of our project, we trained our neural network. Training our neural network depends on backpropagation, which relies on the type of data/ layers, output, and activation function to generate the best model.

Our neural network model focused on converting popularity into a binary value, where 0 represents “not popular” and 1 represents “popular”. Since we had a range of popularity from 0-100, we set a threshold of 50 as we compared popularity distribution in comparison to its counts. These are represented through the following bar plots:



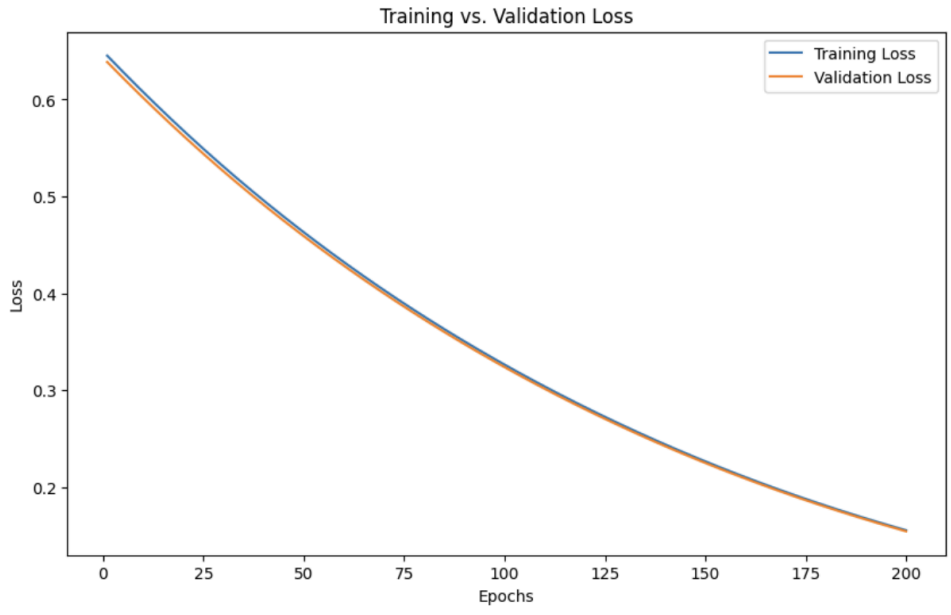
With this, we separated popularity into its own set of labels.

We split the data into its proper training and testing sets. Then, we standardized features to ensure that the training and testing were on the same scale since our feature variables were continuous prior to standardizing.

Using PyTorch, we implemented a neural network. Our model takes in continuous, standardized values as input where each feature is a dimension. Our hidden layer uses the ReLU activation function to introduce nonlinearity, allowing the network to learn patterns without it complicating the gradients. Since we want our output to be binary, using the sigmoid function was useful for the output layer in order to output values within the 0-1 range to be interpreted as probabilities. With this, we used a forward algorithm function with set hyperparameters to define our neural network. We initialized our model with our input & hidden layers and used an optimizer to set the learning rate. We also used Binary Cross Entropy Loss in order to track the loss function within our model.

Then, we trained the model through a ranging number of epochs. To train the model, we stored the losses in empty lists from both training and validation datasets at each epoch in order to address if the model is learning or overfitting. Storing a list helped us determine how the loss changes over time in order to further understand the model's learning behavior. After training the model, we set up an evaluation metric revealing the testing accuracy of our model.

For an improved model, we attempted to track the training loss in order to identify a decreased trend in the plot. This would determine that the model is learning to fit the data. The validation should follow a similar trajectory, decreasing alongside the training loss. We aimed to track if the model was overfitting, which would reveal if the validation loss plotline in our model would increase. If this were the case, we would have implemented regularization methods of neural networks that were discussed in lecture, such as dropout, norm penalties, and early stopping. Our results from training our model is revealed below:



Analyzing our plot, we can see that the model is not overfitting. The behavior of the training and validation loss is as expected. We can see the trends decreasing, ultimately reaching a convergence if we increase the number of epochs and allow the model to run for a longer period of time.

We set up a test accuracy function to determine the efficiency of our model. The model obtained a 78% test accuracy score, providing a proficient score to accept the model's results.

Since our model was not overfitting, it was not necessary to implement regularizations that could improve the trajectories of the model. However, to demonstrate our knowledge, we attempted to add norm penalties in the form of weight decays to the optimizer in order to see if results varied. Not much change occurred with this addition.

Hyperparameters

- The hyperparameters in our model are mainly influenced by learning rate, input layers, output layers, and the number of epochs.
- With a smaller learning rate, the trajectories of our training and validation loss plot were skewed. When we aimed to decrease the learning rate to 0.001, we found that the model's validation loss followed a linear trajectory, revealing that the learning rate was too low for the model. In our final neural network model, we set our learning rate to 0.01.
- We set our input layers to be the dimensions of the features, and set our hidden layers to be an arbitrary value, such as 16.
- We used the entire dataset to represent our batch size, relying on batch gradient descent in comparison to mini-batch or stochastic gradient descent.
- We also set our number of epochs to 200, as we could visually define convergence with this value.

1. Libraries

Install

In [ ]:
```
%pip install -q numpy
%pip install -q scipy
%pip install -q matplotlib
%pip install -q nbstripout
%pip install -q pandas
%pip install -q fsspec
%pip install -q huggingface_hub
%pip install -q scikit-lego
%pip install -q plotly
%pip install -q xlrd
%pip install -q -U kaleido
%pip install -q --upgrade nbformat
%pip install -q torch
```

```
[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python3.11 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

Import

In [ ]:
```
import numpy as np
import pandas as pd
import scipy as sp
import time
import random
from random import sample
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
```

```
import plotly.io as pio
import plotly.subplots as sbplt
import plotly.graph_objects as go
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.metrics import silhouette_score, silhouette_samples, rand_score, adjusted_rand_score, accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, cross_val_score, cross_validate
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.linear_model import  Ridge, Lasso
from itertools import product
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib as mpl
import matplotlib.cm as cm
import warnings
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, f1_score, roc_curve, roc_auc_score, c
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV


pio.renderers.default = 'notebook+pdf'
pd.set_option('display.width', 500)
pd.set_option('display.max_columns', 100)
warnings.filterwarnings('ignore')
```

2. Original Data

Import

In [ ]:
```
spotify_df = pd.read_csv("hf://datasets/maharshipandya/spotify-tracks-dataset/dataset.csv")
spotify_df.head()
```

Out[ ]:

	Unnamed: 0	track_id	artists	album_name	track_name	popularity	duration_ms	explicit	danceabi
0	0	5SuOikwiRyPMVoIQDJUgSV	Gen Hoshino	Comedy	Comedy	73	230666	False	0.6
1	1	4qPNDBW1i3p13qLCt0Ki3A	Ben Woodward	Ghost (Acoustic)	Ghost - Acoustic	55	149610	False	0.4
2	2	1iJBSr7s7jYXzM8EGcbK5b	Ingrid Michaelson;ZAYN	To Begin Again	To Begin Again	57	210826	False	0.4
3	3	6lfxq3CG4xtTiEg7opyCyx	Kina Grannis	Crazy Rich Asians (Original Motion Picture Sou...	Can't Help Falling In Love	71	201933	False	0.2
4	4	5vjLSffimiIP26QG5WcN2K	Chord Overstreet	Hold On	Hold On	82	198853	False	0.6

Data Characterstics

Drop extra ID col

In [ ]:
```
spotify_df = spotify_df.drop(columns=['Unnamed: 0'])
```

Shape

In [ ]:
```
# Print out the features
print("Features:", spotify_df.columns.tolist())

# Get the shape of the data
print("Shape of the data:", spotify_df.shape)
spotify_df.describe()
```

```
Features: ['track_id', 'artists', 'album_name', 'track_name', 'popularity', 'duration_ms', 'explicit', 'danceabili
ty', 'energy', 'key', 'loudness', 'mode', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', 'valenc
e', 'tempo', 'time_signature', 'track_genre']
Shape of the data: (114000, 20)
```

Out[ ]:

	popularity	duration_ms	danceability	energy	key	loudness	mode	speechine
count	114000.000000	1.140000e+05	114000.000000	114000.000000	114000.000000	114000.000000	114000.000000	114000.0000
mean	33.238535	2.280292e+05	0.566800	0.641383	5.309140	-8.258960	0.637553	0.0846
std	22.305078	1.072977e+05	0.173542	0.251529	3.559987	5.029337	0.480709	0.1057
min	0.000000	0.000000e+00	0.000000	0.000000	0.000000	-49.531000	0.000000	0.0000
25%	17.000000	1.740660e+05	0.456000	0.472000	2.000000	-10.013000	0.000000	0.0359
50%	35.000000	2.129060e+05	0.580000	0.685000	5.000000	-7.004000	1.000000	0.0489
75%	50.000000	2.615060e+05	0.695000	0.854000	8.000000	-5.003000	1.000000	0.0845
max	100.000000	5.237295e+06	0.985000	1.000000	11.000000	4.532000	1.000000	0.9650

Missingness

The `check_missingness` function provides a detailed analysis of missing values in the dataset and visualizes the distribution of missingness across variables. This step is essential for understanding data quality and guiding preprocessing strategies.

Practical Steps in Missingness Handling

1. **Identify Missing Values**:
   - Calculates the percentage of missing values for each column, enabling a clear view of the data quality.
   - Filters columns with non-zero missingness for focused analysis.
2. **Overall Statistics**:
   - Computes the total number of missing values and their percentage across the entire dataset, offering a summary of the dataset's completeness.
3. **Visualization**:
   - Generates a bar plot showing the percentage of missing values for columns with non-zero missingness.
   - The visualization helps prioritize which columns need imputation, removal, or further inspection.
4. **Binary Missingness Data**:
   - Creates a binary representation of missing values (1 for missing, 0 for present), useful for downstream analysis such as identifying patterns in missing data.
5. **Customizable Output**:
   - Dynamically adjusts the visualization size based on the number of columns, ensuring readability.
   - Handles empty DataFrames gracefully, providing clear feedback when no analysis is needed.

Outcome

By applying this function:

- Missing values are systematically analyzed and quantified, offering a comprehensive view of data quality.
- The visualizations assist in identifying variables with critical levels of missingness, helping to decide whether to impute, drop, or investigate further.
- The dataset is better prepared for preprocessing, ensuring minimal disruptions during analysis or modeling.

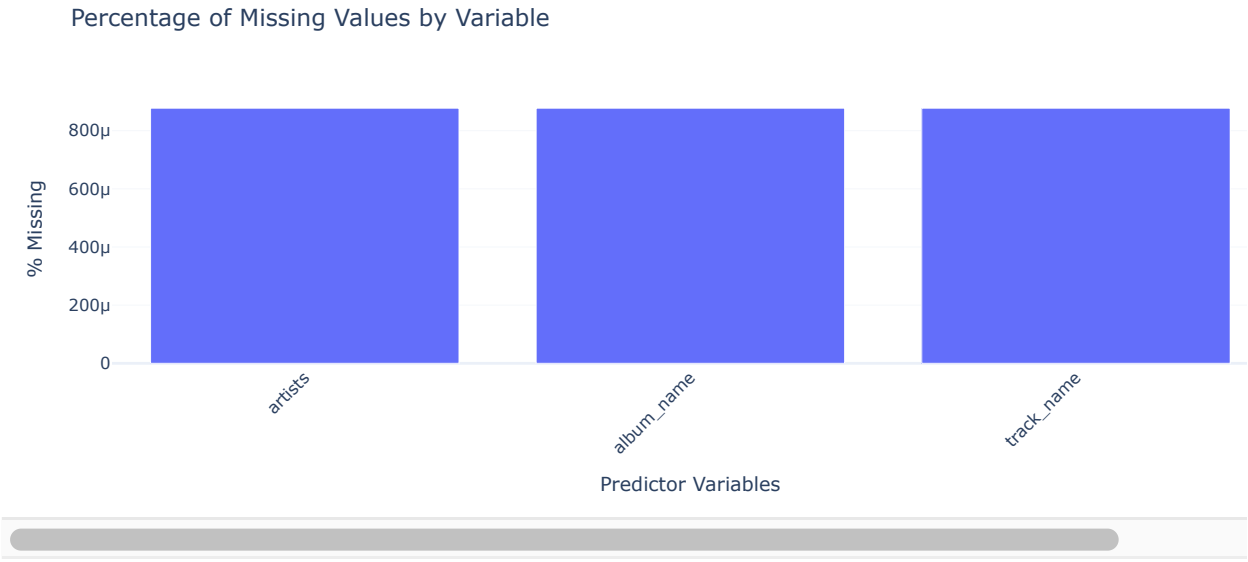
This function is a crucial step in maintaining dataset integrity, enabling informed decisions on how to handle missing data effectively.

In [ ]:

```
def check_missingness(df):
    """
    Analyzes and visualizes missing values in the DataFrame.

    Parameters:
    df (pandas.DataFrame): Input DataFrame.

    Returns:
    dict: Analysis results including missingness percentages and statistics.
    """
    if df.empty:
        print("The DataFrame is empty. No missingness to analyze.")
        return {
            'total_missing': 0,
            'total_missing_percent': 0,
            'missing_by_column': pd.Series(dtype=float),
            'missing_binary': pd.DataFrame(),
            'missing_analysis_df': pd.DataFrame(),
            'columns_with_missing': []
        }

    print("Analyzing missingness...")
```

```
    # Calculate missingness percentage
    missing_prop_percent = df.isna().mean() * 100

    # Create binary missingness DataFrame
    missing_binary = df.isna().astype(int)

    # Get total missingness
    total_missing = df.isna().sum().sum()
    total_missing_percent = (total_missing / (df.size)) * 100  # Use df.size for clarity

    print("\nPercentage of missing values per column (sorted in descending order):")
    non_zero_missing = missing_prop_percent[missing_prop_percent > 0]
    print(non_zero_missing.sort_values(ascending=False))

    # Create missing value DataFrame for visualization
    missing_df = pd.DataFrame({
        'Variable': missing_prop_percent.index,
        'Missing Percent': missing_prop_percent.values
    }).sort_values('Missing Percent', ascending=False)

    # Only show visualization if there are missing values
    if total_missing > 0:
        # Create visualization
        fig = px.bar(
            missing_df[missing_df['Missing Percent'] > 0],  # Filter to non-zero only
            x='Variable',
            y='Missing Percent',
            labels={'Variable': 'Predictor Variables',
                    'Missing Percent': '% Missing'},
            title='Percentage of Missing Values by Variable',
            template='plotly_white'
        )

        fig.update_layout(
            font=dict(size=12),
            xaxis_tickangle=-45,
            height=max(400, min(700, len(missing_df) * 20)),  # Dynamic height adjustment
            width=1000
        )

        fig.show()
    else:
        print("\nNo missing values found in the dataset.")

    return {
        'total_missing': total_missing,
        'total_missing_percent': total_missing_percent,
        'missing_by_column': missing_prop_percent,
        'missing_binary': missing_binary,
        'missing_analysis_df': missing_df,
        'columns_with_missing': missing_df[missing_df['Missing Percent'] > 0]['Variable'].tolist()
    }

check_missingness(spotify_df)
```

```
Analyzing missingness...

Percentage of missing values per column (sorted in descending order):
artists       0.000877
album_name    0.000877
track_name    0.000877
dtype: float64
```


Percentage of Missing Values by Variable
% Missing
800µ
600µ
400µ
200µ
0
artists
album_name
track_name
Predictor Variables

Out[ ]:
```
{'total_missing': np.int64(3),
 'total_missing_percent': np.float64(0.00013157894736842105),
 'missing_by_column': track_id            0.000000
 artists             0.000877
 album_name          0.000877
 track_name          0.000877
 popularity          0.000000
 duration_ms         0.000000
 explicit            0.000000
 danceability        0.000000
 energy              0.000000
 key                 0.000000
 loudness            0.000000
 mode                0.000000
 speechiness         0.000000
 acousticness        0.000000
 instrumentalness    0.000000
 liveness            0.000000
 valence             0.000000
 tempo               0.000000
 time_signature      0.000000
 track_genre         0.000000
 dtype: float64,
 'missing_binary':         track_id  artists  album_name  track_name  popularity  duration_ms  explicit  danceab
ility  energy  key  loudness  mode  speechiness  acousticness  instrumentalness  liveness  valence  tempo  time_
signature  track_genre
 0              0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 1              0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 2              0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 3              0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 4              0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 ...          ...      ...         ...         ...         ...          ...       ...          ...     ...  ...
...   ...          ...          ...               ...       ...      ...    ...            ...         ...
 113995         0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 113996         0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 113997         0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 113998         0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0
 113999         0        0           0           0           0            0         0            0       0    0
0     0            0            0                 0         0        0      0              0           0

 [114000 rows x 20 columns],
 'missing_analysis_df':             Variable  Missing Percent
 1            artists         0.000877
 2         album_name         0.000877
 3         track_name         0.000877
 0           track_id         0.000000
 4         popularity         0.000000
 5        duration_ms         0.000000
 6           explicit         0.000000
 7       danceability         0.000000
 8             energy         0.000000
 9                key         0.000000
 10          loudness         0.000000
 11              mode         0.000000
 12       speechiness         0.000000
 13      acousticness         0.000000
 14  instrumentalness         0.000000
 15          liveness         0.000000
 16           valence         0.000000
 17             tempo         0.000000
 18    time_signature         0.000000
 19       track_genre         0.000000,
 'columns_with_missing': ['artists', 'album_name', 'track_name']}
```

Duplicates

The `check_duplicates` function is designed to identify and analyze duplicate entries within a dataset, ensuring data integrity and consistency. Duplicates can arise from errors in data collection or redundant records, and addressing them is crucial for accurate analysis and modeling.

Practical Steps in Duplicate Handling

1. **Identify Exact Duplicates**:
    - Finds rows where all column values are identical. These are flagged as exact duplicates to be removed if necessary.
2. **Track-Specific Duplicates**:

- For the `track_id` column, identifies duplicate track entries where the same `track_id` appears multiple times. This ensures each track is represented uniquely in the dataset.

3. **Statistical Insights**:
   - Provides key metrics such as the minimum and maximum number of duplicates per `track_id` for a deeper understanding of the duplicate distribution.
4. **Visualization**:
   - Generates a bar plot (sampled if necessary) to visually represent the distribution of duplicate counts across `track_id`. This helps highlight patterns or anomalies in the dataset.
5. **Examples of Duplicates**:
   - Outputs a small sample of duplicate entries to facilitate manual inspection and verification.

Outcome

By leveraging this function:

- Duplicate entries can be systematically identified and visualized.
- Insights into duplicate patterns enable informed decisions about data cleaning and preprocessing.
- The dataset is prepared for further analysis, free from redundant or conflicting records.

This practical approach ensures the dataset's uniqueness and reliability, a foundational step for building robust models and drawing meaningful insights.

In [ ]:
```python
def check_duplicates(original, sample_size=100):
    """
    Analyzes and visualizes duplicate entries in the DataFrame

    Parameters:
    df (pandas.DataFrame): Input DataFrame
    sample_size (int): Number of samples to show in visualization

    Returns:
    tuple: (DataFrame without duplicates, duplicate analysis DataFrame)
    """
    print("\nAnalyzing duplicates...")
    df = original.copy()

    # Find exact duplicates
    duplicated_rows = df[df.duplicated(keep=False)]
    duplicated_rows_sorted = duplicated_rows.sort_values(by='track_id')

    print(f"\nFound {len(duplicated_rows)} exact duplicate rows")

    # Check for duplicate track IDs
    duplicates = df[df['track_id'].duplicated(keep=False)]
    print(f"\nFound {len(duplicates)} rows with duplicate track_ids")

    if len(duplicates) > 0:
        duplicate_counts = df.groupby('track_id').size().loc[lambda x: x > 1]

        # Create DataFrame for duplicates
        duplicate_df = duplicate_counts.reset_index()
        duplicate_df.columns = ['track_id', 'count']

        # Print statistics
        print("\nDuplicate statistics:")
        print(f"Min duplicates per track: {duplicate_df['count'].min()}")
        print(f"Max duplicates per track: {duplicate_df['count'].max()}")

        # Only show visualization if there are duplicates to show
        if len(duplicate_df) > 0:
            # Create visualization
            fig = px.bar(duplicate_df.sample(min(sample_size, len(duplicate_df))),
                         x='track_id',
                         y='count',
                         title='Duplicate Track Counts (Sample)')

            fig.update_layout(
                xaxis_title="Track Id",
                yaxis_title="Duplicate Counts",
                xaxis_tickangle=-45
            )

            fig.show()

        print("\nExample of duplicates (first 5 tracks):")
        print(duplicates.sort_values(by='track_id').head())

    else:
        print("\nNo duplicates found in dataset")
```

```
    return {
        'total_exact_duplicates': len(duplicated_rows),
        'total_track_id_duplicates': len(duplicates),
        'duplicate_analysis': duplicate_df if 'duplicate_df' in locals() else None,
        'example_duplicates': duplicates.sort_values(by='track_id') if len(duplicates) > 0 else None
    }
check_duplicates(spotify_df)
```

```
Analyzing duplicates...

