MSIN0097 Individual Assignment - XRKQ3

April 3, 2023

Word count: 2085

Data Set:

https://www.kaggle.com/datasets/vicsuperman/prediction-of-music-genre

Notebook:

https://drive.google.com/drive/folders/14IqNfK1ujp7AQl-oe-WB_5W4lDggVLLi?usp=sharing

Table of Contents

1 Introduction

Understanding the factors that drive a song's popularity can help musicians and music companies better understand their audience and make more informed decisions about marketing and distribution strategies (Alexis, 2023). Thus, with this Spotify dataset, this project aims to predict the popularity of songs (dependent variable) based on the other key variables including acousticness, danceability, duration, energy, instrumentalness, liveness, loudness, mode, speechiness, and valence. The primary research question is which factors affect the popularity of a song, and to what extent. Another question is whether music genre influences a song's popularity. By exploring these research questions, I hope to gain insights into the drivers of a song's success and develop a model that accurately predicts song popularity.

To answer the research question traditional machine learning models such as linear regression and random forest will be trained and built on the data, and their performance will be evaluated. This performance will then be compared to the performance of deep learning models (neural networks). The deep learning models will be fine-tuned by adjusting hyperparameters, such as the number of hidden layers or the learning rate, using techniques such as grid search, in order to further improve

their performance. By comparing the performance of different types of models and fine-tuning the best ones, we can identify the most effective approach for modeling the data and making accurate predictions.

2 Data Preparation

[1]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
import tensorflow as tf
from keras.callbacks import EarlyStopping
from keras.layers import Dense, Dropout
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

```
2023-04-01 01:40:45.752667: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 AVX512F AVX512_VNNI FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

[2]:
```
df = pd.read_csv("music_genre 2.csv")
```

2.1 Data Overview

This dataset contains information on various songs from Spotify. It includes information such as the popularity of the song, the acousticness, danceability, duration, energy, instrumentalness, key, liveness, loudness, mode, speechiness, tempo, valence, and genre.

[3]:
```
df.head()
```

[3]:

	instance_id	artist_name	track_name	popularity	\
0	32894.0	Röyksopp	Röyksopp's Night Out	27.0	
1	46652.0	Thievery Corporation	The Shining Path	31.0	
2	30097.0	Dillon Francis	Hurricane	28.0	
3	62177.0	Dubloadz	Nitro	34.0	

```
4      24907.0           What So Not      Divide & Conquer        32.0

   acousticness  danceability  duration_ms  energy  instrumentalness key  \
0       0.00468         0.652         -1.0   0.941           0.79200  A#
1       0.01270         0.622     218293.0   0.890           0.95000   D
2       0.00306         0.620     215613.0   0.755           0.01180  G#
3       0.02540         0.774     166875.0   0.700           0.00253  C#
4       0.00465         0.638     222369.0   0.587           0.90900  F#

   liveness  loudness   mode  speechiness              tempo obtained_date  \
0     0.115    -5.201  Minor       0.0748            100.889         4-Apr
1     0.124    -7.043  Minor       0.0300  115.00200000000001         4-Apr
2     0.534    -4.617  Major       0.0345            127.994         4-Apr
3     0.157    -4.498  Major       0.2390            128.014         4-Apr
4     0.157    -6.266  Major       0.0413            145.036         4-Apr

   valence music_genre
0    0.759  Electronic
1    0.531  Electronic
2    0.333  Electronic
3    0.270  Electronic
4    0.323  Electronic
```

[4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50005 entries, 0 to 50004
Data columns (total 18 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   instance_id       50000 non-null  float64
 1   artist_name       50000 non-null  object
 2   track_name        50000 non-null  object
 3   popularity        50000 non-null  float64
 4   acousticness      50000 non-null  float64
 5   danceability      50000 non-null  float64
 6   duration_ms       50000 non-null  float64
 7   energy            50000 non-null  float64
 8   instrumentalness  50000 non-null  float64
 9   key               50000 non-null  object
 10  liveness          50000 non-null  float64
 11  loudness          50000 non-null  float64
 12  mode              50000 non-null  object
 13  speechiness       50000 non-null  float64
 14  tempo             50000 non-null  object
 15  obtained_date     50000 non-null  object
 16  valence           50000 non-null  float64
```

```
 17  music_genre       50000 non-null  object
dtypes: float64(11), object(7)
memory usage: 6.9+ MB
```

2.2 Data Cleaning

Based on the dataset information, it seems that the columns 'instance_id', 'artist_name', 'track_name', 'obtained_date', and 'tempo' do not contribute to the target variable 'popularity' or any of the predictor variables. 'instance_id' appears to be a unique identifier for each row of data and therefore does not provide any information about the song's popularity. Similarly, 'artist_name' and 'track_name' are simply the names of the artist and track, which are not likely to have a strong relationship with popularity. 'obtained_date' is the date that the data was obtained, which is not likely to be related to the popularity of a song. Tempo was dropped because it is an object data type, meaning that it contains random non-numeric values, making it unsuitable for modeling.

[5]:
```
#dropping irrelevant columns
df = df.drop(['instance_id', 'artist_name', 'track_name', 'obtained_date',␣
 ↪'tempo'], axis=1)
```

[6]:
```
df.head()
```

[6]:

	popularity	acousticness	danceability	duration_ms	energy	\
0	27.0	0.00468	0.652	-1.0	0.941	
1	31.0	0.01270	0.622	218293.0	0.890	
2	28.0	0.00306	0.620	215613.0	0.755	
3	34.0	0.02540	0.774	166875.0	0.700	
4	32.0	0.00465	0.638	222369.0	0.587	

	instrumentalness	key	liveness	loudness	mode	speechiness	valence	\
0	0.79200	A#	0.115	-5.201	Minor	0.0748	0.759	
1	0.95000	D	0.124	-7.043	Minor	0.0300	0.531	
2	0.01180	G#	0.534	-4.617	Major	0.0345	0.333	
3	0.00253	C#	0.157	-4.498	Major	0.2390	0.270	
4	0.90900	F#	0.157	-6.266	Major	0.0413	0.323	

	music_genre
0	Electronic
1	Electronic
2	Electronic
3	Electronic
4	Electronic

[7]:
```
#checking for missing values
print(df.isnull().sum())
```

```
popularity          5
acousticness        5
```

```
danceability        5
duration_ms         5
energy              5
instrumentalness    5
key                 5
liveness            5
loudness            5
mode                5
speechiness         5
valence             5
music_genre         5
dtype: int64
```

[8]:
```
#dropping missing values
df.dropna(how="all",inplace=True)
```

[9]:
```
df.isna().sum()
```

[9]:
```
popularity          0
acousticness        0
danceability        0
duration_ms         0
energy              0
instrumentalness    0
key                 0
liveness            0
loudness            0
mode                0
speechiness         0
valence             0
music_genre         0
dtype: int64
```

[10]:
```
df.head(10)
```

[10]:

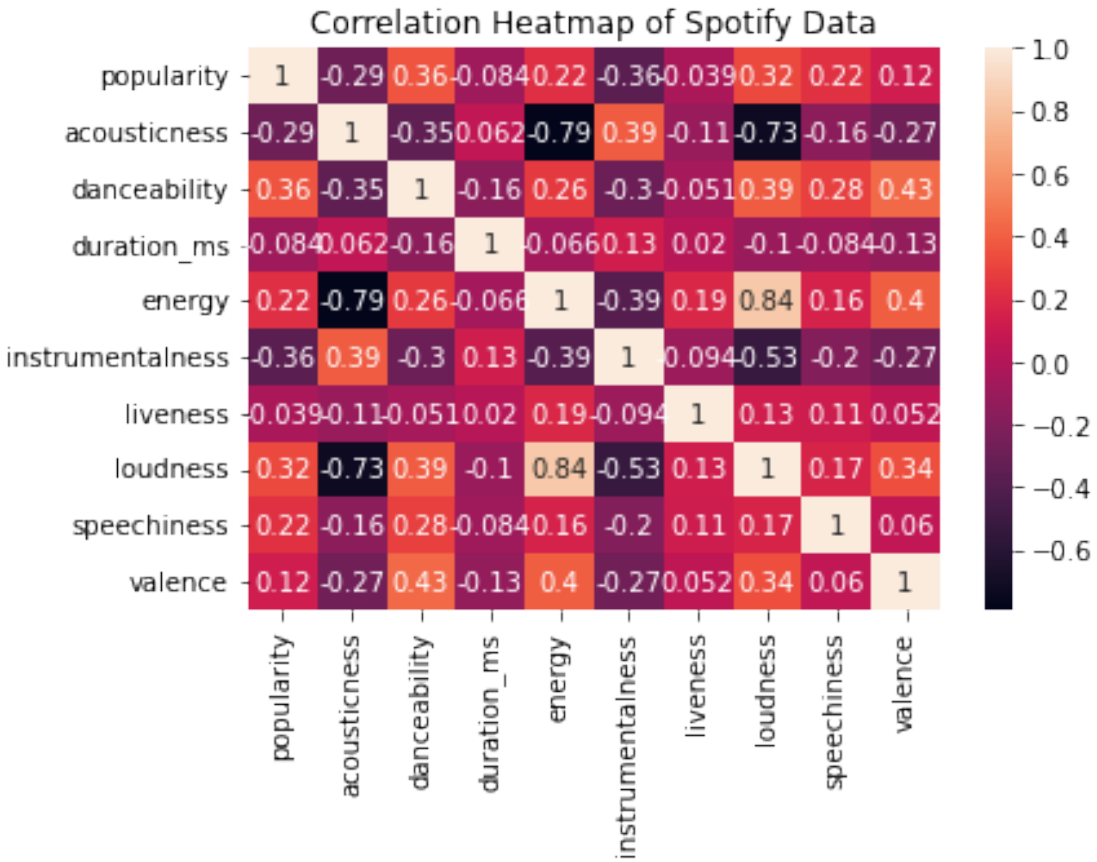
	popularity	acousticness	danceability	duration_ms	energy	\
0	27.0	0.00468	0.652	-1.0	0.941	
1	31.0	0.01270	0.622	218293.0	0.890	
2	28.0	0.00306	0.620	215613.0	0.755	
3	34.0	0.02540	0.774	166875.0	0.700	
4	32.0	0.00465	0.638	222369.0	0.587	
5	47.0	0.00523	0.755	519468.0	0.731	
6	46.0	0.02890	0.572	214408.0	0.803	
7	43.0	0.02970	0.809	416132.0	0.706	
8	39.0	0.00299	0.509	292800.0	0.921	
9	22.0	0.00934	0.578	204800.0	0.731	

```
   instrumentalness key  liveness  loudness   mode  speechiness  valence  \
0          0.792000  A#    0.1150    -5.201  Minor       0.0748    0.759
1          0.950000   D    0.1240    -7.043  Minor       0.0300    0.531
2          0.011800  G#    0.5340    -4.617  Major       0.0345    0.333
3          0.002530  C#    0.1570    -4.498  Major       0.2390    0.270
4          0.909000  F#    0.1570    -6.266  Major       0.0413    0.323
5          0.854000   D    0.2160   -10.517  Minor       0.0412    0.614
6          0.000008   B    0.1060    -4.294  Major       0.3510    0.230
7          0.903000   G    0.0635    -9.339  Minor       0.0484    0.761
8          0.000276   F    0.1780    -3.175  Minor       0.2680    0.273
9          0.011200   A    0.1110    -7.091  Minor       0.1730    0.203

  music_genre
0  Electronic
1  Electronic
2  Electronic
3  Electronic
4  Electronic
5  Electronic
6  Electronic
7  Electronic
8  Electronic
9  Electronic
```

3 Data Exploration and Visualization

To explore the relationships between the different variables in the dataset as well as identify the strength and direction of linear relationships between variables, a correlation matrix was created. This matrix provides insights into which variables may be most important in predicting the outcome variable (popularity).

[11]:
```
# Creating the correlation matrix
plt.plot(figsize=(15,15))
# Creating the heatmap
sns.heatmap(df[['popularity', 'acousticness', 'danceability', 'duration_ms',␣
 ↪'energy', 'instrumentalness', 'liveness', 'loudness', 'speechiness',␣
 ↪'valence']].corr(),annot=True)
plt.title('Correlation Heatmap of Spotify Data')
plt.show()
```



The correlation matrix indicates that there are both positive and negative correlations between different variables. For example, there is a strong positive correlation between energy and loudness (0.84), suggesting that songs with high energy tend to be louder. Additionally, there is a negative correlation between acousticness and energy (-0.79), suggesting that songs that are more acoustic tend to have less energy.

Interesting findings related to the target variable include a negative correlation between acousticness and popularity (-0.29), indicating that more popular songs tend to be less acoustic, as well as a positive correlation between danceability and popularity (0.36), suggesting that more danceable songs tend to be more popular.