Found 894 exact duplicate rows

Found 40900 rows with duplicate track_ids

Duplicate statistics:
Min duplicates per track: 2
Max duplicates per track: 9
```


```
Example of duplicates (first 5 tracks):
                    track_id           artists               album_name                                track_n
ame  popularity  duration_ms  explicit  danceability  energy  key  loudness  mode  speechiness  acousticness  inst
rumentalness  liveness  valence    tempo  time_signature track_genre
15028   001APMDOl3qtx1526T11n1  Pink Sweat$;Kirby                  New RnB                                     Bet
ter           0       176320     False         0.613   0.471    1    -6.644     0       0.1070       0.31600
0.000001    0.1170    0.406  143.064               4       chill
103211  001APMDOl3qtx1526T11n1  Pink Sweat$;Kirby                  New RnB                                     Bet
ter           0       176320     False         0.613   0.471    1    -6.644     0       0.1070       0.31600
0.000001    0.1170    0.406  143.064               4        soul
85578   001YQlnDSduXd5LgBd66gT        Soda Stereo  Soda Stereo (Remastered)  El Tiempo Es Dinero - Remasterizado 2
007          38       177266     False         0.554   0.921    2    -4.589     1       0.0758       0.01940
0.088100    0.3290    0.700  183.571               1   punk-rock
100420  001YQlnDSduXd5LgBd66gT        Soda Stereo  Soda Stereo (Remastered)  El Tiempo Es Dinero - Remasterizado 2
007          38       177266     False         0.554   0.921    2    -4.589     1       0.0758       0.01940
0.088100    0.3290    0.700  183.571               1         ska
91801   003vvx7Niy0yvhvHt4a68B        The Killers                 Hot Fuss                                Mr. Brights
ide          86       222973     False         0.352   0.911    1    -5.230     1       0.0747       0.00121
0.000000    0.0995    0.236  148.033               4        rock
```

```
Out[ ]: {'total_exact_duplicates': 894,
 'total_track_id_duplicates': 40900,
 'duplicate_analysis':                      track_id  count
 0      001APMDOl3qtx1526T11n1      2
 1      001YQlnDSduXd5LgBd66gT      2
 2      003vvx7Niy0yvhvHt4a68B      3
 3      004h8smbIoAkUNDJvVKwkG      2
 4      006rHBBNLJMpQs8fRC2GDe      3
 ...                       ...    ...
 16636  7ztSVy67w9rXpKg5L2zN5l      2
 16637  7zubR9uYAWjb5KPZTMm85e      4
 16638  7zumacGldlmxpoP8bpaeLe      2
 16639  7zv2vmZq8OjS54BxFzI2wM      2
 16640  7zwn1eykZtZ5LODrf7c0tS      3

 [16641 rows x 2 columns],
 'example_duplicates':                      track_id            artists                              a
lbum_name                            track_name  popularity  duration_ms  explicit  danceability  energy  ke
y  loudness  mode  speechiness  acousticness  instrumentalness  liveness  valence    tempo  time_signature  trac
k_genre
 15028   001APMDOl3qtx1526T11n1  Pink Sweat$;Kirby                                   New RnB
Better           0       176320     False         0.613   0.471    1    -6.644     0       0.1070      0.316000
0.000001    0.1170    0.406  143.064            4        chill
 103211  001APMDOl3qtx1526T11n1  Pink Sweat$;Kirby                                   New RnB
Better           0       176320     False         0.613   0.471    1    -6.644     0       0.1070      0.316000
0.000001    0.1170    0.406  143.064            4         soul
 85578   001YQlnDSduXd5LgBd66gT        Soda Stereo                      Soda Stereo (Remastered)  El Tiempo E
s Dinero - Remasterizado 2007         38       177266     False         0.554   0.921    2    -4.589     1
0.0758      0.019400          0.088100    0.3290    0.700  183.571              1    punk-rock
 100420  001YQlnDSduXd5LgBd66gT        Soda Stereo                      Soda Stereo (Remastered)  El Tiempo E
s Dinero - Remasterizado 2007         38       177266     False         0.554   0.921    2    -4.589     1
0.0758      0.019400          0.088100    0.3290    0.700  183.571              1         ska
 91801   003vvx7Niy0yvhvHt4a68B        The Killers                                  Hot Fuss
Mr. Brightside          86       222973     False         0.352   0.911    1    -5.230     1       0.0747
0.001210          0.000000    0.0995    0.236  148.033              4         rock
 ...                       ...                 ...                                 ...
...        ...         ...       ...          ...     ...  ...       ...   ...          ...          ...
...       ...      ...      ...          ...       ...
 72679   7zv2vmZq8OjS54BxFzI2wM             Attila                   Soundtrack to a Party (Bonus)
Lets Start the Party          25       125859      True         0.592   0.932    1    -5.412     1       0.0558
0.000005          0.859000    0.0730    0.677  133.987              4    metalcore
 22326   7zv2vmZq8OjS54BxFzI2wM             Attila                   Soundtrack to a Party (Bonus)
Lets Start the Party          25       125859      True         0.592   0.932    1    -5.412     1       0.0558
0.000005          0.859000    0.0730    0.677  133.987              4  death-metal
 2004    7zwn1eykZtZ5LODrf7c0tS  The Neighbourhood  Hard To Imagine The Neighbourhood Ever Changing
You Get Me So High          83       153000     False         0.551   0.881    7    -6.099     0       0.0542
0.186000          0.079100    0.1520    0.387   88.036              4     alt-rock
 3100    7zwn1eykZtZ5LODrf7c0tS  The Neighbourhood  Hard To Imagine The Neighbourhood Ever Changing
You Get Me So High          83       153000     False         0.551   0.881    7    -6.099     0       0.0542
0.186000          0.079100    0.1520    0.387   88.036              4  alternative
 91401   7zwn1eykZtZ5LODrf7c0tS  The Neighbourhood  Hard To Imagine The Neighbourhood Ever Changing
You Get Me So High          83       153000     False         0.551   0.881    7    -6.099     0       0.0542
0.186000          0.079100    0.1520    0.387   88.036              4         rock

 [40900 rows x 20 columns]}
```

3. Data Clean

Clean Function

Mean Imputation: Handling Missing Values

Theoretical Approach

To ensure the dataset is complete and ready for analysis, we need to handle missing values effectively. Missing data can lead to issues in both exploratory data analysis and modeling, such as biased estimates or algorithm errors.

Imputation Strategy:

- **Numeric Columns**: We use the mean to impute missing values in numeric columns. This approach minimizes bias for continuous variables and maintains the overall distribution of the data.
- **Categorical Columns**: We use the mode to impute missing values in categorical columns. This ensures that the most frequent category is preserved, reducing the risk of introducing outliers or inconsistent labels.

This strategy ensures the dataset remains representative of its original characteristics while minimizing potential disruptions caused by missing data.

Example Scenarios:

- **Numeric Feature Example**: Variables like `tempo` or `loudness`, which represent continuous measurements, are imputed with their mean to preserve their numerical consistency.
- **Categorical Feature Example**: Variables like `track_genre` or `key` are imputed with the mode to reflect the most common attribute within the data.

By applying these imputation techniques, we create a robust foundation for subsequent preprocessing and analysis steps.

Practical Implementation

While exploring the dataset, we observed that missing values were limited to labels such as `artists`, `album_name`, and `track_name`. These features are unlikely to directly impact the modeling process and had minimal missingness.

As a result:

- **Decision**: We dropped these columns (`artists`, `album_name`, and `track_name

In [ ]:
```python
def impute_missing_values(df):
    """
    Handles missing values in the DataFrame:
    - Drops specified columns with minimal missing values.
    - For object (string) columns: uses mode.
    - For numeric columns: uses mean.
    - For boolean columns: replaces missing values with False (or mode if preferred).

    Parameters:
    df (pandas.DataFrame): Input DataFrame

    Returns:
    pandas.DataFrame: DataFrame with missing values handled.
    """
    # Define columns to drop
    cols_to_drop = ['artists', 'album_name', 'track_name']

    # Drop specified columns
    df.drop(columns=cols_to_drop, errors='ignore',inplace=True)

    # Handle missing values for remaining columns
    for column in df.columns:
        if pd.api.types.is_object_dtype(df[column]):
            # Impute missing values in string columns with the mode
            if not df[column].mode().empty:  # Handle edge cases where mode might fail
                df[column].fillna(df[column].mode()[0], inplace=True)
        elif pd.api.types.is_bool_dtype(df[column]):
            # Impute missing values in boolean columns with False (or mode if preferred)
            df[column].fillna(False, inplace=True)
        elif pd.api.types.is_numeric_dtype(df[column]):
            # Impute missing values in numeric columns with the mean
            df[column].fillna(df[column].mean(), inplace=True)
```

Removing duplicates

Theoretical Approach

Duplicate records in a dataset can distort analysis, lead to biased results, and reduce the efficiency of algorithms by introducing redundant computations. To maintain data integrity and ensure the dataset accurately represents unique entities, we must systematically remove **duplicates**.

Duplicate Removal Strategy:

1. **Exact Duplicates**:
   - These occur when all columns in a row are identical. Removing these ensures that each record contributes uniquely to the analysis.
2. **Track-Specific Duplicates**:
   - For features like `track_id`, which uniquely identify songs, retaining only the first occurrence of each ensures that the dataset focuses on distinct tracks, even if associated metadata differs slightly across records.

By applying this strategy, we preserve the uniqueness and relevance of the dataset, making it more suitable for modeling and exploratory data analysis.

Practical Implementation

In our dataset, **duplicates** were addressed in two steps:

1. **Exact Duplicates**:
   - Removed rows where all columns were identical, ensuring there are no redundant entries.
2. **Track-Specific Duplicates**:
   - For the `track_id` column, only the first occurrence of each unique track was retained, as subsequent entries were likely repetitive or irrelevant.

Outcome: This approach ensures that our dataset contains only unique songs, with each track represented exactly once. This step is critical for downstream tasks like feature engineering and modeling, where **duplicates** could otherwise bias predictions or metrics.

This combination of theoretical considerations and practical execution guarantees a clean and reliable dataset for analysis.

In [ ]:
```python
def remove_duplicates(df):
    """
    Removes duplicates by:
    1. Removing exact duplicates.
    2. Aggregating features with 'genre_' in their name using a logical OR operation.
    3. Averaging the 'popularity' column for aggregated rows.
    4. Retaining only the first entry's genres for each track ID.

    Parameters:
    df (pandas.DataFrame): Input DataFrame with one-hot encoded genres.

    Returns:
    None (modifies DataFrame in-place).
    """
    # Step 1: Remove exact duplicates
    print("Num rows before removing duplicates:", len(df))
    df.drop_duplicates(inplace=True)
    print("Num rows after removing exact duplicates:", len(df))
    total_track_id_duplicates = len(df) - df['track_id'].nunique()
    print("Num rows involved in ID duplicates (including original rows):", total_track_id_duplicates)
    # Step 2: Identify boolean columns
    bool_cols = [
        col for col in df.columns
        if col.startswith(('genre_', 'key_', 'mode_')) or col == 'explicit'
    ]

    # Step 3: Group by track_id and aggregate
    grouped = df.groupby('track_id').agg(
        {**{col: 'max' for col in bool_cols},  # Logical OR (Union) for genres
         'popularity': 'mean',                    # Average popularity
         **{col: 'first' for col in df.columns if col not in bool_cols + ['popularity', 'track_id']}}  # Keep fir
    ).reset_index()

    # Step 4: Drop the original rows
    df.drop(df.index, inplace=True)

    # Step 5: Append the aggregated data
    for col in grouped.columns:
        df[col] = grouped[col]
    print("Num rows after aggreagting inexact duplicates:", len(df))
```

Removing Outliers

Theoretical Approach

Outliers in a dataset can significantly distort statistical analyses and model performance. These extreme values can arise from data entry errors, anomalies, or rare events and may bias results if not addressed appropriately.

Outlier Removal Strategy:

1. **Z-Score Method**:
   - The Z-score measures how far a data point is from the mean in terms of standard deviations. A threshold value (commonly 3) is used to identify extreme values.
   - Data points with Z-scores greater than the threshold are considered outliers.

This method is effective for numeric features with roughly normal distributions and ensures that the remaining dataset reflects typical values.

Practical Implementation

In our dataset, outliers were removed from all numeric columns using the Z-score method:

1. **Numeric Columns**:
   - Z-scores were calculated for each numeric feature to identify values that deviate significantly from the mean.
2. **Threshold**:
   - A threshold of 3 was used, meaning values more than 3 standard deviations away from the mean were considered outliers.
3. **Row Removal**:
   - Rows containing outliers in any numeric column were dropped to ensure data consistency.

Outcome: This process eliminated extreme values, reducing noise and improving the quality of the dataset. By removing outliers, we ensure that subsequent analyses and models are not skewed by anomalous data points.

This approach is crucial for datasets with features like `tempo` or `loudness`, where extreme values might misrepresent typical song characteristics.

In [ ]:
```python
def remove_outliers(df, threshold=3):
    """
    Removes outliers from numeric columns in-place using z-score method

    Parameters:
    df (pandas.DataFrame): Input DataFrame
    threshold (int): Z-score threshold for outlier detection

    Returns:
    None
    """
    # Get numeric columns
    numeric_columns = df.select_dtypes(include=[np.number]).columns

    # Calculate z-scores for numeric columns
    z_scores = sp.stats.zscore(df[numeric_columns])
    abs_z_scores = np.abs(z_scores)

    # Create mask for rows to keep
    keep_mask = (abs_z_scores < threshold).all(axis=1)

    # Drop rows with outliers in-place
    drop_track_idx = df.index[~keep_mask]
    df.drop(index=drop_track_idx, inplace=True)
```

One-Hot Encoding

Theoretical Approach

Categorical variables in datasets often need to be converted into a numerical format for use in machine learning models. One-hot encoding is a common technique that transforms categorical variables into binary features, preserving the categorical information while ensuring compatibility with numerical algorithms.

One-Hot Encoding Strategy:

1. **Binary Features**:
   - Each unique category in a column is represented as a new binary feature (column). A value of 1 indicates the presence of that category, while 0 indicates its absence.
2. **Custom Labels**:
   - For specific features like `key` or `mode`, meaningful labels are assigned to ensure interpretability.

This approach allows models to learn from categorical data without imposing unintended ordinal relationships between categories.

Practical Implementation

In our dataset, one-hot encoding was applied to key categorical variables:

1. **Key Encoding**:
   - The `key` column, representing musical keys, was mapped to descriptive labels (e.g., `C`, `D♯/E♭`).
   - Negative values such as `-1` were treated as `0` to represent "No Key" for consistency.
2. **Mode Encoding**:
   - The `mode` column, representing whether a track is in a major or minor key, was encoded into two binary features: `Minor` and `Major`.
3. **Track Genre Encoding**:
   - The `track_genre` column was encoded dynamically to reflect all unique genres in the dataset, ensuring a scalable and adaptable approach.

Outcome: By applying one-hot encoding, categorical data was transformed into a numerical format while preserving interpretability. This step ensures compatibility with machine learning algorithms and provides clarity in feature importance analysis.

Benefits of One-Hot Encoding:

- Avoids introducing unintended ordinal relationships between categories.
- Facilitates clear model interpretation, especially with labeled binary features.
- Ensures numerical compatibility across all features for preprocessing and modeling.

This implementation creates a dataset that retains the original information from categorical variables while being fully ready for analysis and modeling tasks.

In [ ]:
```python
def one_hot_encode(df, columns_with_labels):
    """
    One-hot encodes columns into binary features with descriptive labels.