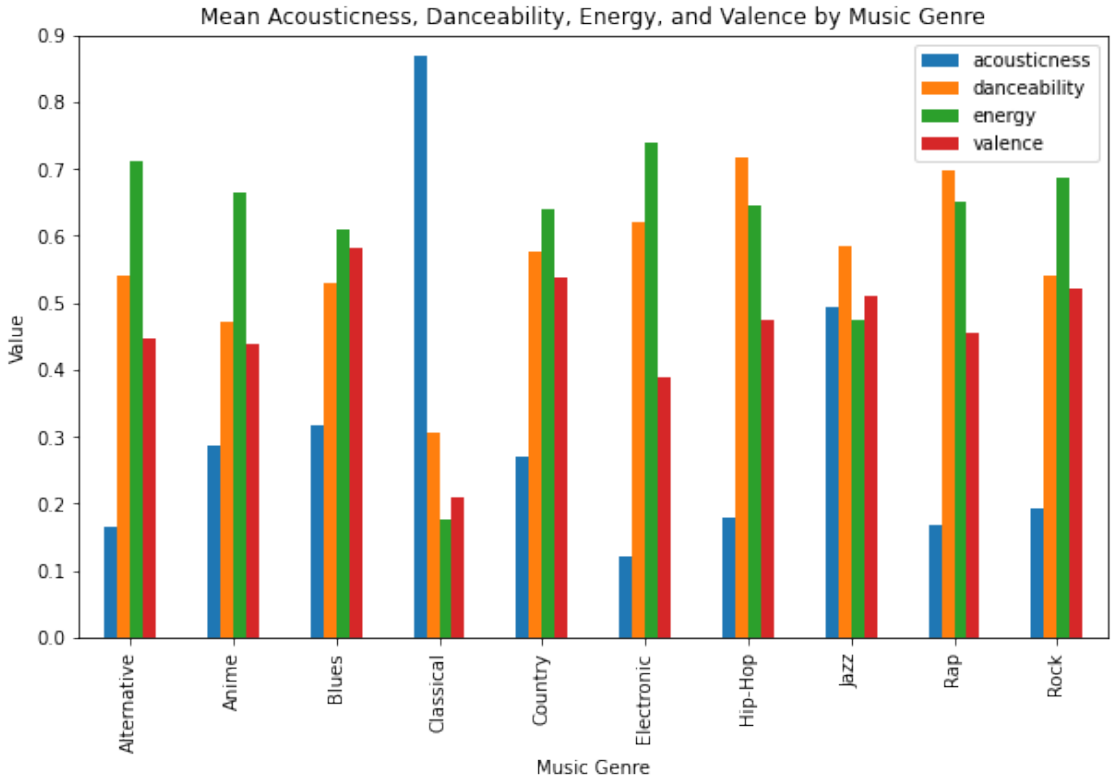
[12]:
```
df.describe()
```

[12]:

	popularity	acousticness	danceability	duration_ms	energy	\
count	50000.000000	50000.000000	50000.000000	5.000000e+04	50000.000000	
mean	44.220420	0.306383	0.558241	2.212526e+05	0.599755	
std	15.542008	0.341340	0.178632	1.286720e+05	0.264559	
min	0.000000	0.000000	0.059600	-1.000000e+00	0.000792	
25%	34.000000	0.020000	0.442000	1.748000e+05	0.433000	
50%	45.000000	0.144000	0.568000	2.192810e+05	0.643000	

```
75%       56.000000      0.552000      0.687000  2.686122e+05      0.815000
max       99.000000      0.996000      0.986000  4.830606e+06      0.999000

       instrumentalness      liveness      loudness   speechiness  \
count      50000.000000  50000.000000  50000.000000  50000.000000
mean           0.181601      0.193896     -9.133761      0.093586
std            0.325409      0.161637      6.162990      0.101373
min            0.000000      0.009670    -47.046000      0.022300
25%            0.000000      0.096900    -10.860000      0.036100
50%            0.000158      0.126000     -7.276500      0.048900
75%            0.155000      0.244000     -5.173000      0.098525
max            0.996000      1.000000      3.744000      0.942000

            valence
count  50000.000000
mean       0.456264
std        0.247119
min        0.000000
25%        0.257000
50%        0.448000
75%        0.648000
max        0.992000
```

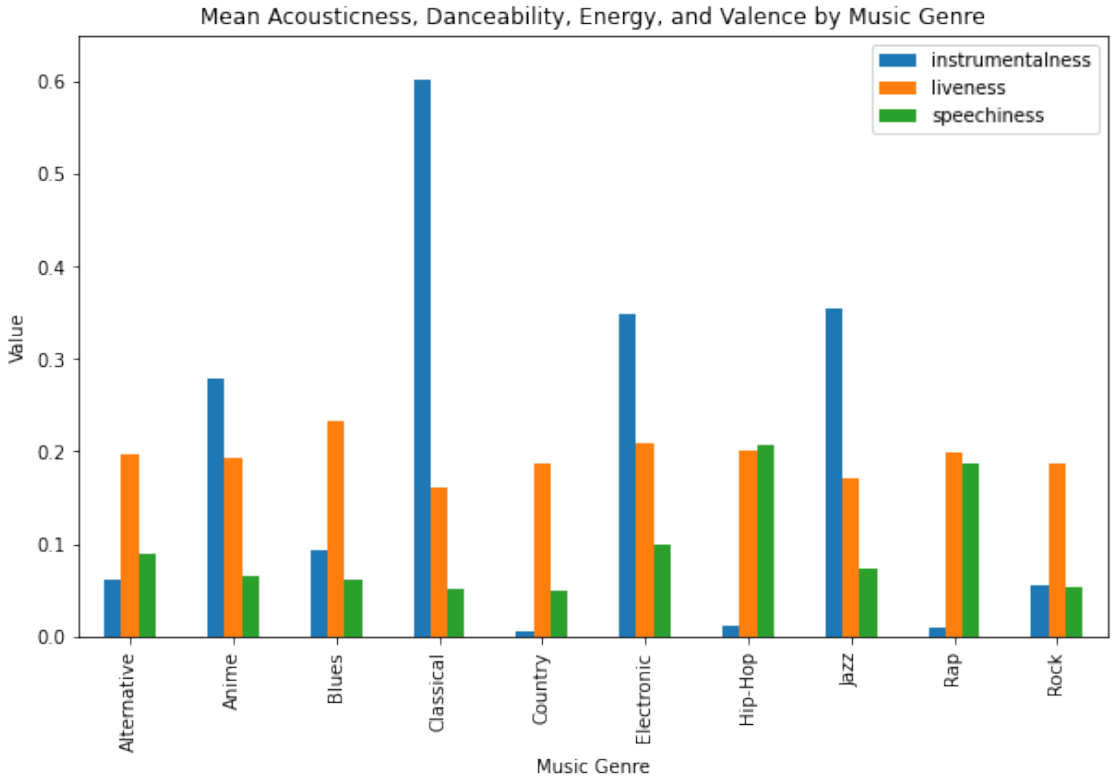
[13]:
```
#Creating a figure
fig, ax = plt.subplots(figsize=(10,6))
# Grouping the DataFrame by music genre and calculate the mean of the columns␣
 ↪'acousticness', 'danceability', 'energy', and 'valence'
# Plotting the resulting DataFrame as a bar plot with the specified axes
df.groupby('music_genre')[['acousticness', 'danceability', 'energy',␣
 ↪'valence']].mean().plot(kind='bar', ax=ax)

# Setting the y-axis limit to be between 0 and 0.9
plt.ylim([0,0.9])

# Setting the x and y labels and title for the plot
plt.xlabel('Music Genre')
plt.ylabel('Value')
plt.title('Mean Acousticness, Danceability, Energy, and Valence by Music Genre')

# showing the plot
plt.show()
```