    Parameters:
    df (pandas.DataFrame): Input DataFrame
    columns_with_labels (dict): Dictionary where keys are column names and values are
                                dictionaries mapping values to their labels
```

```
    Returns:
    pandas.DataFrame: DataFrame with new binary columns
    """
    columns_to_drop = []
    for col, value_labels in columns_with_labels.items():
        columns_to_drop.append(col)
        # For key column, replace -1 with 0 first
        if col == 'key':
            df[col] = df[col].replace(-1, 0)

        # Create binary columns for each value's label
        for value, label in value_labels.items():
            if col == 'key' and value == -1:
                continue  # Skip 'No Key' since we're treating it as 0/C

            new_col = f"{col}_{label}"
            df[new_col] = (df[col] == value)  # Removed .astype(int)

    columns_to_drop = list(set(columns_to_drop))
    df.drop(columns=columns_to_drop, inplace=True)

def encode(df):
    track_genre_values = df['track_genre'].unique()
    track_genre_map = {track_genre: track_genre for track_genre in track_genre_values}
    # Define labels for classification columns
    columns_with_labels = {
        'key': {
            0: 'C',
            1: 'C♯/D♭',
            2: 'D',
            3: 'D♯/E♭',
            4: 'E',
            5: 'F',
            6: 'F♯/G♭',
            7: 'G',
            8: 'G♯/A♭',
            9: 'A',
            10: 'A♯/B♭',
            11: 'B'
        },
        'mode': {
            0: 'Minor',
            1: 'Major'
        },
        # 'time_signature': {
        #     3: 'ts_3/4',
        #     4: 'ts_4/4',
        #     5: 'ts_5/4',
        #     6: 'ts_6/4',
        #     7: 'ts_7/4',

        # },
        'track_genre':
            track_genre_map

    }

    one_hot_encode(df, columns_with_labels)
```

Bool Conversion

Theoretical Approach

Boolean data types (True/False) often represent binary states or conditions in datasets. While boolean representations are intuitive for human understanding, numerical representations (e.g., 1/0) are preferred by many machine learning algorithms, including logistic regression, which inherently models probabilities and requires numerical input.

Boolean Conversion Strategy:

1. **Boolean to Integer (1/0)**:
   - Logistic regression and other numerical algorithms require binary features in numerical format.
   - Example: True → 1, False → 0.
2. **Integer (1/0) to Boolean**:
   - In certain stages, such as post-processing or interpretability analysis, numerical outputs may need to be converted back to boolean values for better human understanding.
   - Example: 1 → True, 0 → False.

This strategy ensures compatibility with machine learning algorithms while retaining flexibility for interpretability when required.

Practical Implementation

In our dataset, boolean conversion was implemented to address specific requirements:

1. **Boolean to Integer for Logistic Regression**:
   - Features such as `is_explicit` or `is_popular` were converted to numerical format to enable seamless integration with logistic regression models, which require numerical input for binary classification tasks.
2. **Integer to Boolean for Interpretability**:
   - After model predictions or feature engineering processes, numerical outputs were converted back to boolean format for clearer human interpretation.

Outcome: This process facilitates smooth preprocessing and modeling by converting boolean features to numerical format when needed, ensuring compatibility with algorithms like logistic regression while maintaining the ability to revert for interpretability.

Benefits of Boolean Conversion:

- Ensures compatibility with logistic regression and other machine learning algorithms requiring numerical input.
- Preserves interpretability by allowing conversions back to boolean format when needed.
- Provides flexibility to adapt boolean features for various stages of the data processing pipeline.

By dynamically converting boolean columns to integers and vice versa, this function ensures the dataset is ready for both machine learning tasks and human interpretation, bridging the gap between raw data and analytical requirements.

In [ ]:
```python
def convert_to_bool(df, conv_bool_to_int):
    """
    Converts all boolean columns in DataFrame to integers (1/0) or vice versa

    Parameters:
    df (pandas.DataFrame): Input DataFrame
    conv_bool_to_int (bool): If True, converts bool to int. If False, converts int to bool
    """
    bool_columns = df.select_dtypes(include=bool).columns

    if conv_bool_to_int:
        for col in bool_columns:
            df[col] = df[col].astype(int)
    else:
        for col in bool_columns:
            df[col] = df[col].astype(bool)

    return df
```

Final Function: clean_data()

The `clean_data` function serves as a comprehensive pipeline that integrates all essential cleaning methods to prepare the dataset for analysis and modeling. This modular design allows for flexibility in enabling or disabling specific steps based on the dataset's requirements.

Steps in the Cleaning Pipeline

1. **Duplicate Removal**:
   - Calls the `remove_duplicates` function to eliminate exact and track-specific duplicates, ensuring that each record is unique.
   - Uses `check_duplicates` to verify the cleaning process.
2. **Outlier Removal (Optional)**:
   - If the `rmOutliers` flag is set to `True`, applies the `remove_outliers` function to eliminate extreme values from numeric columns.
3. **Imputation**:
   - Calls the `impute_missing_values` function to handle missing values by imputing with the mean for numeric columns and the mode for categorical columns.
4. **Missingness Check**:
   - Uses `check_missingness` to analyze and visualize the extent of missing data, ensuring that all missing values have been addressed appropriately.
5. **One-Hot Encoding (Optional)**:
   - If the `oneHot` flag is set to `True`, applies the `encode` function to transform categorical features into binary columns for compatibility with machine learning models.
6. **Boolean Conversion**:
   - Calls the `convert_to_bool` function to convert boolean columns into integers (1/0) or revert integers back to boolean, depending on the `conv_bool_to_int` flag.
7. **Column Renaming (Optional)**:
   - If the `rename_columns` flag is set to `True`, renames specific columns for clarity or standardization.

Outcome

The `clean_data` function ensures that the dataset is systematically cleaned and transformed, making it suitable for both exploratory analysis and advanced modeling tasks. By combining all cleaning methods into a single pipeline, this function enhances efficiency, consistency, and flexibility in preprocessing.

In [ ]:
```
def clean_data(orginal, oneHot=True, rmOutliers=False,conv_bool_to_int=False,rename_columns=False):
    cleaned = orginal.copy()
    if oneHot: encode(cleaned)
    remove_duplicates(cleaned)
    check_duplicates(cleaned)
    if rmOutliers: remove_outliers(cleaned)
    impute_missing_values(cleaned)
    check_missingness(cleaned)
    convert_to_bool(cleaned,conv_bool_to_int)
    if rename_columns: cleaned.rename(columns={'track_id': 'id','track_genre':'genre'}, inplace=True)
    return cleaned
```

Clean()

In [ ]:
```
# Clean the data
spotify_df_cleaned = clean_data(spotify_df)
spotify_df_cleaned.describe()
```

```
Num rows before removing duplicates: 114000
Num rows after removing exact duplicates: 113550
Num rows involved in ID duplicates (including original rows): 23809
Num rows after aggreagting inexact duplicates: 89741

Analyzing duplicates...

Found 0 exact duplicate rows

Found 0 rows with duplicate track_ids

No duplicates found in dataset
Analyzing missingness...

Percentage of missing values per column (sorted in descending order):
Series([], dtype: float64)

No missing values found in the dataset.
```

Out[ ]:

	popularity	duration_ms	danceability	energy	loudness	speechiness	acousticness	instrumentalness
count	89741.000000	8.974100e+04	89741.000000	89741.000000	89741.000000	89741.000000	89741.000000	89741.000000
mean	33.199533	2.291418e+05	0.562166	0.634458	-8.499004	0.087442	0.328289	0.173413
std	20.575442	1.129477e+05	0.176691	0.256605	5.221490	0.113277	0.338321	0.323848
min	0.000000	0.000000e+00	0.000000	0.000000	-49.531000	0.000000	0.000000	0.000000
25%	19.000000	1.730400e+05	0.450000	0.457000	-10.322000	0.036000	0.017100	0.000000
50%	33.000000	2.132930e+05	0.576000	0.676000	-7.185000	0.048900	0.188000	0.000058
75%	49.000000	2.642930e+05	0.692000	0.853000	-5.108000	0.085900	0.625000	0.097600
max	100.000000	5.237295e+06	0.985000	1.000000	4.532000	0.965000	0.996000	1.000000

4. Exploratory Data Analysis

Correlation Analysis Between Popularity and Numeric Features

Bar Chart

This analysis explores the relationships between the popularity of tracks and other numeric features in the dataset. By calculating Pearson correlation coefficients, we quantify the strength and direction of these relationships.

The process involves creating a correlation matrix that identifies how strongly each numeric feature relates to popularity. Instead of visualizing the entire matrix, we focus on plotting a bar chart to highlight the most important correlations. This approach makes it easier to interpret and prioritize features based on their influence on popularity.

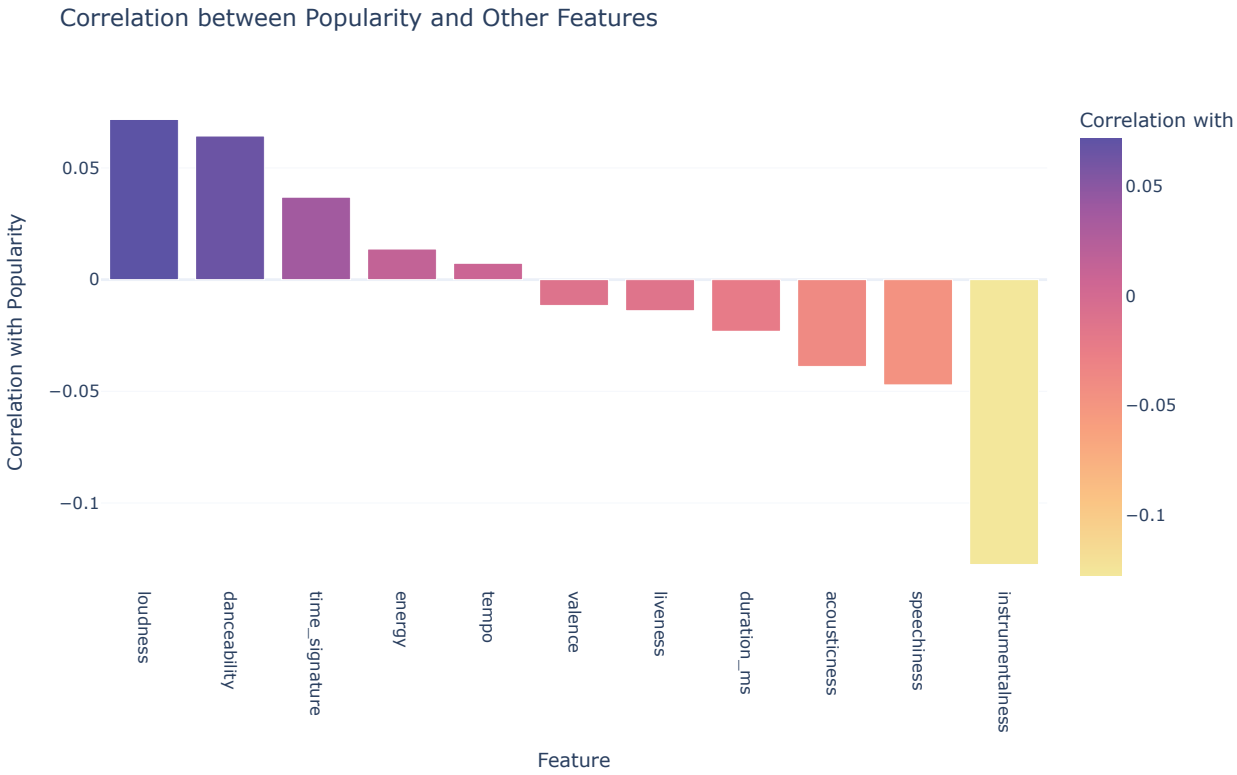
Key Steps:

1. **Data Preparation**:
    - Filter numeric columns to ensure only relevant features are included in the analysis.
    - Handle missing data by ensuring sufficient non-NaN values for meaningful calculations.

2. **Correlation Calculation**:
    - Compute Pearson correlation coefficients between `popularity` and other numeric features.
    - Store the results in a structured format for visualization.
3. **Bar Chart Visualization**:
    - Display the correlations as a bar chart where:
        - Features are shown on the x-axis.
        - Correlation coefficients are shown on the y-axis.
    - A color gradient is applied to emphasize the strength of the correlation.

The bar chart provides an intuitive view of which features have strong positive or negative correlations with popularity. Features with higher absolute correlations can be prioritized for further analysis or modeling. This step offers a data-driven basis for understanding the factors influencing a track's success.

In [ ]:
```
numeric_df = spotify_df_cleaned.select_dtypes(include=[np.number])
# Initialize an empty dictionary to store correlations
correlation_data = {}

# Calculate correlation coefficients between popularity and other numeric variables
for column in numeric_df.columns:
    if column != 'popularity':  # Skip popularity itself
        # Ensure the column has at least two valid (non-NaN) values
        if numeric_df[column].notna().sum() >= 2:
            corr, _ = sp.stats.pearsonr(numeric_df['popularity'], numeric_df[column])
            correlation_data[column] = corr
        else:
            print(f"Skipping {column} due to insufficient data for correlation calculation.")

# Convert to DataFrame for easier plotting
correlation_df = pd.DataFrame(list(correlation_data.items()), columns=['Feature', 'Correlation'])
correlation_df = correlation_df.sort_values(by="Correlation", ascending=False)

# Plotting with Plotly Express
fig = px.bar(
    correlation_df,
    x='Feature',
    y='Correlation',
    title='Correlation between Popularity and Other Features',
    labels={'Feature': 'Feature', 'Correlation': 'Correlation with Popularity'},
    template='plotly_white',
    color='Correlation',
    color_continuous_scale=px.colors.sequential.Sunset
)

fig.update_layout(
    xaxis_title="Feature",
    yaxis_title="Correlation with Popularity",
    xaxis_tickangle=90,
    height=600,
    width=1000
)

fig.show()
```

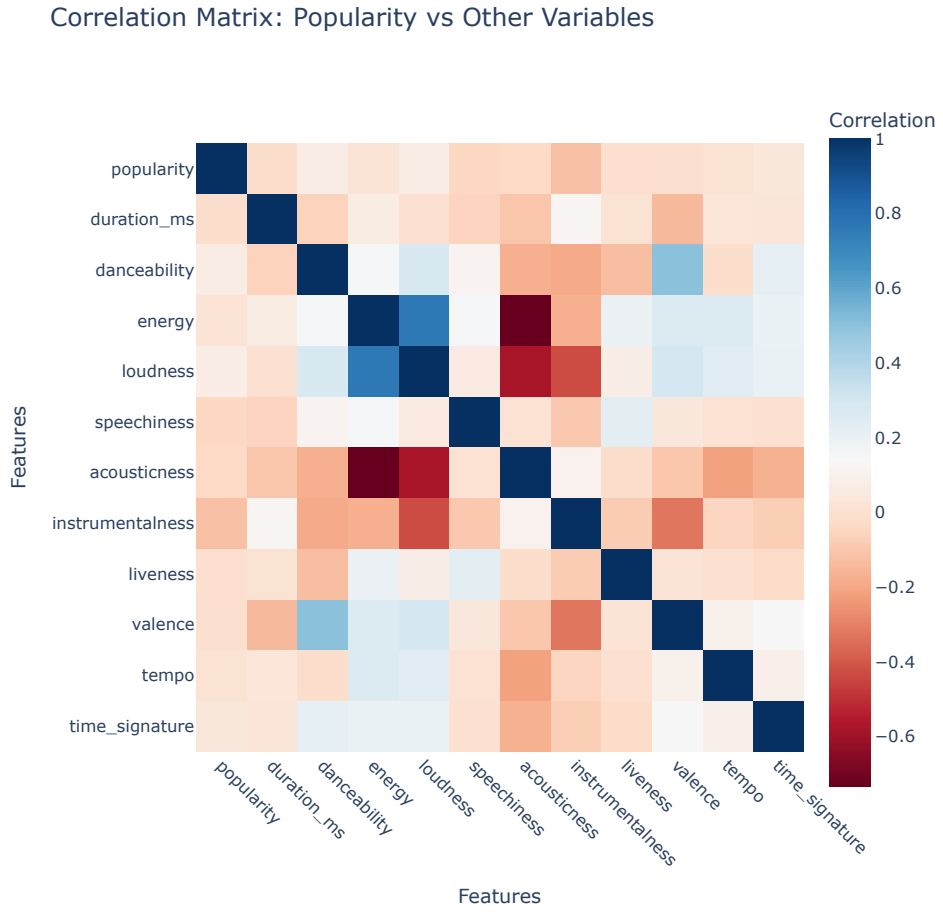


Heat Map

To visually represent the results, a heat map is generated from the correlation matrix. This allows for quick identification of features with strong positive or negative correlations with popularity, providing insights into the factors that might influence a track's success. For example, features such as tempo, danceability, or energy could exhibit significant correlations with popularity.

The heat map provides an intuitive way to understand the structure of the data, highlighting the most influential features. This step is crucial for feature selection and serves as a foundation for predictive modeling and further exploratory analysis.

In [ ]:
```
# Set up numerical columns
numerical_columns = spotify_df_cleaned.select_dtypes(include=[np.number]).columns

# Correlation matrix
corr_matrix = spotify_df_cleaned[numerical_columns].corr()

# Create the heatmap using Plotly Express
fig = px.imshow(
    corr_matrix,
    labels={'x': 'Features', 'y': 'Features', 'color': 'Correlation'},
    x=numerical_columns,
    y=numerical_columns,
    color_continuous_scale=px.colors.diverging.RdBu,  # Use a diverging colorscale like RdBu
    title='Correlation Matrix: Popularity vs Other Variables'
)

# Update layout for better readability
fig.update_layout(
    title_font_size=18,
    xaxis_tickangle=45,
    height=700,
    width=700
)

# Display the heatmap
fig.show()
```



Histograms of numeric features

Histograms are an essential tool for exploratory data analysis (EDA), providing insights into the distribution of numeric variables in the dataset. By plotting histograms for each numeric feature, we can achieve the following:

1. **Understand Data Distribution**:
   - Visualize how each numeric variable is distributed (e.g., normal, skewed, uniform).
   - Identify if the data is centered around specific values or spread out across a range.
2. **Detect Outliers**:
   - Highlight extreme values that might affect the analysis or modeling process.
   - Determine whether these outliers are valid or the result of errors.
3. **Assess Data Quality**:
   - Check for issues such as missing data or irregular value distributions.
   - Confirm the range of values aligns with expectations for each feature.§

In [ ]:
```python
# Function to plot histograms for numeric variables in a 3-wide subplot layout
def plot_numeric_histograms_subplot(df, title_prefix="Distribution of"):
    """
    Plots histograms for all numeric variables in the DataFrame in a 3-wide subplot layout.

    Parameters:
    df (pandas.DataFrame): Input DataFrame containing numeric variables.
    title_prefix (str): Prefix for histogram titles.

    Returns:
    None (displays a single subplot figure).
    """
    # Select numeric columns
    numeric_columns = df.select_dtypes(include=[np.number]).columns

    # Determine number of rows needed for 3 columns per row
    num_cols = 3
    num_rows = (len(numeric_columns) + num_cols - 1) // num_cols

    # Create a subplot figure
    fig = sbplt.make_subplots(rows=num_rows, cols=num_cols, subplot_titles=[f"{title_prefix} {col}" for col in nu

    # Add histograms to the subplots
```

```
    for i, column in enumerate(numeric_columns):
        row = i // num_cols + 1
        col = i % num_cols + 1
        histogram = go.Histogram(x=df[column], nbinsx=30, name=column)
        fig.add_trace(histogram, row=row, col=col)

    # Update layout for aesthetics
    fig.update_layout(
        height=num_rows * 300,  # Adjust height dynamically based on rows
        width=1000,             # Fixed width for consistent appearance
        title_text="Numeric Variable Distributions",
        template="plotly_white",
        showlegend=False        # Hide legend for simplicity
    )
    fig.update_annotations(font_size=12)  # Adjust font size for subplot titles

    # Show the figure
    fig.show()
```

In [ ]:
```
plot_numeric_histograms_subplot(spotify_df_cleaned)
```

Bar Plots for Categorical Values

Bar charts provide an intuitive and visually appealing way to analyze the distribution of categorical features in the dataset. This analysis focuses on the `key_`, `mode_`, and `track_genre_` features, which represent important musical attributes of tracks. By visualizing their distributions, we can gain insights into the underlying patterns of the data.

Why Analyze These Features?

1. **Key Distribution**:
   - The `key_` features represent the musical key in which tracks are composed (e.g., C, D, G).

- Understanding the distribution of keys can provide insights into the musical preferences or patterns in the dataset.
- This analysis can reveal whether certain keys are more common, potentially reflecting trends in music production or listener preferences.

2. **Mode Distribution**:

- The `mode_` features indicate whether a track is in a major or minor key.
- Visualizing the distribution of modes helps understand the emotional tone of the music, as major keys are often associated with happiness and energy, while minor keys evoke sadness or intensity.
- The balance between major and minor modes may reveal interesting trends in music composition.

3. **Genre Distribution**:

- The `track_genre_` features represent the genres assigned to each track (e.g., pop, rock, jazz).
- Analyzing genre distribution is critical for understanding the diversity of the dataset and identifying dominant genres.
- This can help prioritize genres for deeper analysis or tailor models for specific use cases.

Benefits of Visualization

1. **Data Quality Checks**:

- Bar charts allow for quick identification of anomalies, such as missing or improperly encoded data in keys, modes, or genres.

2. **Pattern Identification**:

- Highlights the prevalence of certain keys, modes, or genres, revealing patterns or trends that could guide further analysis.

3. **Modeling Insights**:

- Understanding these distributions helps inform feature engineering for machine learning models.
- For example, imbalanced distributions may suggest a need for rebalancing or targeted modeling.

4. **Actionable Insights**:

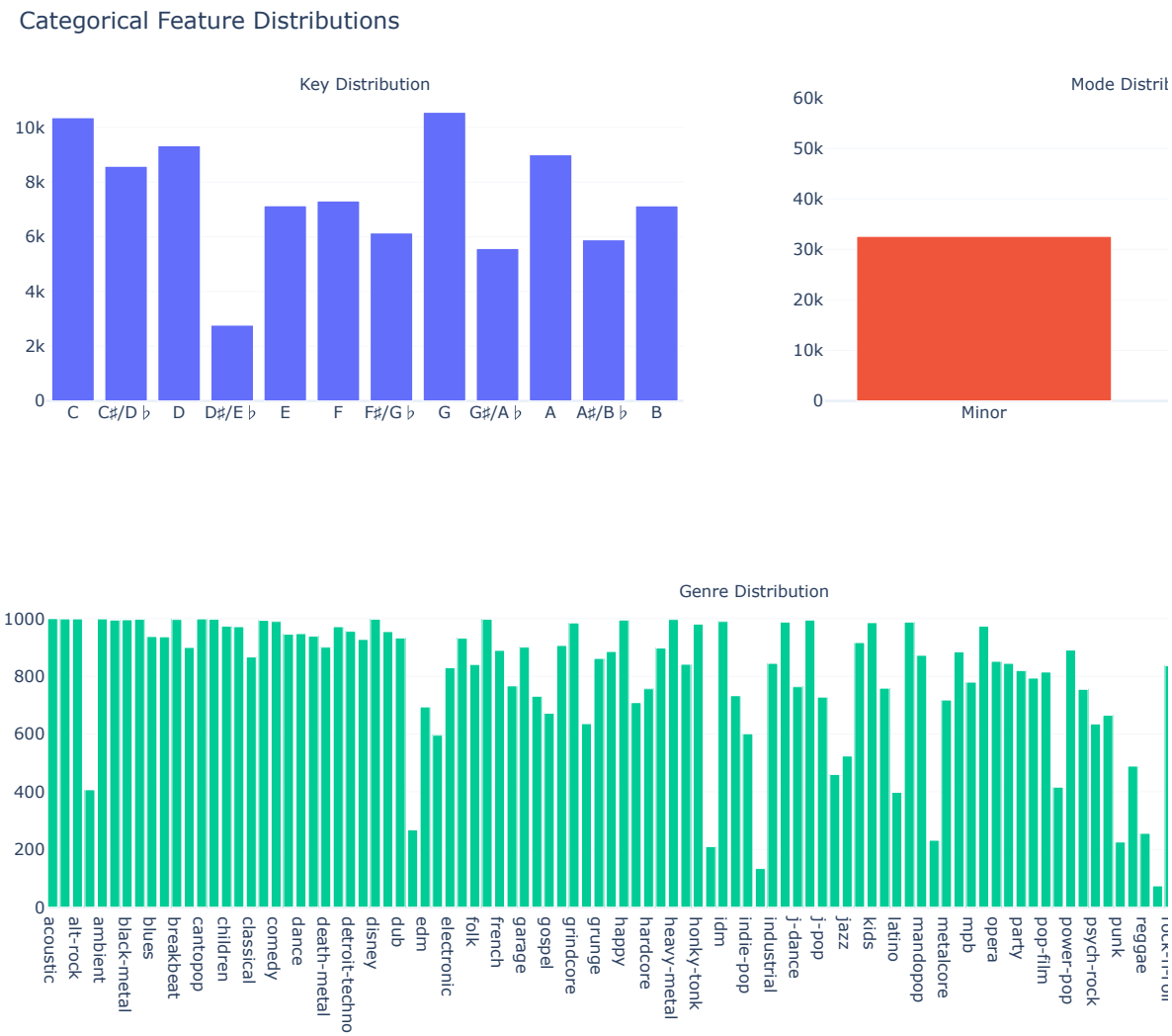
- Insights from these distributions can be used to tailor recommendations, prioritize popular keys or genres, and create meaningful segmentation for deeper analysis.

By visualizing these features, we ensure a comprehensive understanding of the musical attributes represented in the dataset, forming the foundation for more advanced analytical tasks.

In [ ]:
```
# Refactored code for having keys and modes on the first row, and genres spanning the bottom row
def plot_categorical_bars_subplot(df, key_prefixes=['key_'], mode_prefixes=['mode_'], genre_prefixes=['track_genre
    """
    Creates bar charts for the counts of key_, mode_, and genre_ features with a custom layout.
    Keys and modes are on the first row, and genres span the entire bottom row.

    Parameters:
    df (pandas.DataFrame): Input DataFrame containing one-hot encoded features.
    key_prefixes (list): List of prefixes to identify key columns.
    mode_prefixes (list): List of prefixes to identify mode columns.
    genre_prefixes (list): List of prefixes to identify genre columns.

    Returns:
    None (displays a single subplot figure).
    """
    # Identify columns based on prefixes
    key_columns = [col for col in df.columns if any(col.startswith(prefix) for prefix in key_prefixes)]
    mode_columns = [col for col in df.columns if any(col.startswith(prefix) for prefix in mode_prefixes)]
    genre_columns = [col for col in df.columns if any(col.startswith(prefix) for prefix in genre_prefixes)]

    # Create data for each feature type
    key_counts = df[key_columns].sum().reset_index()
    key_counts.columns = ['Feature', 'Count']
    key_counts['Feature'] = key_counts['Feature'].str.replace('key_', '', regex=False)

    mode_counts = df[mode_columns].sum().reset_index()
    mode_counts.columns = ['Feature', 'Count']
    mode_counts['Feature'] = mode_counts['Feature'].str.replace('mode_', '', regex=False)

    genre_counts = df[genre_columns].sum().reset_index()
    genre_counts.columns = ['Feature', 'Count']
    genre_counts['Feature'] = genre_counts['Feature'].str.replace('track_genre_', '', regex=False)

    # Titles for subplots
    titles = ['Key Distribution', 'Mode Distribution', 'Genre Distribution']

    # Create a subplot figure
    fig = sbplt.make_subplots(
        rows=2, cols=2,
        subplot_titles=titles,
        specs=[[{"colspan": 1}, {"colspan": 1}], [{"colspan": 2}, None]]
    )

    # Add bar charts for keys and modes in the first row
    fig.add_trace(go.Bar(x=key_counts['Feature'], y=key_counts['Count'], name=titles[0]), row=1, col=1)
    fig.add_trace(go.Bar(x=mode_counts['Feature'], y=mode_counts['Count'], name=titles[1]), row=1, col=2)
```

```
    # Add bar chart for genres spanning the second row
    fig.add_trace(go.Bar(x=genre_counts['Feature'], y=genre_counts['Count'], name=titles[2]), row=2, col=1)

    # Update layout for aesthetics
    fig.update_layout(
        height=800,  # Adjust height
        width=1200,  # Adjust width
        title_text="Categorical Feature Distributions",
        template="plotly_white",
        showlegend=False  # Hide legend for simplicity
    )
    fig.update_annotations(font_size=12)  # Adjust font size for subplot titles

    # Show the figure
    fig.show()
```

In [ ]:
```
plot_categorical_bars_subplot(spotify_df_cleaned)
```



5. Regression Analysis

In [ ]:
```
# Load preprocessed dataset
spotify_df = spotify_df
spotify_clean = spotify_df_cleaned.copy()
spotify_clean['explicit'] = spotify_clean['explicit'].astype(int)
# spotify_clean
```

A numeric response variable that we will model for regression is danceability.

The single variable that we will use as a predictor is energy. We chose the response and predictor variables based on their higher correlation which makes more sense for a single linear regression model. Choosing popularity would most likely show a nonlinear relationship.

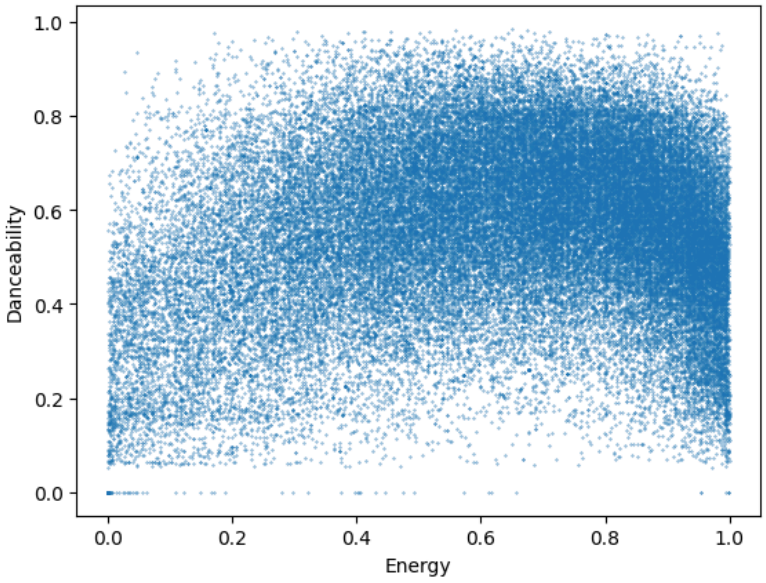
In [ ]:
```
# Separate data frame into test, training, and validation sets
# train = 70%, test = 15%, validation = 15%
spotify_train, temp_data = train_test_split(spotify_clean, test_size=0.30, random_state=42)
# Then, split the temp_data into validation and test sets
spotify_valid, spotify_test = train_test_split(temp_data, test_size=0.5, random_state=42)

print(spotify_train.shape)
print(spotify_valid.shape)
print(spotify_test.shape)
```

```
(62818, 142)
(13461, 142)
(13462, 142)
```

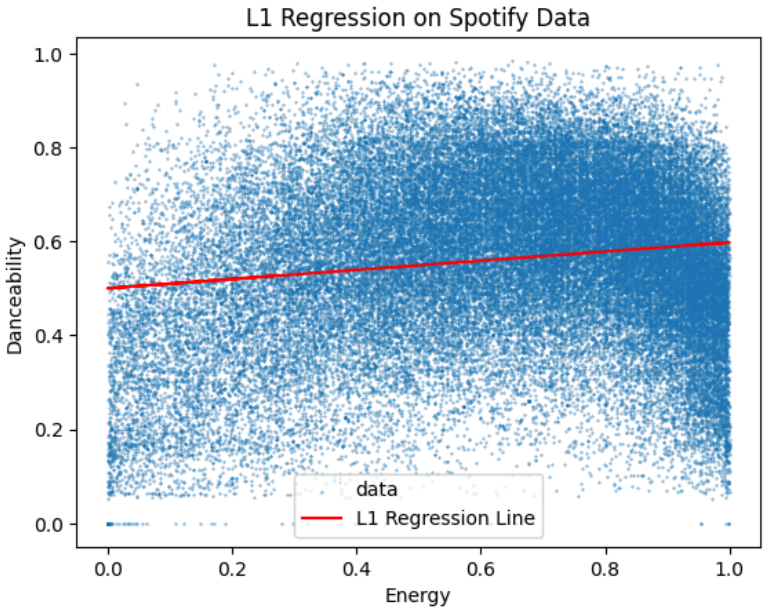
L1 Regression Model

In [ ]:
```
# Plot a scatter plot of energy vs danceability
plt.scatter(spotify_train["energy"], spotify_train["danceability"], s = 0.1)
plt.xlabel("Energy")
plt.ylabel("Danceability")
```

Out[ ]:
```
Text(0, 0.5, 'Danceability')
```



In [ ]:
```
X_train = spotify_train["energy"].values.reshape(-1, 1)  # Convert to 2D array
y_train = spotify_train["danceability"]

model = Lasso(alpha=0, max_iter=10000)  # alpha = 0 means no regularization
model.fit(X_train, y_train)

# Predict
y_pred = model.predict(X_train)

# Plot actual data
plt.scatter(spotify_train["energy"], spotify_train["danceability"], s=0.1, label='data')

# Plot predicted data (L1 regression line)
plt.plot(spotify_train["energy"], y_pred, color='red', label='L1 Regression Line')  # Use plot for a smooth line

# Add labels and legend
plt.xlabel('Energy')
plt.ylabel('Danceability')
plt.title('L1 Regression on Spotify Data')
plt.legend()

plt.show()
```



The L1 regression model with no regularization appears to be underfitting the data. While it looks like there is a non-linear curve to the relationship between energy and danceability, the regression line does not capture this. Increasing the degree of the polynomial might help with reducing the underfitting.

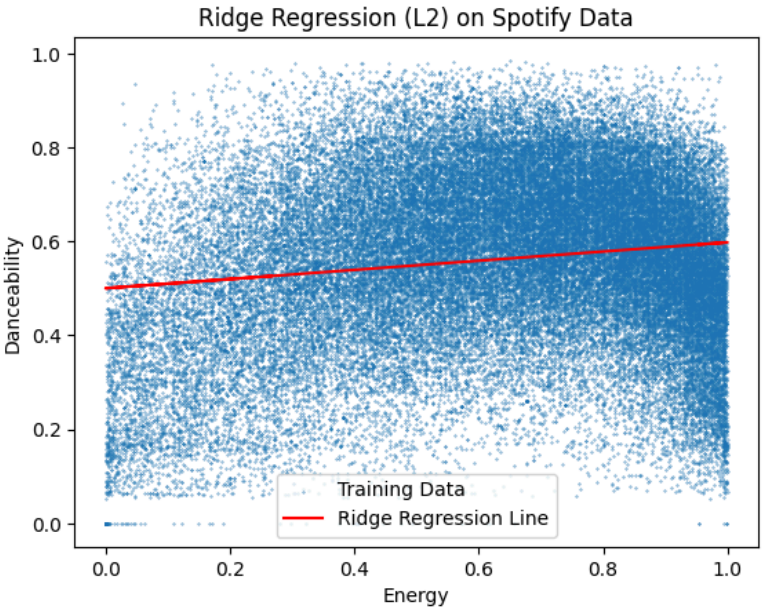
L1 Metrics: MAD & MAE

In [ ]:
```
#Mean Absolute Deviation (MAD)
y_mean = np.mean(y_train)
mad = np.mean(np.abs(y_train - y_mean))

#Mean Absolute Error (MAE)
mae = mean_absolute_error(y_train, y_pred)

print(mad)
print(mae)
```

```
0.1422003832581965
0.1409759073049339
```

L2 Regression Model

In [ ]:
```
#predictor variables
X_train = spotify_train["energy"].values.reshape(-1, 1)
y_train = spotify_train["danceability"]

X_valid = spotify_valid["energy"].values.reshape(-1, 1)
y_valid = spotify_valid["danceability"]

#L2 regression
ridge_model = Ridge(alpha=0.0, max_iter=10000)  # Adjust alpha if needed
ridge_model.fit(X_train, y_train)

y_train_pred_ridge = ridge_model.predict(X_train)
y_valid_pred_ridge = ridge_model.predict(X_valid)

plt.scatter(spotify_train["energy"], spotify_train["danceability"], s=0.1, label='Training Data')
plt.plot(spotify_train["energy"], y_train_pred_ridge, color='red', label='Ridge Regression Line')
plt.xlabel('Energy')
plt.ylabel('Danceability')
plt.title('Ridge Regression (L2) on Spotify Data')
plt.legend()
plt.show()
```



The Ridge regression line is almost flat, indicating that the model is not capturing much of the variation in danceability. The scatter of the data points suggests a more complex, non-linear relationship, but the regression line is linear and very simplistic.

L2 Metrics: rMSE

In [ ]:
```
#true values
y_train = spotify_train["danceability"]

y_pred = model.predict(X_train)

#residuals (errors)
error = y_train - y_pred

squared_error = error ** 2

#MSE calculation
L2_error = np.mean(squared_error)