This plot shows the mean values of four audio features (acousticness, danceability, energy, and valence) for each music genre in the dataset. This plot allows us to compare the average values of these audio features across different music genres and identify any patterns or differences in the audio characteristics of each genre. Danceability and energy seem to be the most important features in many of the genres with acousticness as the least important feature. However, it can be seen that classical deviates from the norm with acousticness as the most important feature.

[14]:
```
# Creating a figure and axis object
fig, ax = plt.subplots(figsize=(10,6))
# Grouping the data by music genre and calculate the mean for instrumentalness,␣
 ↪liveness, and speechiness
df.groupby('music_genre')[['instrumentalness', 'liveness', 'speechiness']].
 ↪mean().plot(kind='bar', ax=ax)

# setting y-axis limit
plt.ylim([0,0.65])

# setting labels and title
plt.xlabel('Music Genre')
plt.ylabel('Value')
plt.title('Mean Acousticness, Danceability, Energy, and Valence by Music Genre')
```

```
# show the plot
plt.show()
```



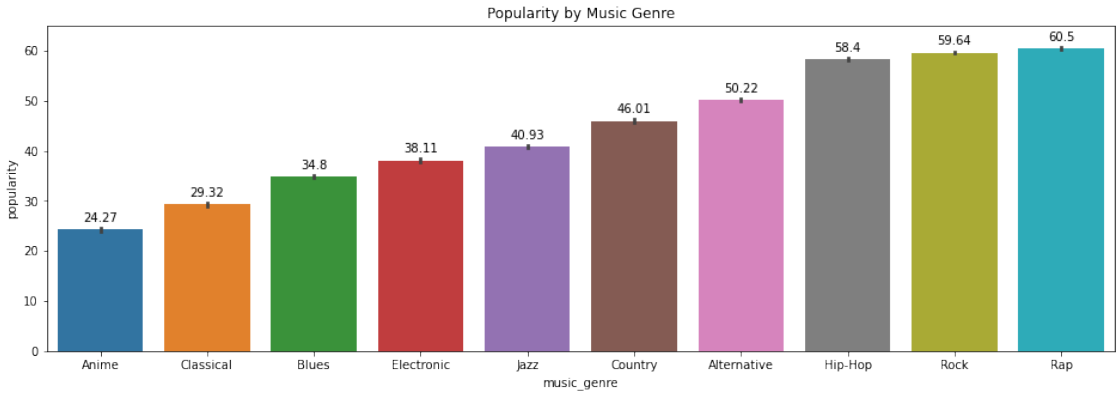
This plot shows the mean values of instrumentalness, liveness, and speechiness for each music genre in the dataset. It is a bar chart that compares the different genres based on these three variables.

From this plot, we can see that the instrumentalness tends to be higher for the classical and electronic music genres, while the liveness tends to be higher for the rock, alternative, blues and country genres. Speechiness tends to be higher for rap and hip-hop genres compared to other genres.

This plot helps us understand the differences in these variables across different music genres and provides insights into the characteristics that define each genre.

[15]:
```
# Creating figure object
plt.figure(figsize=[16, 5])
# Creating a bar plot showing the mean popularity for each music genre, sorted␣
 ↪by popularity from lowest to highest
sns.barplot(x='music_genre', y='popularity', data=df, order=df.
 ↪groupby('music_genre')['popularity'].mean().sort_values().index)
# Setting plot title
plt.title('Popularity by Music Genre')
# Setting the limit of the y-axis from 0 to 65
```

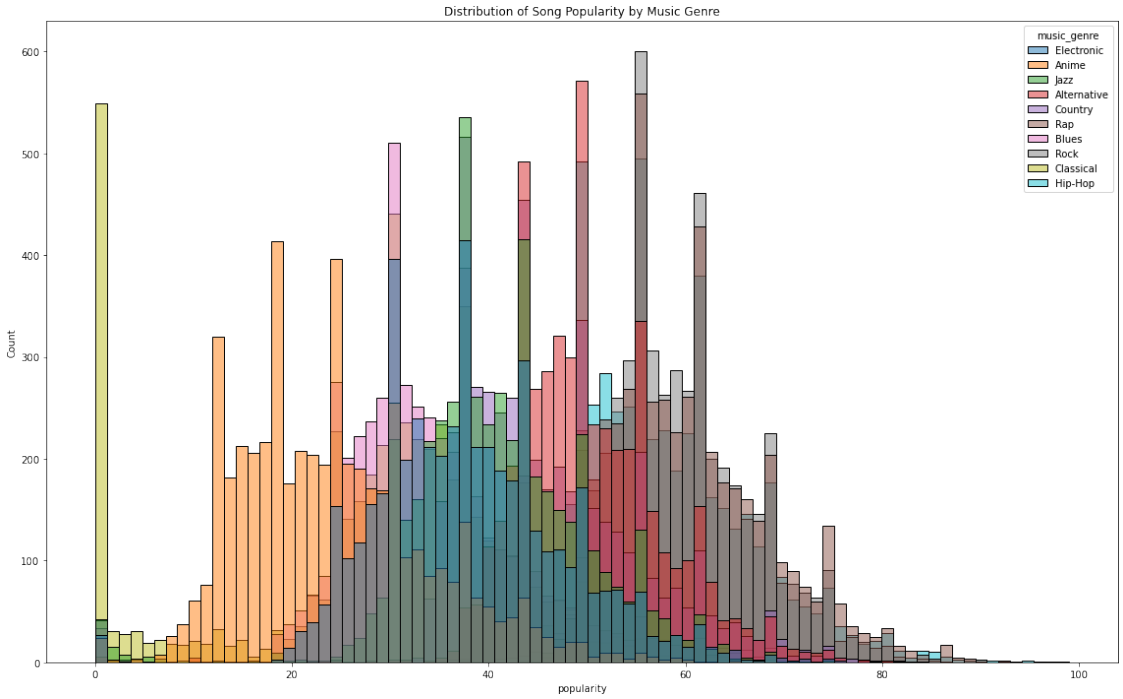
```
plt.ylim([0,65])
# Add values to bars
# iterating through each row of the grouped dataframe sorted by mean popularity␣
 ↪and reset the index
# adding the value of the mean popularity to the top of each bar
for index, row in df.groupby('music_genre')['popularity'].mean().sort_values().
 ↪reset_index().iterrows():
    plt.text(index, row['popularity'] + 1.5, str(round(row['popularity'], 2)),␣
 ↪ha='center')
```



This plot shows the average popularity of songs by music genre. Each bar represents a different music genre, and the height of the bar represents the average popularity of songs in that genre. The popularity of different genres can be compared the most popular or least popular genres can be identified.

As represented by the graph, the most popular genre is rap with rock and hiphop as close seconds. The least popular genres are anime and classical.

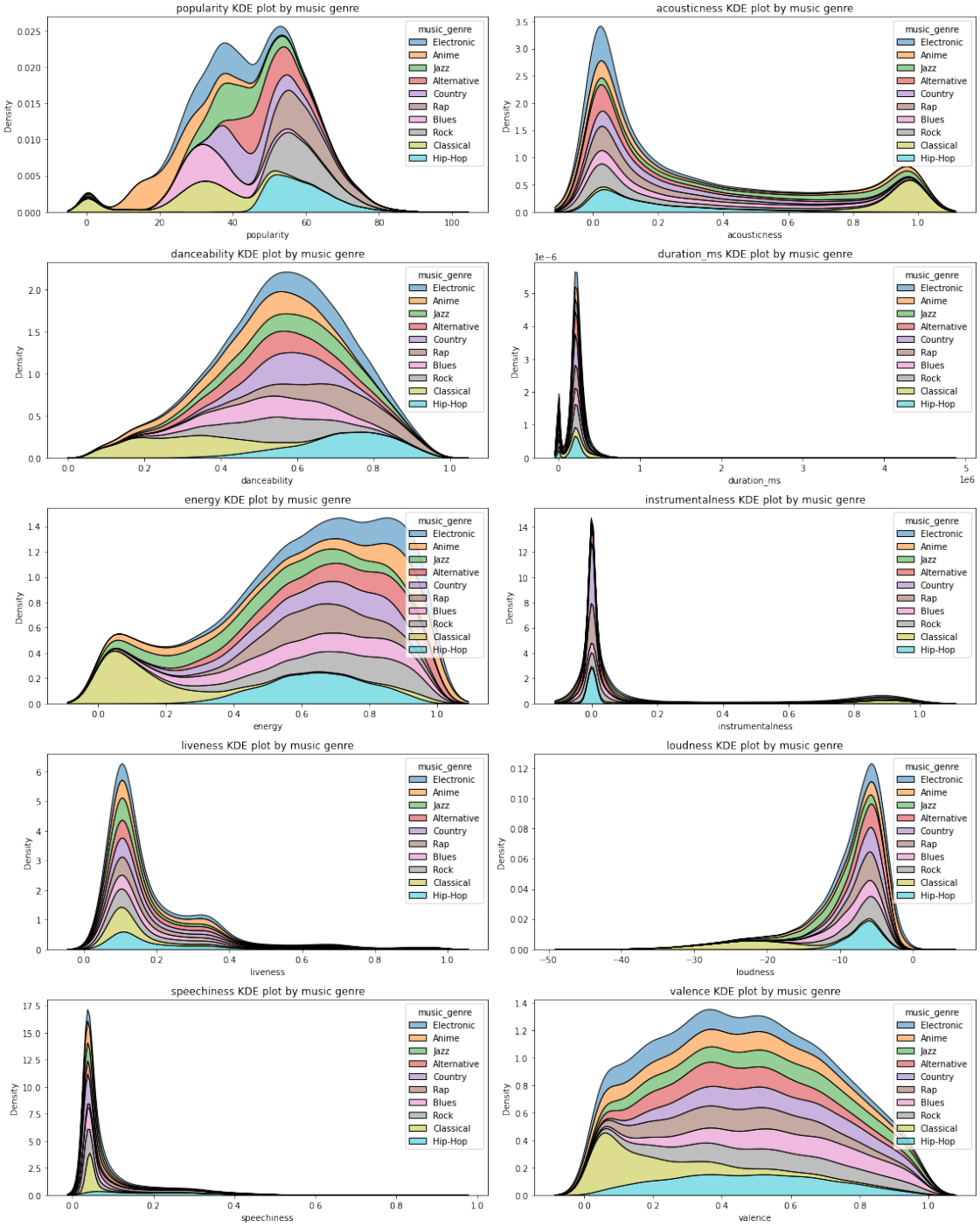
[16]:
```
# Setting the figure size
plt.figure(figsize=(16, 10))
# Creating histogram plot with different color by music genre
sns.histplot(data=df, x='popularity', hue='music_genre')
# Setting the title and adjusting the spacing
plt.title('Distribution of Song Popularity by Music Genre')
plt.tight_layout()
plt.show()
```



The plot shows the distribution of song popularity across different music genres. The plot is split by music genre, with each genre having a different color. This plot allows us to see how the popularity scores are distributed within each genre and if there are any differences in the distribution between genres. It also provides insight to the overall popularity level of each genre. It can be seen which genres tend to have more songs with high popularity scores. For example, rock and hiphop show a higher distribution of song popularity. Whereas, classical shows a skewed distribution, where it is mostly unpopular.