#square root for MSE
rMSE = np.sqrt(L2_error)

print("rMSE:", rMSE)
```

```
rMSE: 0.17484623339039435
```

Since both L1 and L2 regression models appear to be underfitting the data and scatter plots suggest a non-linear relationship between energy and danceability, regularization is not necessary. Regularization is primarily a technique to prevent overfitting, not underfitting.

Logistic Regression Analysis

In [ ]:
```
from sklearn.model_selection import cross_val_score
import plotly.express as px
import plotly.figure_factory as ff
import plotly.graph_objects as go
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone
```

Binary Categorical Response Variable/Predictor Variables

The binary categorical response variable we chose was explicitness. We wanted to determine whether speechiness, our predictor variable, could predict whether a song was explicit. We believe that there may be a relationship since rap songs are usually more speechy and tend to be more explicit.

In [ ]:
```
spotify_train, temp_data = train_test_split(spotify_clean, test_size=0.40, random_state=42)

spotify_valid, spotify_test = train_test_split(temp_data, test_size=0.5, random_state=42)

X_train = spotify_train[["speechiness"]]
y_train = spotify_train["explicit"]
```

```
X_val = spotify_valid[["speechiness"]]
y_val = spotify_valid["explicit"]
```

Modeling Logistic Regression

In [ ]:
```
# Binary response variable = explicitness
# Initialize the logistic regression model
lr_all = LogisticRegression(solver='liblinear')

# Fit the model
lr_all.fit(X_train, y_train)

# Get the intercept and coefficients
intercept = lr_all.intercept_
coefficients = lr_all.coef_

print("Intercept:", intercept)
print("Coefficients:", coefficients)
```

```
Intercept: [-3.01164052]
Coefficients: [[5.61027602]]
```

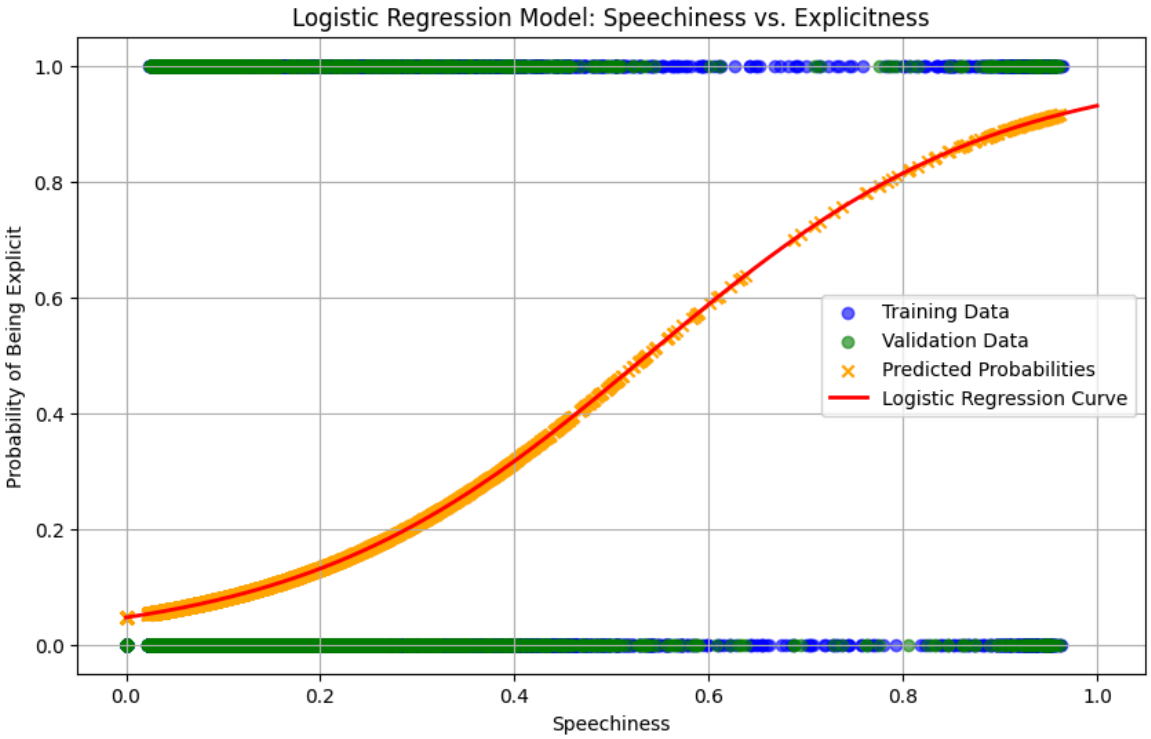
In [ ]:
```
pred_val_sample = pd.DataFrame(dict(
    explicit=y_val,  # Use y_val for actual values
    lr_predict=lr_all.predict_proba(X_val)[:, 1],  # Predicted probabilities
    lr_predict_binary=lr_all.predict(X_val)  # Binary predictions
))
pred_val_sample
```

Out[ ]:

	explicit	lr_predict	lr_predict_binary
26135	0	0.058665	0
24370	0	0.060202	0
21286	1	0.098413	0
19905	0	0.161333	0
73536	0	0.063090	0
...	...	...	...
8334	0	0.059007	0
80048	0	0.071050	0
20993	1	0.235578	0
40304	0	0.054649	0
3586	0	0.055056	0

17948 rows × 3 columns

In [ ]:
```
# Generate predictions over a range of speechiness values
speechiness_range = np.linspace(0, 1, 300).reshape(-1, 1)
predicted_probabilities = lr_all.predict_proba(speechiness_range)[:, 1]

plt.figure(figsize=(10, 6))

# Plot training data
plt.scatter(X_train, y_train, color='blue', alpha=0.6, label='Training Data')

# Plot validation data with predicted probabilities
plt.scatter(X_val, y_val, color='green', alpha=0.6, label='Validation Data')
plt.scatter(X_val, pred_val_sample['lr_predict'], color='orange', marker='x', label='Predicted Probabilities')

plt.plot(speechiness_range, predicted_probabilities, color='red', linewidth=2, label='Logistic Regression Curve')

plt.xlabel('Speechiness')
plt.ylabel('Probability of Being Explicit')
plt.title('Logistic Regression Model: Speechiness vs. Explicitness')
plt.legend()
plt.grid(True)
plt.show()
```



The curve shows a clear positive relationship between speechiness and the likelihood of a song being explicit. This supports the hypothesis that songs with higher speechiness (e.g., rap songs) are more likely to be explicit.

Around a speechiness value of 0.5 to 0.6, the probability of being explicit starts to exceed 50%.

The predictions generally align well with the data points, indicating that the logistic regression model captures the trend effectively.

Confusion Matrix

In [ ]:
```
conf_lr = metrics.confusion_matrix(y_true=pred_val_sample['explicit'],
                                   y_pred=pred_val_sample['lr_predict_binary'])
conf_lr
```

Out[ ]:
```
array([[16301,    80],
       [ 1467,   100]])
```

The model struggles to correctly identify explicit songs, as shown by the 1,432 false negatives.

The model rarely misclassifies non-explicit songs as explicit (only 70 instances).

Prediction Accuracy

In [ ]:
```
# (conf_lr[0, 0] + conf_lr[1, 1]) / conf_lr.sum()
pred_acc = metrics.accuracy_score(y_true=pred_val_sample['explicit'],
                      y_pred=pred_val_sample['lr_predict_binary'])

pred_acc
```

Out[ ]:
```
0.9138065522620905
```

Prediction Error

In [ ]:
```
pred_err = 1 - pred_acc
pred_err
```

Out[ ]:
```
0.08619344773790949
```

True Positive Rate (Sensitivity/Recall)

In [ ]:
```
# (conf_lr[1, 1]) / conf_lr[1,:].sum()
tpr = metrics.recall_score(y_true=pred_val_sample['explicit'],
                    y_pred=pred_val_sample['lr_predict_binary'])
print(tpr)
```

```
0.06381620931716656
```

The model is only correctly identifying about 6.5% of the actual explicit songs.

This may suggest that there may be far more non-explicit songs than explicit ones in the dataset, leading to a bias toward predicting the majority class (non-explicit). The model might also be overly conservative when predicting explicit songs.

True Negative Rate

In [ ]:
```
# (conf_lr[0, 0]) / conf_lr[0,:].sum()
tnr = metrics.recall_score(y_true=pred_val_sample['explicit'],
                           y_pred=pred_val_sample['lr_predict_binary'],
                           pos_label=0)
print(tnr)
```

```
0.9951162932665893
```

The model correctly identifies 99.57% of the non-explicit songs. It is excellent at identifying non-explicit songs.

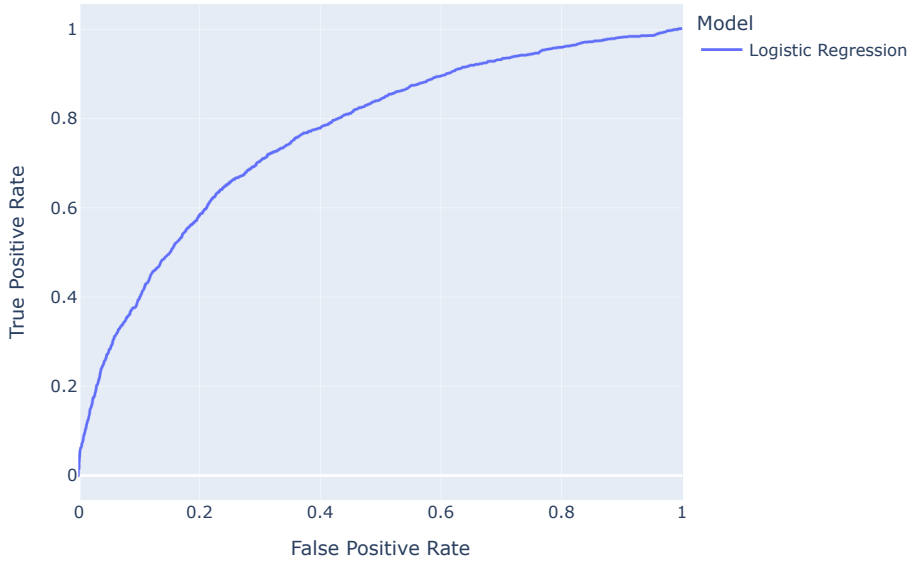
ROC Curve

ROC curves plot true positive rates against true negative rates to compare models and algorithms.

In [ ]:
```
lr_fpr_sample, lr_tpr_sample, lr_thresholds_sample = metrics.roc_curve(pred_val_sample['explicit'], pred_val_samp
lr_thresholds_sample
```

Out[ ]:
```
array([       inf, 0.91571066, 0.91527662, ..., 0.05296392, 0.05276728,
       0.04690276])
```

In [ ]:
```
roc_lr_sample = pd.DataFrame({
    'False Positive Rate': lr_fpr_sample,
    'True Positive Rate': lr_tpr_sample,
    'Model': 'Logistic Regression'
}, index=lr_thresholds_sample)


roc_sample_df = pd.concat([roc_lr_sample])


px.line(roc_sample_df, y='True Positive Rate', x='False Positive Rate',
        color='Model',
        width=700, height=500
)
```



The curve suggests that the model performs reasonably well but not perfectly.

AUC

In [ ]:
```
lr_auc_sample = metrics.roc_auc_score(pred_val_sample['explicit'], pred_val_sample['lr_predict'])
print('Logistic regression AUC:', lr_auc_sample.round(3))
```

```
Logistic regression AUC: 0.764
```

AUC = 0.763 indicates that the model has a fair performance in distinguishing between explicit and non-explicit songs.

Cross-Fold Validation

In [ ]:
```
# This does stratified Kfolds for us...
roc_auc_scores = cross_val_score(lr_all, X_train, y_train, cv=5, scoring='roc_auc')
roc_auc_scores
```

Out[ ]:
```
array([0.78082547, 0.79623763, 0.78132138, 0.77171201, 0.77740713])
```

In [ ]:
```
# Use the shuffle and random state if want data shuffled before splitting
X = spotify_train[["speechiness"]]  # Feature set
y = spotify_train["explicit"]        # Target variable
skfolds = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)


i = 1
for train_index, test_index in skfolds.split(X, y):
    clone_lr = clone(lr_all)  # Clone the model to avoid fitting multiple times
    X_train_folds = X.iloc[train_index]
    y_train_folds = y.iloc[train_index]
    X_test_fold = X.iloc[test_index]
    print("Test indices for fold:", test_index)
    clone_lr.fit(X_train_folds, y_train_folds)
    y_pred = clone_lr.predict(X_test_fold)

    # Calculate AUC and Accuracy
    auc_sample = metrics.roc_auc_score(y.iloc[test_index], y_pred)
    accuracy_sample = metrics.accuracy_score(y.iloc[test_index], y_pred)

    # Print results
    print('Fold: ', i)
    print('AUC: ', auc_sample)
    print('Accuracy: ', accuracy_sample)

    i+=1
```

```
Test indices for fold: [   10    16    22 ... 53837 53838 53842]
Fold:  1
AUC:  0.5350748178493747
Accuracy:  0.9163339214411738
Test indices for fold: [    0     6     7 ... 53836 53839 53843]
Fold:  2
AUC:  0.5423986625776989
Accuracy:  0.9154981892469124
Test indices for fold: [    1    25    28 ... 53820 53827 53833]
Fold:  3
AUC:  0.5329961527403109
Accuracy:  0.9142910205218683
Test indices for fold: [    2     3     9 ... 53823 53835 53841]
Fold:  4
AUC:  0.5334027807773141
Accuracy:  0.9150338935834339
Test indices for fold: [    4     5     8 ... 53831 53832 53840]
Fold:  5
AUC:  0.5334701608288136
Accuracy:  0.9142830609212481
```

The AUC values are consistently low, around 0.53 to 0.54. The model is barely better than random chance at distinguishing explicit songs from non-explicit ones during each fold.

The accuracy is consistently high, around 91.4% to 91.7%. This reflects the class imbalance in the dataset, where the majority class (non-explicit) dominates predictions. The model's high accuracy likely comes from predicting most songs as non-explicit.

Threshold for positive predictions

In [ ]:
```
i = 1
for train_index, test_index in skfolds.split(X, y):
    clone_lr = clone(lr_all)  # Clone the model to avoid fitting multiple times
    X_train_folds = X.iloc[train_index]
    y_train_folds = y.iloc[train_index]
    X_test_fold = X.iloc[test_index]

    clone_lr.fit(X_train_folds, y_train_folds)  # Fit the model
    y_pred_proba = clone_lr.predict_proba(X_test_fold)[:, 1]  # Get predicted probabilities for the positive clas

    # Calculate the median threshold from predicted probabilities
    median_threshold = np.median(y_pred_proba)

    # Classify based on the median of predicted probabilities
    y_pred = (y_pred_proba >= median_threshold).astype(int)

    # Calculate AUC and Accuracy
    auc_sample = metrics.roc_auc_score(y.iloc[test_index], y_pred_proba)
    accuracy_sample = metrics.accuracy_score(y.iloc[test_index], y_pred)

    # Print results
    print('Fold: ', i)
```

```
    print('Median Threshold: ', median_threshold)
    print('AUC: ', auc_sample)
    print('Accuracy: ', accuracy_sample)

    i += 1
```

```
Fold:  1
Median Threshold:  0.06090896383406083
AUC:  0.7834813888320545
Accuracy:  0.5596619927569877
Fold:  2
Median Threshold:  0.0609876793451701
AUC:  0.7745722006910061
Accuracy:  0.557711950970378
Fold:  3
Median Threshold:  0.06091469357066971
AUC:  0.7840030152428796
Accuracy:  0.5582691057665521
Fold:  4
Median Threshold:  0.06102204976565319
AUC:  0.7810260028158554
Accuracy:  0.5571547961742037
Fold:  5
Median Threshold:  0.06085084147396123
AUC:  0.7844666670379168
Accuracy:  0.5605497771173849
```

The median threshold for classification is consistently low (around 0.060), indicating that the predicted probabilities for explicit songs are generally low.

The AUC scores range between 0.769 and 0.785, showing a moderate improvement compared to the previous results (around 0.53).

This suggests the model's ability to distinguish between explicit and non-explicit songs has improved.

The accuracy scores are now much lower (around 55.5% to 55.9%) compared to the previous 91%.

This drop in accuracy is due to the model now predicting more explicit songs, addressing the class imbalance but sacrificing overall accuracy for better balance.

The two thresholds that we tested were 0.5 (default) and the median of speechiness. In the first instance, we had a very high accuracy (0.914 to 0.917) but a very low AUC score (0.536 to 0.538). Area under the curve scores indicate whether a model is doing well at predicting, and since the AUC scores were close to 0.5, this indicates that our model was essentially guessing.

When we changed the threshold to the median of the predicted probabilities of whether a song was explicit or not, there was a higher AUC score (0.769 - 0.784) and a low accuracy score (0.555 - 0.559). Although the accuracy was low, we figured that AUC would be a better metric since upon further evaluation above, our dataset has class imbalances (more not explicit than explicit). Therefore, we thought that choosing the median of the predicted probabilities as the threshold would be the best option since it best manages class imbalances. By choosing this as our threshold, we were able to increase our AUC score. (The accuracy may be improved by reducing class imbalances)

Using the class_weight='balanced' parameter

In [ ]:
```
# count the number of explicit and non-explicit songs in the dataset
explicit_counts = spotify_clean['explicit'].value_counts()

# calculate the percentage distribution of each class
explicit_percentage = spotify_clean['explicit'].value_counts(normalize=True) * 100

explicit_counts, explicit_percentage
```

Out[ ]:
```
(explicit
 0    82037
 1     7704
 Name: count, dtype: int64,
 explicit
 0    91.415295
 1     8.584705
 Name: proportion, dtype: float64)
```

Class distribution in the dataset: Non-explicit songs (0): 82,036 (91.42%) Explicit songs (1): 7,704 (8.58%)

There is a significant class imbalance in the dataset, with far more non-explicit songs (91.4%) than explicit songs (8.6%). This imbalance leads to the model being biased towards predicting non-explicit songs more frequently.

We will try to use the class_weight='balanced' parameter in LogisticRegression to handle imbalance later. Non-explicit songs (majority class) get a lower weight while explicit songs (minority class) get a higher weight.

In [ ]:
```
# Binary response variable = explicitness
# Initialize the logistic regression model
lr_all = LogisticRegression(solver='liblinear', class_weight='balanced')
```

```
# Fit the model
lr_all.fit(X_train, y_train)

# Get the intercept and coefficients
intercept = lr_all.intercept_
coefficients = lr_all.coef_

print("Intercept:", intercept)
print("Coefficients:", coefficients)
```

```
Intercept: [-1.00138]
Coefficients: [[8.65276072]]
```

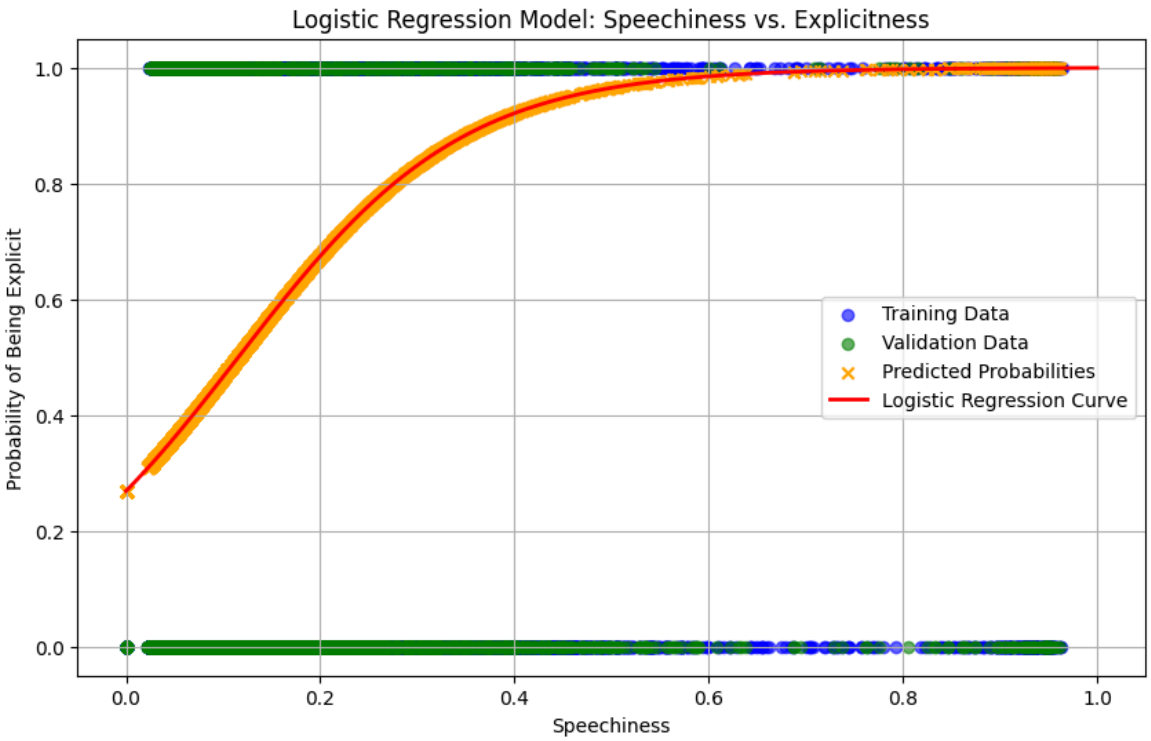
In [ ]:
```
pred_val_sample = pd.DataFrame(dict(
    explicit=y_val,  # Use y_val for actual values
    lr_predict=lr_all.predict_proba(X_val)[:, 1],  # Predicted probabilities
    lr_predict_binary=lr_all.predict(X_val)  # Binary predictions
))
pred_val_sample
```

Out[ ]:

	explicit	lr_predict	lr_predict_binary
26135	0	0.345903	0
24370	0	0.355557	0
21286	1	0.556585	1
19905	0	0.750493	1
73536	0	0.373392	0
...	...	...	...
8334	0	0.348059	0
80048	0	0.420366	0
20993	1	0.861532	1
40304	0	0.320158	0
3586	0	0.322800	0

17948 rows × 3 columns

In [ ]:
```
# Generate predictions over a range of speechiness values
speechiness_range = np.linspace(0, 1, 300).reshape(-1, 1)
predicted_probabilities = lr_all.predict_proba(speechiness_range)[:, 1]

plt.figure(figsize=(10, 6))

# Plot training data
plt.scatter(X_train, y_train, color='blue', alpha=0.6, label='Training Data')

# Plot validation data with predicted probabilities
plt.scatter(X_val, y_val, color='green', alpha=0.6, label='Validation Data')
plt.scatter(X_val, pred_val_sample['lr_predict'], color='orange', marker='x', label='Predicted Probabilities')

plt.plot(speechiness_range, predicted_probabilities, color='red', linewidth=2, label='Logistic Regression Curve')

plt.xlabel('Speechiness')
plt.ylabel('Probability of Being Explicit')
plt.title('Logistic Regression Model: Speechiness vs. Explicitness')
plt.legend()
plt.grid(True)
plt.show()
```



The red logistic curve reaches a probability of 1 very quickly and stays flat across half of the speechiness range, suggesting that the model is predicting a high probability of explicitness for most speechiness values.

The orange crosses (predicted probabilities) follow the red curve closely and also saturate near 1. This shows that the model is biased toward predicting high probabilities of explicitness.

Even with class_weight='balanced', the model still struggles with the severe imbalance, causing it to predict explicitness frequently.

The default decision threshold of 0.5 might not be suitable for this dataset. Explicit predictions might need a higher threshold to balance sensitivity and specificity.

In [ ]:
```
y_val_pred = lr_all.predict(X_val)
y_val_proba = lr_all.predict_proba(X_val)[:, 1]

conf_matrix = metrics.confusion_matrix(y_true=y_val, y_pred=y_val_pred)

accuracy = accuracy_score(y_true=y_val, y_pred=y_val_pred)

sensitivity = metrics.recall_score(y_true=y_val, y_pred=y_val_pred)

specificity = metrics.recall_score(y_true=y_val, y_pred=y_val_pred, pos_label=0)

conf_matrix, accuracy, sensitivity, specificity
```

Out[ ]:
```
(array([[13985,  2396],
        [  797,   770]]),
 0.8220971696010697,
 np.float64(0.4913848117421825),
 np.float64(0.8537329833343508))
```

The sensitivity (49.67%) is significantly better compared to the earlier value (around 6%). The specificity remains relatively high at 85.52%.

The model is now better at identifying explicit songs but still struggles with false positives.

In [ ]:
```
auc_with_balanced = metrics.roc_auc_score(y_true=y_val, y_score=y_val_proba)
auc_with_balanced
```

Out[ ]:
```
np.float64(0.7643643446243599)
```

This AUC score indicates that the model with class weighting achieves 76.3% in distinguishing between explicit and non-explicit songs. This is consistent with the previous AUC score when class weights were not balanced.

The model now balances sensitivity and specificity better, even though the overall accuracy is slightly reduced.

In [ ]:
```
y_val_proba = lr_all.predict_proba(X_val)[:, 1]

# Initialize a range of thresholds from 0 to 1
thresholds = np.linspace(0, 1, 100)
```

```
sensitivities = []
specificities = []

for threshold in thresholds:
    y_val_pred_tuned = (y_val_proba >= threshold).astype(int)
    sensitivities.append(metrics.recall_score(y_val, y_val_pred_tuned))
    specificities.append(metrics.recall_score(y_val, y_val_pred_tuned, pos_label=0))

# Find the threshold that balances sensitivity and specificity
best_threshold_index = np.argmin(np.abs(np.array(sensitivities) - np.array(specificities)))
best_threshold = thresholds[best_threshold_index]

best_threshold, sensitivities[best_threshold_index], specificities[best_threshold_index]
```

Out[ ]:
```
(np.float64(0.393939393939394),
 np.float64(0.7121888959795788),
 np.float64(0.6886636957450705))
```

Best threshold: 0.404 This threshold achieves a balance between sensitivity and specificity.

At optimal threshold:

Sensitivity (True Positive Rate): 68.28%

Specificity (True Negative Rate): 71.33%

Using class_weight='balanced' improves sensitivity significantly. We have good model performance since the AUC score remains 0.763. Additionally, by adjusting the threshold to 0.404, we achieve a better balance between sensitivity and specificity.

6. KNN/Decision Trees/Random Forest

K-Nearest Neighbors Algorithm

In [ ]:
```
spotify_clean = spotify_df_cleaned.copy()
pop_labels = [1 if x >= 50 else 0 for x in spotify_clean["popularity"]]
spotify_clean['popularity'] = [1 if x >= 30 else 0 for x in spotify_clean["popularity"]]
spotify_clean['explicit'] = spotify_clean['explicit'].astype(int)
spotify_clean = spotify_clean.iloc[:,:20]
# Split the data into training, validation, and test sets
spotify_train, spotify_valid = train_test_split(spotify_clean, test_size=0.2, random_state=42)
spotify_clean
```

Out[ ]:

	track_id	popularity	duration_ms	explicit	danceability	energy	loudness	speechiness	acousticnes
0	0000vdREvCVMxbQTkS888c	1	160725	1	0.910	0.37400	-9.844	0.1990	0.07570
1	000CC8EParg64OmTxVnZ0p	1	322933	0	0.269	0.51600	-7.361	0.0366	0.40600
2	000Iz0K615UepwSJ5z2RE5	0	515360	0	0.686	0.56000	-13.264	0.0462	0.00114
3	000RDCYioLteXcutOjeweY	1	190203	0	0.679	0.77000	-3.537	0.1900	0.05830
4	000qpdoc97IMTBvF8gwcpy	0	331240	0	0.519	0.43100	-13.606	0.0291	0.00096
...	...	...	...	...	...	...	...	...	
89736	7zxHiMmVLt4LGWpOMqOpUh	1	325156	0	0.766	0.38200	-11.464	0.0324	0.69800
89737	7zxpdh3EqMq2JCkOI0EqcG	0	109573	0	0.529	0.00879	-32.266	0.0587	0.99600
89738	7zyYmIdjqqiX6kLryb7QBx	1	260573	0	0.423	0.36000	-9.458	0.0372	0.72800
89739	7zybSU9tFO9HNlwmGF7stc	1	234300	0	0.649	0.83400	-11.430	0.0397	0.26800
89740	7zz7iNGlWhmfFE7zlXkMma	0	144973	0	0.263	0.95700	-6.887	0.0897	0.00002

89741 rows × 20 columns

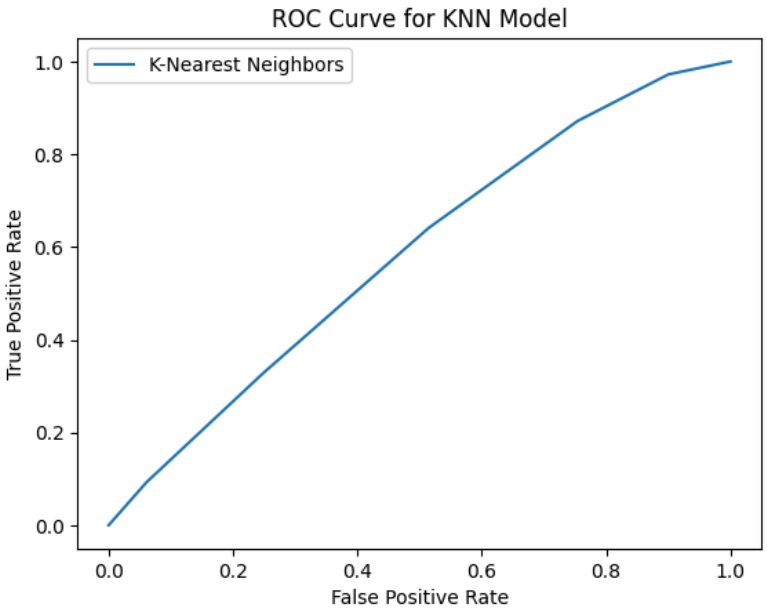
In [ ]:
```
X_train = spotify_train[["danceability", "speechiness"]]
y_train = spotify_train["popularity"]
X_val = spotify_valid[["danceability", "speechiness"]]
y_val = spotify_valid["popularity"]
```

In [ ]:
```
# 1. Initialize and Train the KNN Model
knn_model = KNeighborsClassifier(n_neighbors=5)  # You can adjust 'n_neighbors' based on model performance
knn_model.fit(X_train, y_train)

# Predict on validation data
knn_predictions = knn_model.predict(X_val)
knn_probabilities = knn_model.predict_proba(X_val)[:, 1]  # Probability for the positive class
```

In [ ]:
```
# 2. Calculate Confusion Matrix, Prediction Accuracy, Prediction Error, TPR, TNR, and F1 Score
conf_matrix = confusion_matrix(y_val, knn_predictions)
print("Confusion Matrix:\n", conf_matrix)

accuracy = accuracy_score(y_val, knn_predictions)
error = 1 - accuracy
print("Accuracy:", accuracy)
print("Error:", error)

tpr = recall_score(y_val, knn_predictions)
tnr = recall_score(y_val, knn_predictions, pos_label=0)
print("True Positive Rate:", tpr)
print("True Negative Rate:", tnr)
```

```
Confusion Matrix:
 [[3945 4184]
 [3524 6296]]
Accuracy: 0.5705610340408936
Error: 0.42943896595910636
True Positive Rate: 0.6411405295315682
True Negative Rate: 0.48529954483946364
```

In [ ]:
```
# 3. Calculate and Plot ROC Curve and AUC on Validation Set using 5-Fold Cross-Validation
fpr, tpr, thresholds = roc_curve(y_val, knn_probabilities)
plt.plot(fpr, tpr, label="K-Nearest Neighbors")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for KNN Model")
plt.legend()
plt.show()

auc = roc_auc_score(y_val, knn_probabilities)
print("AUC Score:", auc)

# 5-Fold Cross-Validation for AUC
cv_auc = cross_val_score(knn_model, X_train, y_train, cv=5, scoring="roc_auc")
print("5-Fold Cross-Validation AUC Scores:", cv_auc)
print("Average AUC Score:", cv_auc.mean())
```




```
AUC Score: 0.5854540168098976
5-Fold Cross-Validation AUC Scores: [0.57538898 0.57436976 0.57523259 0.57346124 0.57969729]
Average AUC Score: 0.5756299723016142
```

Random Forest Algorithm

In [ ]:
```
features = ['duration_ms',      'explicit',     'danceability', 'energy',       'loudness',     'speechiness',  '

X = spotify_clean[features]
y = spotify_clean['popularity']
```

In [ ]:
```
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```

In [ ]:
```
rf_model = RandomForestClassifier(random_state=42, n_estimators=100, max_depth=10)
rf_model.fit(X_train, y_train)
```

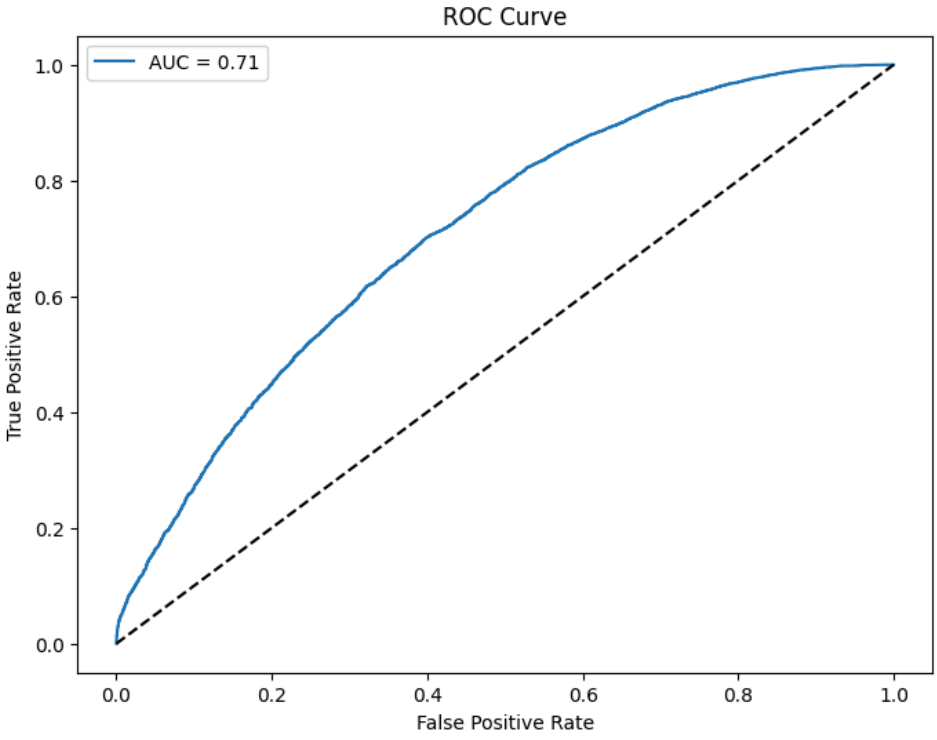
Out[ ]:
```
RandomForestClassifier
RandomForestClassifier(max_depth=10, random_state=42)
```

In [ ]:
```
y_val_pred = rf_model.predict(X_val)
y_test_pred = rf_model.predict(X_test)
y_test_prob = rf_model.predict_proba(X_test)[:, 1]
```

In [ ]:
```
val_mse = mean_squared_error(y_val, y_val_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
val_accuracy = accuracy_score(y_val, y_val_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
test_auc = roc_auc_score(y_test, y_test_prob)

print(f'Validation MSE: {val_mse}')
print(f'Test MSE: {test_mse}')
print(f'Validation Accuracy: {val_accuracy}')
print(f'Test Accuracy: {test_accuracy}')
print(f'Test AUC: {test_auc}')
```

```
Validation MSE: 0.33569199910853575
Test MSE: 0.3383475402529389
Validation Accuracy: 0.6643080008914642
Test Accuracy: 0.6616524597470611
Test AUC: 0.7103942718335018
```

In [ ]:
```
fpr, tpr, _ = roc_curve(y_test, y_test_prob)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'AUC = {test_auc:.2f}')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='best')
plt.show()
```



7. PCA & Clustering

In [ ]:
```
spotify_clean = spotify_df_cleaned.copy()
print(spotify_clean.shape)
spotify_clean['explicit'] = spotify_clean['explicit'].astype(int)

y_labels = [1 if x >= 50 else 0 for x in spotify_clean["popularity"]]
```

```
(89741, 142)
```

In [ ]:
```
pca_features = spotify_clean[["speechiness", "danceability", "energy", "valence", "tempo", "loudness", "acousticn
pca_features
```

Out[ ]:

	speechiness	danceability	energy	valence	tempo	loudness	acousticness
0	0.1990	0.910	0.37400	0.432	104.042	-9.844	0.075700
1	0.0366	0.269	0.51600	0.341	178.174	-7.361	0.406000
2	0.0462	0.686	0.56000	0.108	119.997	-13.264	0.001140
3	0.1900	0.679	0.77000	0.839	161.721	-3.537	0.058300
4	0.0291	0.519	0.43100	0.234	129.971	-13.606	0.000964
...	...	...	...	...	...	...	...
89736	0.0324	0.766	0.38200	0.672	119.992	-11.464	0.698000
89737	0.0587	0.529	0.00879	0.510	82.694	-32.266	0.996000
89738	0.0372	0.423	0.36000	0.291	130.576	-9.458	0.728000
89739	0.0397	0.649	0.83400	0.150	125.004	-11.430	0.268000
89740	0.0897	0.263	0.95700	0.204	172.932	-6.887	0.000023