[17]:
```
# Creating a list of the variables to plot
variables = ['popularity', 'acousticness', 'danceability', 'duration_ms',␣
 ↪'energy',
             'instrumentalness', 'liveness', 'loudness', 'speechiness',␣
 ↪'valence']

# Creating a figure with subplots
fig, axs = plt.subplots(nrows=5, ncols=2, figsize=(16, 20))

# Looping through each variable and plot a KDE plot for each music genre
for i, var in enumerate(variables):
    # Determining the subplot location based on the index
    row = i // 2
    col = i % 2
    # Creating the KDE plot using Seaborn
    sns.kdeplot(data=df, x=var, hue='music_genre', fill=True, alpha=0.5,␣
 ↪multiple='stack', ax=axs[row, col])
```

```
    # Setting the title of the subplot
    axs[row, col].set_title(f'{var} KDE plot by music genre')
    # Setting the x and y label for the subplot
    axs[row, col].set_xlabel(var)
    axs[row, col].set_ylabel('Density')

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



This plot shows the kernel density estimate (KDE) of various audio features of songs (popularity, acousticness, danceability, duration_ms, energy, instrumentalness, liveness, loudness, speechiness, and valence) grouped by music genre. Here the the distribution of each variable can be visualized. In this case, it allows us to see how each audio feature is distributed across different music genres. The stacked KDE plots in each subplot show the distribution of the audio feature for each music

genre, with different colors representing different genres. Thus identifying which audio features are most important in distinguishing different music genres.

From the KDE plots, we can observe that there are certain features such as speechiness , loudness, liveness, instrumentalness and duration that are significantly skewed. Furthermore, we can observe that some features have a relatively high degree of skewness for certain genres, for instance, in valence, most genres seem to follow the normal distribution except for classical music, which is left skewed, indicating that the valence of this genre is typically low. These insights are key to build appropriate regressive models, where highly skewed features can be understood as noise by the highly sensitive regression algorithms.

4 Data Pre-Processing

The data is split into input features (X) and output feature (y), where 'popularity' is the target variable that I am trying to predict. Then, the data is split into training and testing sets using the train_test_split() function from scikit-learn, where 20% of the data is reserved for testing. The training set is further split into training and validation sets with a 80-20% ratio. This is done to evaluate the performance of the model on data that it has not been trained on, and to tune the hyperparameters of the model using the validation set.

[18]:
```
# Splitting the data into input (X) and output (y) features
X = df.drop('popularity', axis=1)
y = df['popularity']

# Splitting the data into training and testing sets
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.
 ↪2, random_state=42)

# Splitting the training set into training and validation sets
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full,␣
 ↪y_train_full, test_size=0.2, random_state=42)

# Printing the shape of the data
print(X_train.shape)
print(y_train.shape)
print(X_valid.shape)
print(y_valid.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(32000, 12)
(32000,)
(8000, 12)
(8000,)
(10000, 12)
(10000,)
```

[19]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 50000 entries, 0 to 50004
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   popularity        50000 non-null  float64
 1   acousticness      50000 non-null  float64
 2   danceability      50000 non-null  float64
 3   duration_ms       50000 non-null  float64
 4   energy            50000 non-null  float64
 5   instrumentalness  50000 non-null  float64
 6   key               50000 non-null  object
 7   liveness          50000 non-null  float64
 8   loudness          50000 non-null  float64
 9   mode              50000 non-null  object
 10  speechiness       50000 non-null  float64
 11  valence           50000 non-null  float64
 12  music_genre       50000 non-null  object
dtypes: float64(10), object(3)
memory usage: 5.3+ MB
```

Encoding and Scaling the Data:

To prepare the data for modeling, two pre-processing steps were performed: encoding categorical features and scaling numerical features.

The categorical features "key", "mode", and "music_genre" have string values that need to be converted to numerical values in order for the model to process them. One-hot encoding is used to convert these categorical features into binary values (0 or 1) for each category. This creates new columns for each category in the data frame.

The numerical features are then scaled using the StandardScaler function. This transforms the numerical features so that they have a mean of 0 and a standard deviation of 1. This normalization process helps to prevent the model from being dominated by features with larger numerical values, which is important especially when feeding the data into neural networks.

[20]:
```
#Encoding categorical features with one hot encoding
#Initializing one-hot encoder object
encoder = OneHotEncoder(handle_unknown='ignore')
# Defining the list of categorical features
cat = ['key','mode', 'music_genre']
# Encoding the categorical features of the training set using the one-hot␣
 ↪encoder object, and saving the transformed data as 'X_train_ohe'
X_train_ohe = encoder.fit_transform(X_train[cat])
# Encoding the categorical features of the validation set using the previously␣
 ↪fitted one-hot encoder object, and saving the transformed data as␣
 ↪'X_valid_ohe'
X_valid_ohe = encoder.transform(X_valid[cat])
```

```
# Encoding the categorical features of the testing set using the previously␣
 ↪fitted one-hot encoder object, and saving the transformed data as␣
 ↪'X_test_ohe'
X_test_ohe = encoder.transform(X_test[cat])

#Scaling numerical features
# Defining the list of numerical features to be scaled
num = ['acousticness', 'danceability', 'duration_ms', 'energy',␣
 ↪'instrumentalness', 'liveness', 'loudness', 'speechiness', 'valence']
# Initializing a StandardScaler object which will be used to scale the␣
 ↪numerical features in the data
scaler = StandardScaler()
# Scaling the numerical features of the training set using the StandardScaler␣
 ↪object, and saving the transformed data as 'X_train_scale'
X_train_scale = scaler.fit_transform(X_train[num])
# Scaling the numerical features of the validation set using the previously␣
 ↪fitted StandardScaler object, and saving the transformed data as␣
 ↪'X_valid_scale'
X_valid_scale = scaler.transform(X_valid[num])
# Scaling the numerical features of the testing set using the previously fitted␣
 ↪StandardScaler object, and saving the transformed data as 'X_test_scale'
X_test_scale = scaler.transform(X_test[num])
```

Concatenating the encoded categorical features and scaled numerical features to create a new dataframe that can be used for modeling.

[21]:
```
#Concatenating the encoded categorical features and scaled scaleerical features
# Obtaining the names of the encoded categorical features for the training,␣
 ↪validation and test set
X_train_ohe_names = encoder.get_feature_names_out(cat)
X_valid_ohe_names = encoder.get_feature_names_out(cat)
X_test_ohe_names = encoder.get_feature_names_out(cat)
# Concatenating the one-hot encoded categorical features and the scaled␣
 ↪numerical features for the training set, validation set, and testing set
X_train = np.concatenate((X_train_ohe.toarray(), X_train_scale), axis=1)
X_valid = np.concatenate((X_valid_ohe.toarray(), X_valid_scale), axis=1)
X_test = np.concatenate((X_test_ohe.toarray(), X_test_scale), axis=1)
# Creating a DataFrame from the concatenated features for the training set,␣
 ↪validation set and test set.
# Assigning the feature names as column names
X_train = pd.DataFrame(X_train, columns=list(X_train_ohe_names)+num)
X_valid = pd.DataFrame(X_valid, columns=list(X_valid_ohe_names)+num)
X_test = pd.DataFrame(X_test, columns=list(X_test_ohe_names)+num)
# Resetting the index of the training set DataFrame, validation set DataFrame␣
 ↪and testing set .
X_train = X_train.reset_index(drop=True)
```

```
X_valid = X_valid.reset_index(drop=True)
X_test = X_test.reset_index(drop=True)
```

[22]:
```
print(X_train.shape)
print(y_train.shape)
print(X_valid.shape)
print(y_valid.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(32000, 33)
(32000,)
(8000, 33)
(8000,)
(10000, 33)
(10000,)
```

[23]:
```
df.describe()
```

[23]:

	popularity	acousticness	danceability	duration_ms	energy	\
count	50000.000000	50000.000000	50000.000000	5.000000e+04	50000.000000	
mean	44.220420	0.306383	0.558241	2.212526e+05	0.599755	
std	15.542008	0.341340	0.178632	1.286720e+05	0.264559	
min	0.000000	0.000000	0.059600	-1.000000e+00	0.000792	
25%	34.000000	0.020000	0.442000	1.748000e+05	0.433000	
50%	45.000000	0.144000	0.568000	2.192810e+05	0.643000	
75%	56.000000	0.552000	0.687000	2.686122e+05	0.815000	
max	99.000000	0.996000	0.986000	4.830606e+06	0.999000	

	instrumentalness	liveness	loudness	speechiness	\
count	50000.000000	50000.000000	50000.000000	50000.000000	
mean	0.181601	0.193896	-9.133761	0.093586	
std	0.325409	0.161637	6.162990	0.101373	
min	0.000000	0.009670	-47.046000	0.022300	
25%	0.000000	0.096900	-10.860000	0.036100	
50%	0.000158	0.126000	-7.276500	0.048900	
75%	0.155000	0.244000	-5.173000	0.098525	
max	0.996000	1.000000	3.744000	0.942000	

	valence
count	50000.000000
mean	0.456264
std	0.247119
min	0.000000
25%	0.257000
50%	0.448000
75%	0.648000

```
max         0.992000
```

5 Machine Learning Models

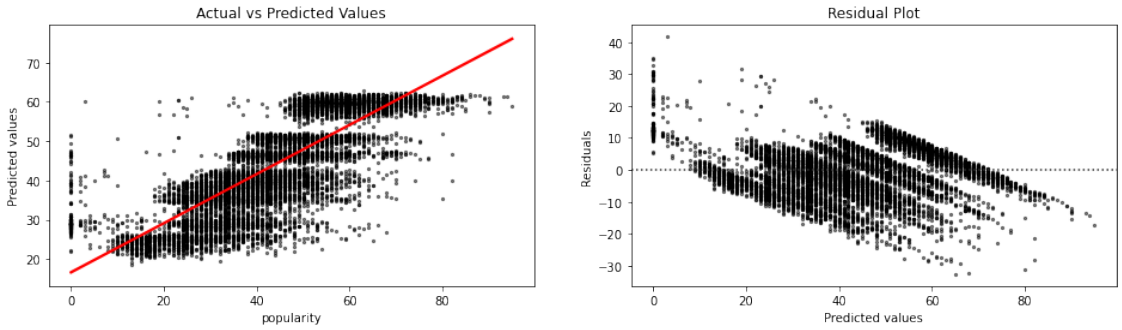
5.1 Linear Regression

To understand the relationship between the different features and how they affect the popularity of a song, a linear regression was performed. Linear regression helped to model the relationship between the dependent variable (popularity) and the independent variables by fitting a linear equation to the data that minimized the difference between the predicted and actual values of popularity. Thus, identifying the most important features that influence the popularity of a song and creating a model that can predict the popularity of songs.

[26]:
```
# Create the model object
model = LinearRegression()

# Fit the model on the training data
model.fit(X_train, y_train)

# Predict on the training and testing sets
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.2f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
```

```
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
axs[1].set_title('Residual Plot')

# Show the plot
plt.show()
```

```
Training set MAE: 7.14
Training set MSE: 89.73
Training set R^2: 0.628501
Testing set MAE: 7.16
Testing set MSE: 91.68
Testing set R^2: 0.625678
```



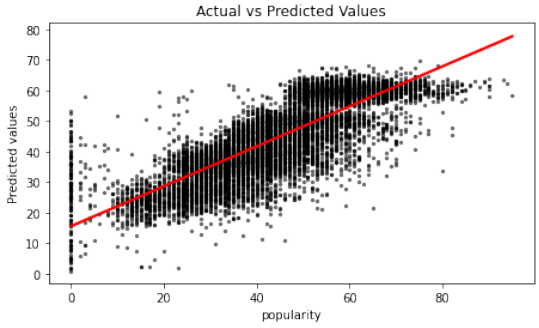
The results of the linear regression show that the model is performing decently well. The training set score of 0.628 indicates that the model is able to explain around 62.85% of the variance in the training set, while the testing set score of 0.625 indicates that the model is able to explain around 62.57% of the variance in the test set. Therefore, the model is making reasonable predictions on new, unseen data. However, the testing set score is slightly lower than the training set score suggesting that the model may be slightly overfitting to the training data and could potentially be improved through regularization or other techniques.

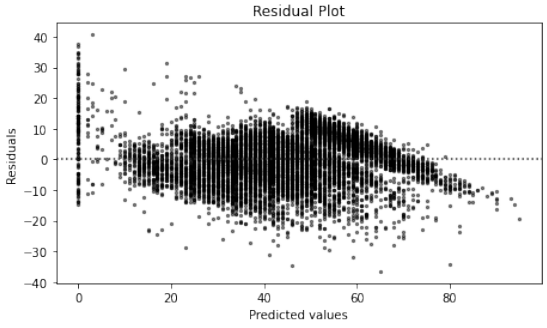
5.2 Random Forest Regressor

A random forest regressor was performed to predict the popularity of songs because it is effective for regression tasks and can handle both numerical and categorical features. The model created a forest of decision trees, where each tree is trained on a random subset of the data and a random subset of the features. The model then combined the predictions of all the trees to make the final prediction.

[43]:
```
# Create the model object
model = RandomForestRegressor(n_estimators=50, random_state=42)

# Fit the model on the training data
model.fit(X_train, y_train)

# Predict on the training and testing sets
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.2f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
axs[1].set_title('Residual Plot')

# Show the plot
plt.show()
```

```
Training set MAE: 2.66
Training set MSE: 12.58
Training set R^2: 0.947914
Testing set MAE: 7.06
Testing set MSE: 87.94
Testing set R^2: 0.640921
```





The training set score of 0.95 indicates that the model fits well to the training data, and it can explain 95% of the variance in the training set. However, the testing set score of 0.64 is lower than the training set score, indicating that the model may be overfitting the training data and not generalizing well to new data. Meaning that the model did not perform as well on new data as it did on the training data.

6 Deep Learning Models

6.1 Neural Network 1

This neural network was defined with two dense layers, where the first layer had 8 neurons and used the ReLU activation function. The ReLU activation function is used to avoid the vanishing gradient problem and speed up the convergence of the optimization algorithm. The second layer had a single neuron since a single output variable is being predicted. No activation function was used for the second layer because in a regression problem we want the output to be an unbounded continuous value, and using an activation function like ReLU or sigmoid would result in a bounded output. Therefore, without an activation function in the output layer, the model can output any real number as the prediction.