89741 rows × 7 columns

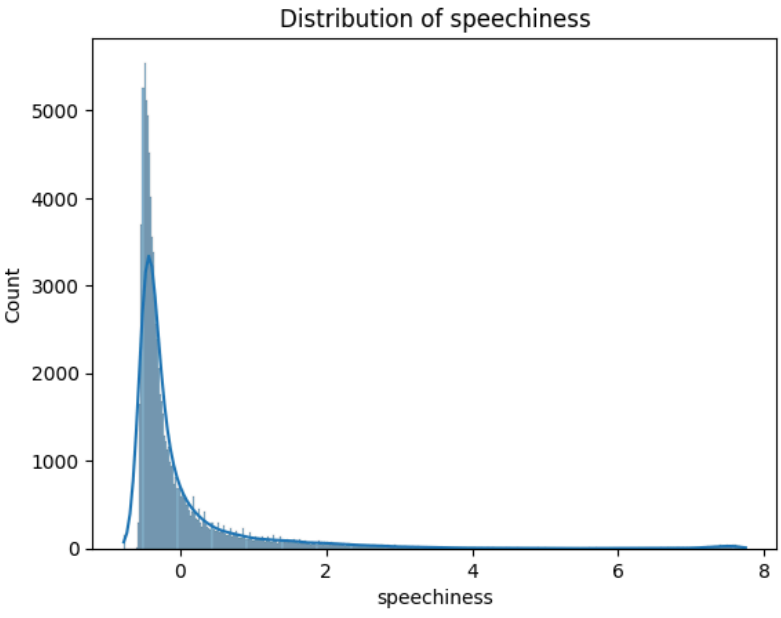
In [ ]:
```
# Standardize data (scale/mean-center)
std_scaler = StandardScaler()
spotify_scaled = std_scaler.fit_transform(pca_features.to_numpy())
spotify_scaled = pd.DataFrame(spotify_scaled, columns = ["speechiness", "danceability", "energy", "valence", "tem
print(f'Speechiness mean: {spotify_scaled["speechiness"].mean()}')
print(f'Speechiness standard deviation: {spotify_scaled["speechiness"].std()}')
print(spotify_scaled.shape)

spotify_scaled.head()
```

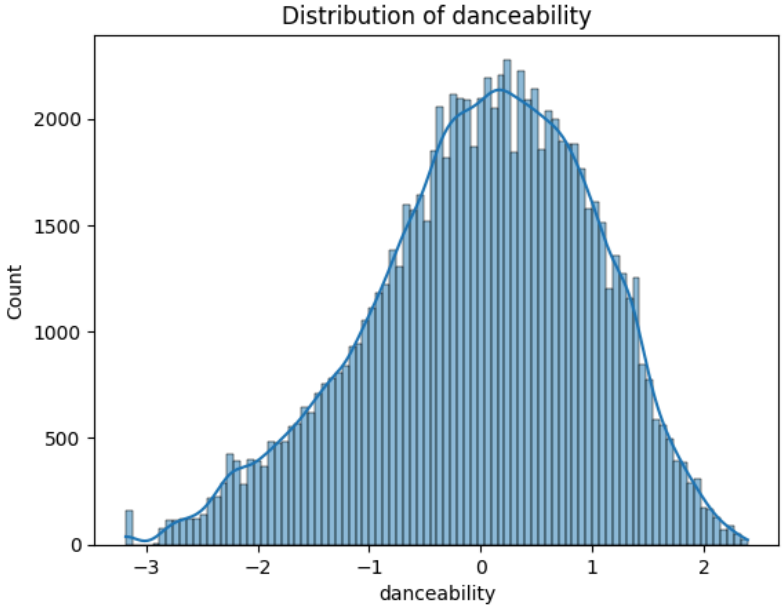
```
Speechiness mean: 1.5336593966718824e-16
Speechiness standard deviation: 1.000005571635916
(89741, 7)
```

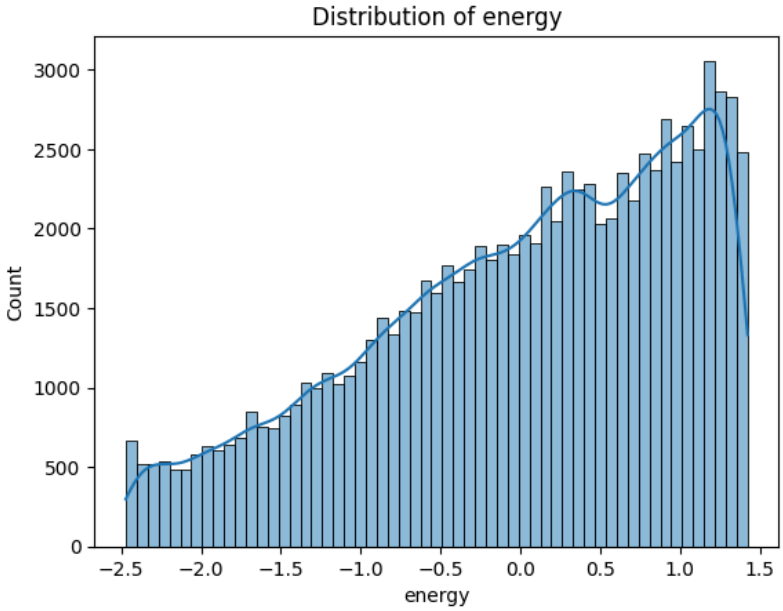
Out[ ]:

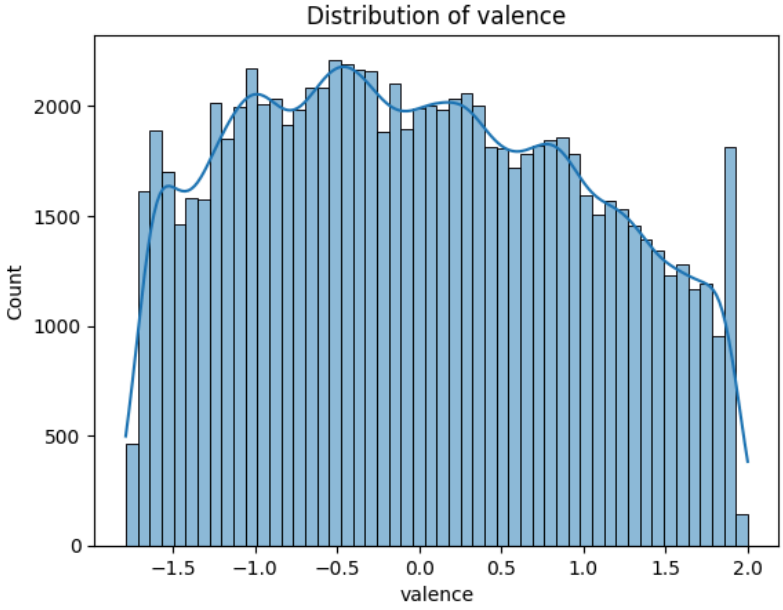
	speechiness	danceability	energy	valence	tempo	loudness	acousticness
0	0.984829	1.968612	-1.015021	-0.142574	-0.598204	-0.257590	-0.746600
1	-0.448831	-1.659208	-0.461638	-0.488763	1.863234	0.217948	0.229697
2	-0.364083	0.700856	-0.290167	-1.375160	-0.068443	-0.912579	-0.966984
3	0.905378	0.661238	0.528216	1.405766	1.316937	0.950310	-0.798031
4	-0.515041	-0.244301	-0.792888	-0.895821	0.262728	-0.978078	-0.967504

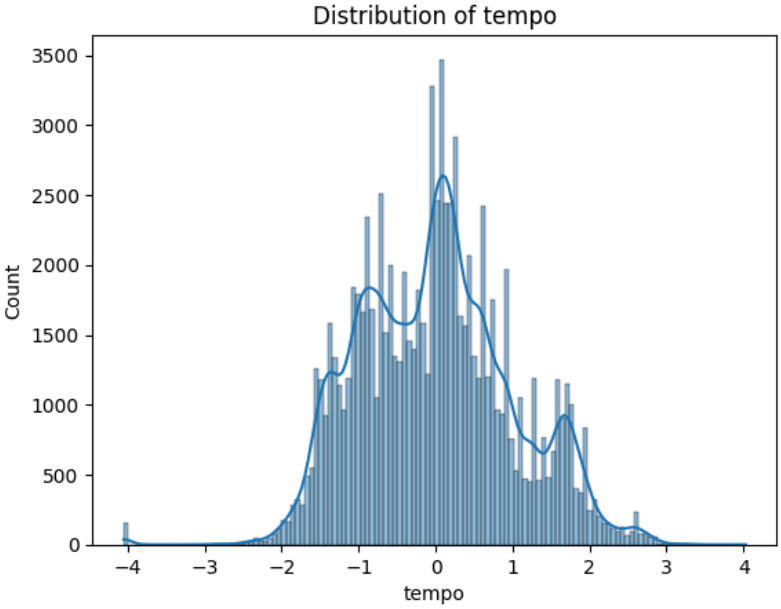
In [ ]:
```
# Visualize distribution of variables
for column in spotify_scaled.columns:
    sns.histplot(spotify_scaled[column], kde=True)
    plt.title(f'Distribution of {column}')
    plt.show()
```

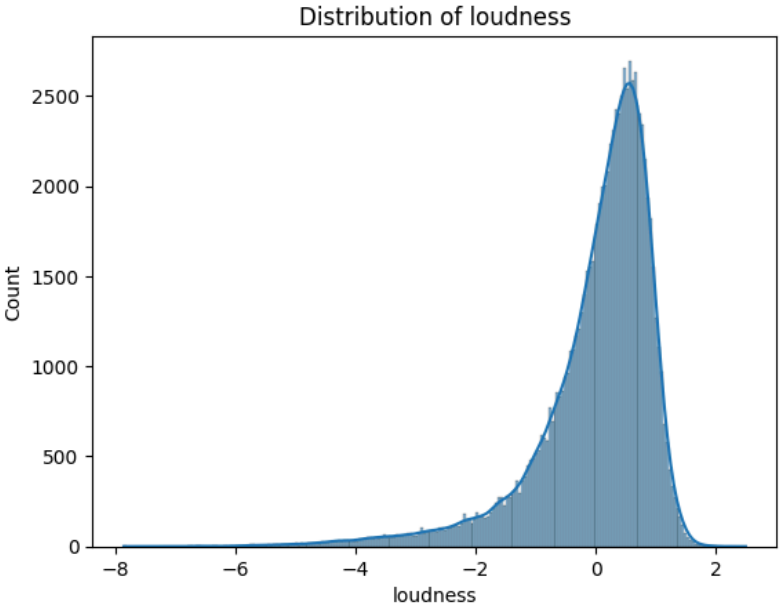


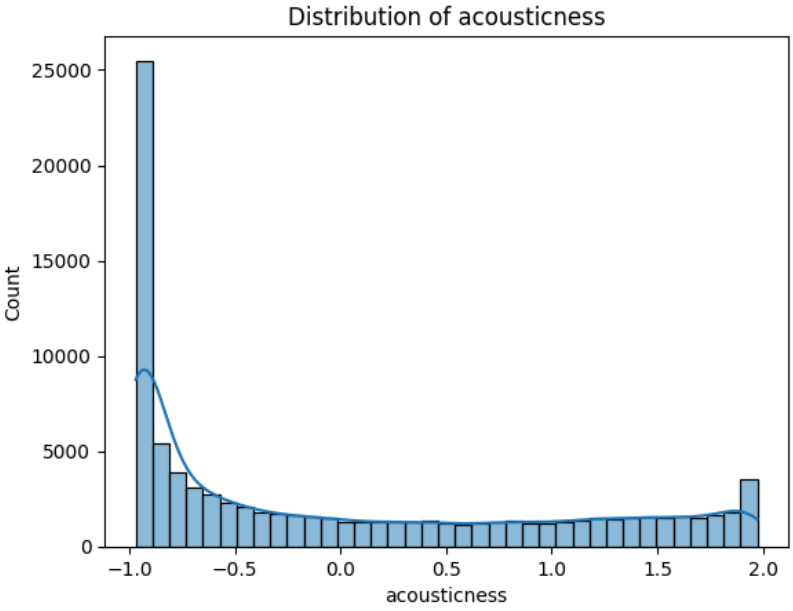

Distribution of danceability
Count
danceability
0
500
1000
1500
2000
−3
−2
−1
0
1
2


Distribution of energy
Count
energy
0
500
1000
1500
2000
2500
3000
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Distribution of valence
Count
valence
0
500
1000
1500
2000
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0







In [ ]:
```
# Perform SVD
pca_U, pca_d, pca_V = np.linalg.svd(spotify_scaled, full_matrices = False)
print(f'Singular values matrix shape: {pca_d.shape}')
print(f'Right singular vectors shape: {pca_V.shape}')
```

```
Singular values matrix shape: (7,)
Right singular vectors shape: (7, 7)
```

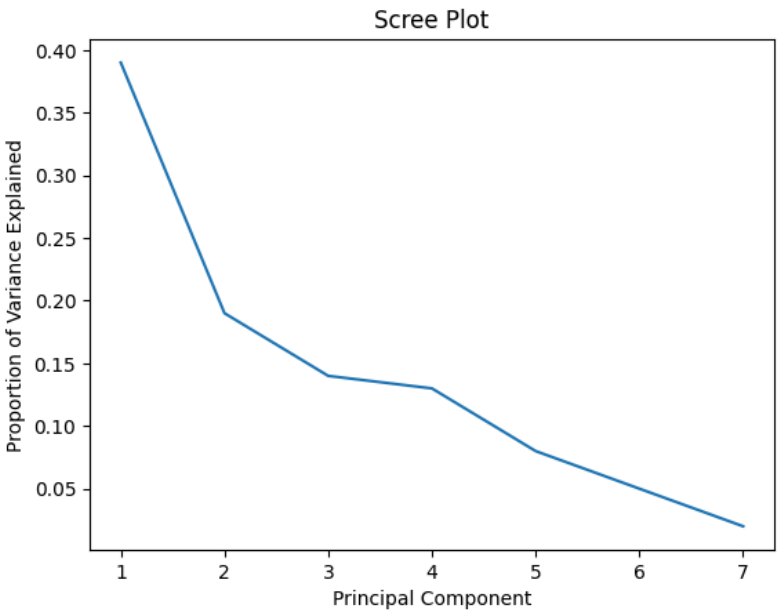
In [ ]:
```
prop_var = np.square(pca_d) / sum(np.square(pca_d))
spotify_pcs = pd.DataFrame(
    {"PC": 1 + np.arange(0, prop_var.shape[0]),
     "variability_explained": prop_var.round(2),
     "cumulative_variability_explained": prop_var.cumsum().round(2)
     })

spotify_pcs
```

Out[ ]:

	PC	variability_explained	cumulative_variability_explained
0	1	0.39	0.39
1	2	0.19	0.58
2	3	0.14	0.72
3	4	0.13	0.85
4	5	0.08	0.92
5	6	0.05	0.98
6	7	0.02	1.00

In [ ]:
```
# Plot scree plot to determine number of PCs to keep
plt.plot(spotify_pcs["PC"], spotify_pcs["variability_explained"])
plt.title('Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Proportion of Variance Explained')
```

Out[ ]:
```
Text(0, 0.5, 'Proportion of Variance Explained')
```



In [ ]:
```
# View PC1
loadings1 = pd.DataFrame(
    {"feature": spotify_scaled.columns,
     "pc1_loading": pca_V[0]
     })
# look at the 10 largest (absolute value) loadings for PC1 but print out the signed value
loadings1.reindex(loadings1["pc1_loading"].abs().sort_values(ascending=False).index) \
    .head(7)
```

Out[ ]:

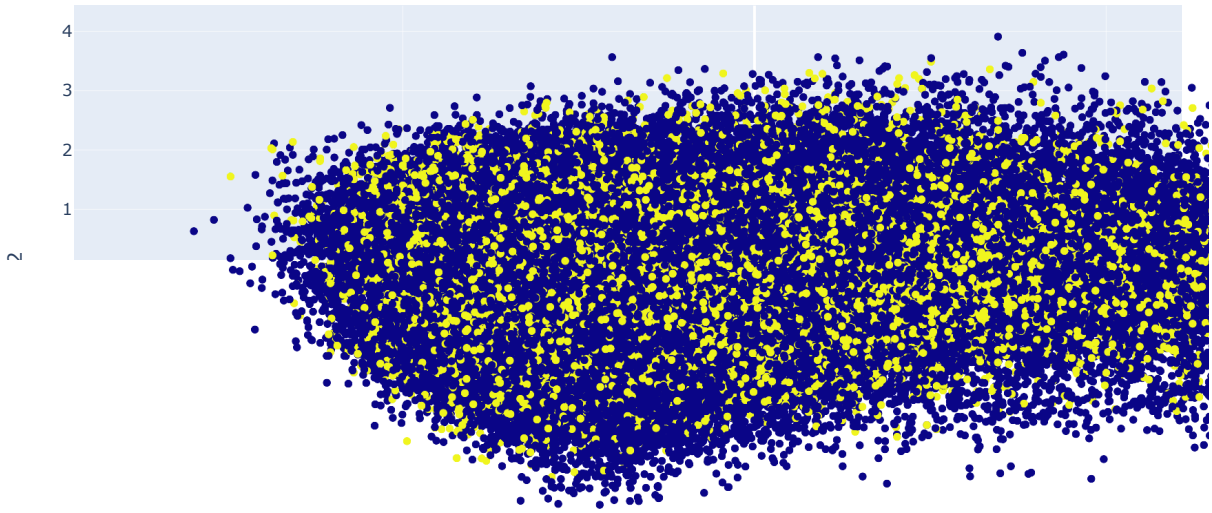
	feature	pc1_loading
2	energy	-0.539456
5	loudness	-0.522157
6	acousticness	0.479310
3	valence	-0.285337
1	danceability	-0.261247
4	tempo	-0.223953
0	speechiness	-0.082331

In [ ]:
```
# View PC2
loadings2 = pd.DataFrame(
    {"feature": spotify_scaled.columns,
     "pc2_loading": pca_V[2]
     })
# look at the 10 largest (absolute value) loadings for PC2 but print out the signed value
loadings2.reindex(loadings2["pc2_loading"].abs().sort_values(ascending=False).index) \
    .head(7)
```

Out[ ]:

	feature	pc2_loading
0	speechiness	-0.958673
3	valence	0.225507
4	tempo	0.106848
2	energy	-0.103887
1	danceability	0.079672
6	acousticness	-0.035195
5	loudness	0.017262

In [ ]:
```
# Project data onto PCs and plot
pca_scaled_spotify = spotify_scaled@pca_V.T
pca_scaled_spotify.columns = ["PC" + str(1+col) for col in pca_scaled_spotify.columns]
pca_scaled_spotify = pd.concat([pd.DataFrame(y_labels), pca_scaled_spotify], axis = 1)
pca_scaled_spotify.rename(columns={0: 'train_labels'}, inplace=True)

px.scatter(pca_scaled_spotify, x="PC1", y="PC2", color = "train_labels",
           hover_name = spotify_scaled.index)
```



In [ ]:
```
random.seed(1)
clustering_sample = pca_scaled_spotify.sample(10000)
clustering_sample.head()
```

Out[ ]:

	train_labels	PC1	PC2	PC3	PC4	PC5	PC6	PC7
74589	1	0.933333	0.439334	0.285651	-0.796895	0.998750	0.633222	-0.366070
33730	0	-0.856757	0.485816	0.568377	-0.493479	1.025636	0.315166	0.446194
24548	0	-1.744007	0.624451	-0.176803	0.838098	-0.093973	-0.352533	0.278110
69912	0	-0.842339	-0.798035	0.660929	1.473501	-0.604658	-0.379879	0.369382
5469	0	2.159465	2.358153	0.619905	3.348939	-0.094010	-1.183646	-0.373883

In [ ]:
```
# Perform hierarchical clustering
hclust = AgglomerativeClustering(metric = 'euclidean', linkage = 'ward',n_clusters = 2)
hclust_labels = hclust.fit(clustering_sample[1:]).labels_
for i in range(6):
```

```
    # print(np.where(body_hclust_labels == i))
    print(f"Number of data points in cluster {i}: {len(np.where(hclust_labels == i)[0])}")

print(f"Silhouette score for hiearchical clustering: {silhouette_score(clustering_sample[1:], hclust_labels)}")
```

```
Number of data points in cluster 0: 6997
Number of data points in cluster 1: 3002
Number of data points in cluster 2: 0
Number of data points in cluster 3: 0
Number of data points in cluster 4: 0
Number of data points in cluster 5: 0
Silhouette score for hiearchical clustering: 0.2504825384427143
```

In [ ]:
```
# Perform kmeans clustering
kmeans = KMeans(n_clusters = 2) # kmeans.labels gets the cluster label for each data point
# Fit_predict computes cluster centers and predicts cluster index for each sample (returns cluster labels for eac
y_kmeans = kmeans.fit_predict(clustering_sample[1:])
kmeans.cluster_centers_ # Big arrays b/c centroid is multidimensional on hundreds of features
kmeans_labels = kmeans.labels_
# Examine the clusters
for i in range(20):
    # print(np.where(body_hclust_labels == i))
    print(f"Number of data points in cluster {i}: {len(np.where(y_kmeans == i)[0])}")

print(f"Silhouette score for K-means: {silhouette_score(clustering_sample[1:], y_kmeans)}")
```

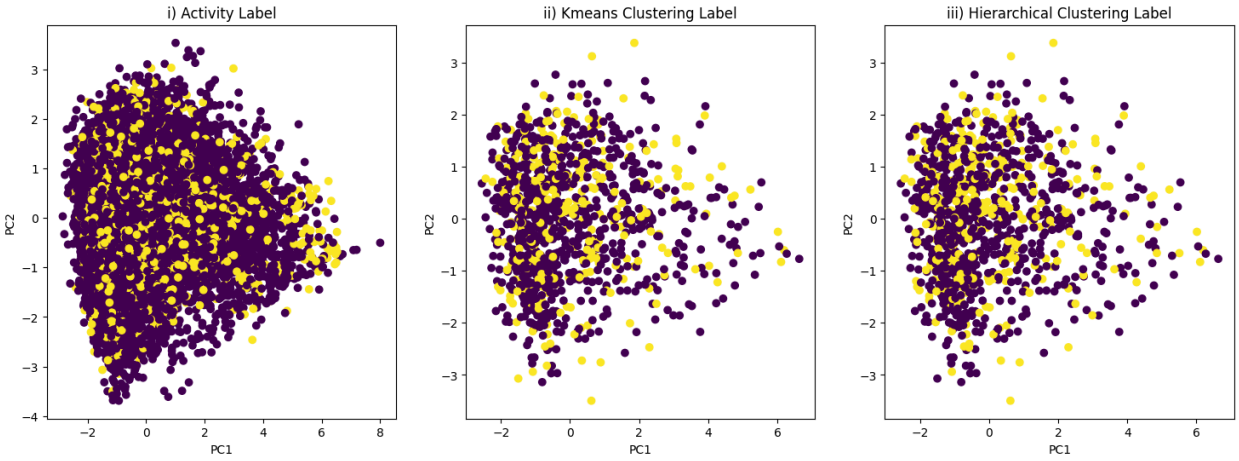
```
Number of data points in cluster 0: 7176
Number of data points in cluster 1: 2823
Number of data points in cluster 2: 0
Number of data points in cluster 3: 0
Number of data points in cluster 4: 0
Number of data points in cluster 5: 0
Number of data points in cluster 6: 0
Number of data points in cluster 7: 0
Number of data points in cluster 8: 0
Number of data points in cluster 9: 0
Number of data points in cluster 10: 0
Number of data points in cluster 11: 0
Number of data points in cluster 12: 0
Number of data points in cluster 13: 0
Number of data points in cluster 14: 0
Number of data points in cluster 15: 0
Number of data points in cluster 16: 0
Number of data points in cluster 17: 0
Number of data points in cluster 18: 0
Number of data points in cluster 19: 0
Silhouette score for K-means: 0.29038817582953835
```

In [ ]:
```
# Calculate rand index
print(f"Rand score for K-means and hierarchical clustering: {rand_score(y_kmeans, hclust_labels)}")
```

```
Rand score for K-means and hierarchical clustering: 0.8542857286328753
```

In [ ]:
```
# Creating a single plot with the colors
clustering_sample = pd.concat([pd.DataFrame(kmeans_labels), clustering_sample], axis = 1)
clustering_sample.rename(columns={0: 'kmeans_labels'}, inplace=True)

clustering_sample = pd.concat([pd.DataFrame(hclust_labels), clustering_sample], axis = 1)
clustering_sample.rename(columns={0: 'hclust_labels'}, inplace=True)

# Plot 1 row, 3 columns
fig, axes = plt.subplots(1, 3, figsize = (18, 6))
scatter1 = axes[0].scatter(x=clustering_sample['PC1'], y=clustering_sample['PC2'], c = clustering_sample['train_la
axes[0].set_title("i) Activity Label")
axes[0].set_xlabel("PC1")
axes[0].set_ylabel("PC2")

scatter1 = axes[1].scatter(x=clustering_sample['PC1'], y=clustering_sample['PC2'], c = clustering_sample['kmeans_
axes[1].set_title("ii) Kmeans Clustering Label")
axes[1].set_xlabel("PC1")
axes[1].set_ylabel("PC2")

scatter1 = axes[2].scatter(x=clustering_sample['PC1'], y=clustering_sample['PC2'], c = clustering_sample['hclust_
axes[2].set_title("iii) Hierarchical Clustering Label")
axes[2].set_xlabel("PC1")
axes[2].set_ylabel("PC2")
```

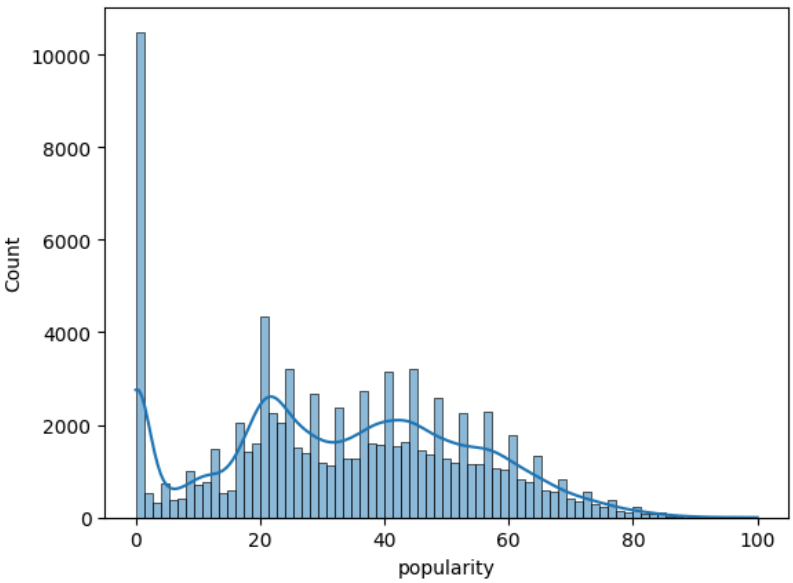
Out[ ]:
```
Text(0, 0.5, 'PC2')
```



8. Neural Networks

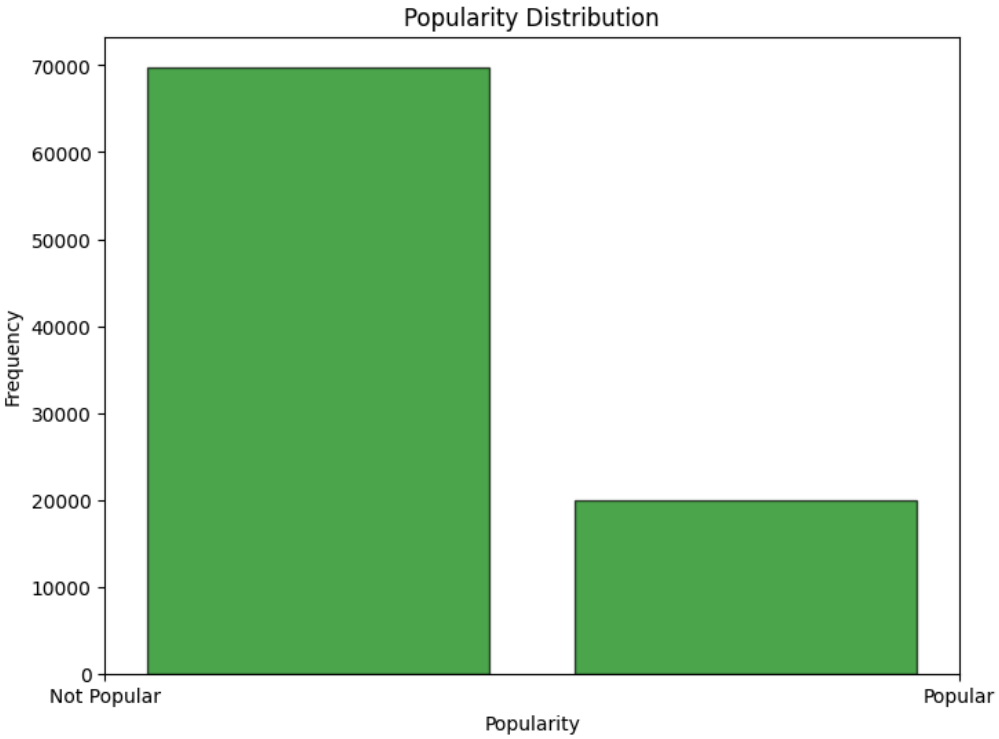
In [ ]:
```
# Load dataset

spotify_clean = spotify_df_cleaned.copy()
```

In [ ]:
```
# Define features and target
features = spotify_clean[["speechiness", "danceability", "energy", "valence", "tempo", "loudness", "acousticness"
target = 'popularity'

#before converting into binary values
sns.histplot(spotify_clean[target], kde= True)
plt.show()
unique_counts = spotify_clean[target].nunique()
print(unique_counts)
```



288

In [ ]:
```
target = (spotify_clean["popularity"] > 50).astype(int)  # set popularity threshold at

# Plot popularity distribution
plt.figure(figsize=(8, 6))
plt.hist(target, bins=2, alpha=0.7, color='green', edgecolor='black', rwidth=0.8)
plt.xticks([0, 1], labels=['Not Popular', 'Popular'])
plt.xlabel("Popularity")
plt.ylabel("Frequency")
plt.title("Popularity Distribution")
plt.show()
```



In [ ]:
```
# Split data
X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.2, random_state=42)
```

In [ ]:
```
# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [ ]:
```
# Convert to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train.values, dtype=torch.float32).view(-1, 1)
y_test = torch.tensor(y_test.values, dtype=torch.float32).view(-1, 1)
```

In [ ]:
```
# Define the neural network
class SpotifyClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(SpotifyClassifier, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
```

HyperParameters

In [ ]:
```
# Hyperparameters
input_dim = X_train.shape[1]
hidden_dim = 16  # Adjust as needed
learning_rate = 0.001
num_epochs = 200
```

Training Neural Network:

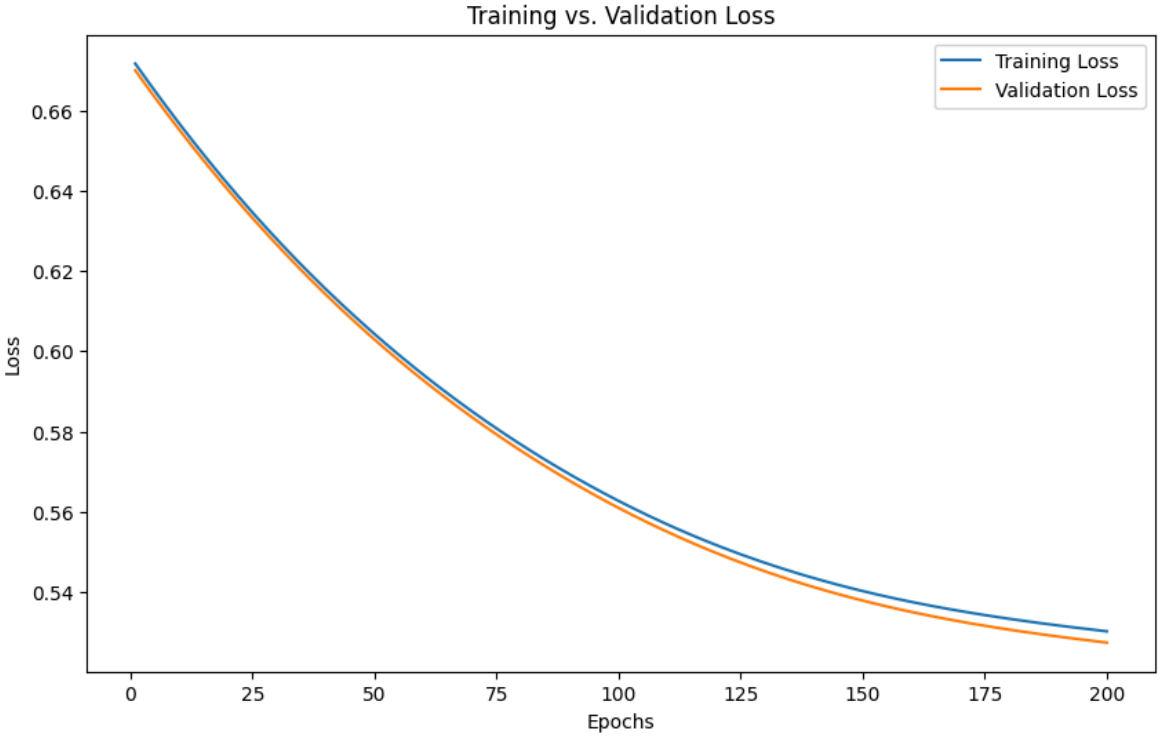
In [ ]:
```
# Initialize model, loss function, and optimizer
model = SpotifyClassifier(input_dim, hidden_dim)
criterion = nn.BCELoss()  # Binary Cross-Entropy Loss
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-5)
```

In [ ]:
```
# Initialize lists to store loss values
train_losses = []
val_losses = []

# Training loop with validation loss tracking
```

```
for epoch in range(num_epochs):
   # Training
   model.train()
   optimizer.zero_grad()
   train_outputs = model(X_train)
   train_loss = criterion(train_outputs, y_train)
   train_loss.backward()
   optimizer.step()


   # Validation
   model.eval()
   with torch.no_grad():
       val_outputs = model(X_test)
       val_loss = criterion(val_outputs, y_test)


   # Store the losses
   train_losses.append(train_loss.item())
   val_losses.append(val_loss.item())


   if (epoch + 1) % 10 == 0:
       print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss.item():.4f}, Val Loss: {val_loss.item():.4

 # Evaluation
model.eval()
with torch.no_grad():
    predictions = model(X_test)
    predictions = (predictions > 0.5).float()
    accuracy = (predictions.eq(y_test).sum() / y_test.shape[0]).item()
    print(f"Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch [10/200], Train Loss: 0.6568, Val Loss: 0.6552
Epoch [20/200], Train Loss: 0.6416, Val Loss: 0.6402
Epoch [30/200], Train Loss: 0.6279, Val Loss: 0.6265
Epoch [40/200], Train Loss: 0.6155, Val Loss: 0.6141
Epoch [50/200], Train Loss: 0.6043, Val Loss: 0.6029
Epoch [60/200], Train Loss: 0.5941, Val Loss: 0.5927
Epoch [70/200], Train Loss: 0.5850, Val Loss: 0.5835
Epoch [80/200], Train Loss: 0.5767, Val Loss: 0.5752
Epoch [90/200], Train Loss: 0.5693, Val Loss: 0.5677
Epoch [100/200], Train Loss: 0.5627, Val Loss: 0.5609
Epoch [110/200], Train Loss: 0.5568, Val Loss: 0.5549
Epoch [120/200], Train Loss: 0.5517, Val Loss: 0.5497
Epoch [130/200], Train Loss: 0.5472, Val Loss: 0.5451
Epoch [140/200], Train Loss: 0.5434, Val Loss: 0.5412
Epoch [150/200], Train Loss: 0.5402, Val Loss: 0.5378
Epoch [160/200], Train Loss: 0.5375, Val Loss: 0.5350
Epoch [170/200], Train Loss: 0.5352, Val Loss: 0.5326
Epoch [180/200], Train Loss: 0.5333, Val Loss: 0.5306
Epoch [190/200], Train Loss: 0.5316, Val Loss: 0.5289
Epoch [200/200], Train Loss: 0.5302, Val Loss: 0.5274
Accuracy: 78.03%
```

In [ ]:
```
# Plot training vs. validation loss
plt.figure(figsize=(10, 6))
plt.plot(range(1, num_epochs + 1), train_losses, label='Training Loss')
plt.plot(range(1, num_epochs + 1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training vs. Validation Loss')
plt.legend()
plt.show()
```


Training vs. Validation Loss
Training Loss
Validation Loss
Loss
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0
25
50
75
100
125
150
175
200
Epochs