The model was then compiled with the mean squared error loss function and the Adam optimizer. The MSE penalizes the difference between the predicted and actual values. The Adam optimizer, was used to adapt the learning rate during training to improve convergence.

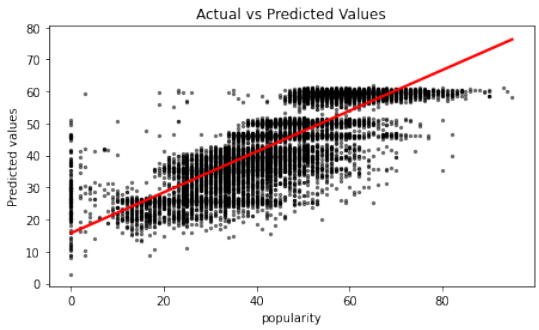
To fit the model, it was trained on the training data for 50 epochs with a batch size of 32. 50 epochs was chosen to increase the training and batch size of 32 was chosen to strike a balance between training speed and stability. Training the model was done to update its parameters (weights and biases) based on the training data to minimize the loss function. Using the validation set, allowed for monitoring the performance of the model on data that it hasn't seen before.

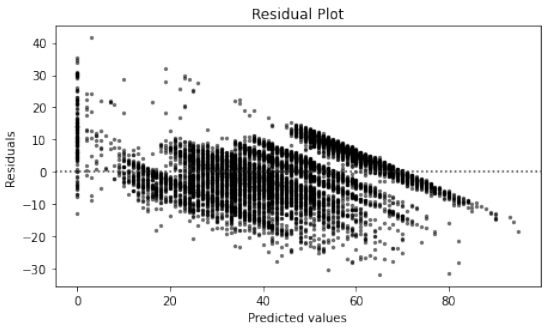
[28]:
```
# Define the model
model = Sequential()
model.add(Dense(8, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(1))

# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam')

# Fit the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32,␣
 ↪validation_data=(X_valid, y_valid), verbose=0)

# Predict on the training and testing sets
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.2f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
```

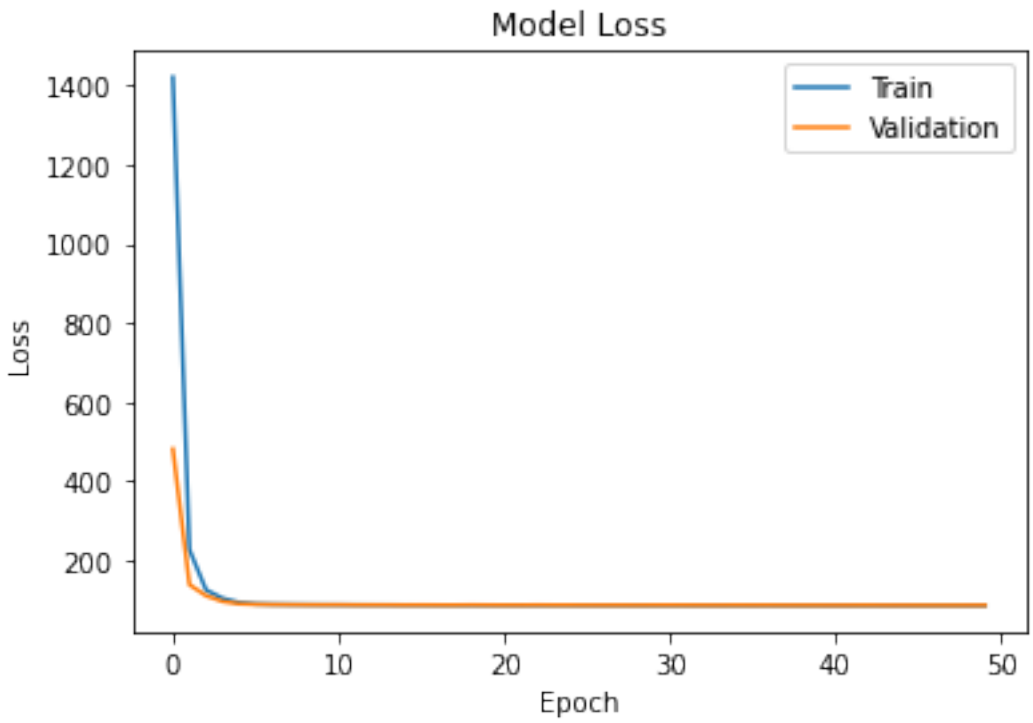
```
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
axs[1].set_title('Residual Plot')

# Show the plot
plt.show()

# Plot the training and validation loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
2023-04-01 01:43:07.834318: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 AVX512F AVX512_VNNI FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

```
1000/1000 [==============================] - 1s 1ms/step
313/313 [==============================] - 0s 1ms/step
Training set MAE: 7.01
Training set MSE: 85.43
Training set R^2: 0.646283
Testing set MAE: 7.02
Testing set MSE: 87.88
Testing set R^2: 0.641169
```







After training, the performance of the model was evaluated using various metrics. Mean absolute error (MAE), mean squared error (MSE), and R-squared to evaluate the model's performance on the training and testing sets. R-squared measures the proportion of the variance in the dependent variable that is predictable from the independent variables. In this case, the R-squared value of 0.64 indicates that the model can explain 64% of the variation in popularity using the given input features. The results of this model suggest that it can predict the popularity of a song with an error of approximately 7. This level of error is reasonable given the range of popularity values and the complexity of the problem. However, there may be additional factors not accounted for in the data that affect the popularity of a song, leading to the model's errors.

6.2 Neural Network 2: Adding an extra layer and more neurons

In the second neural network, an additional layer with more neurons was added to the first neural network. This was done to increase the model complexity, in the hopes that it would improve the performance of the model, especially given that the r2 score of the first neural network on the training set was worse than that of the random forest model.

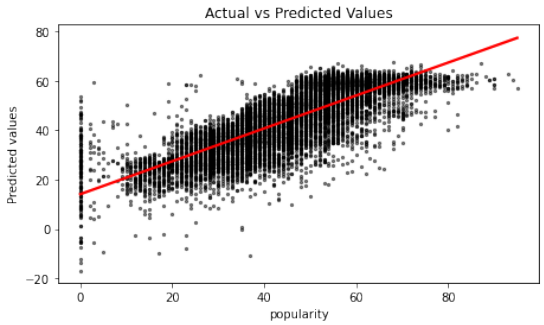
The first hidden layer of the model had 256 neurons, and the second had 128 neurons. The activation function used in the first and second layer was ReLU. ReLU is a widely used activation function in deep learning models because it helps to overcome the vanishing gradient problem that can occur in deep neural networks.

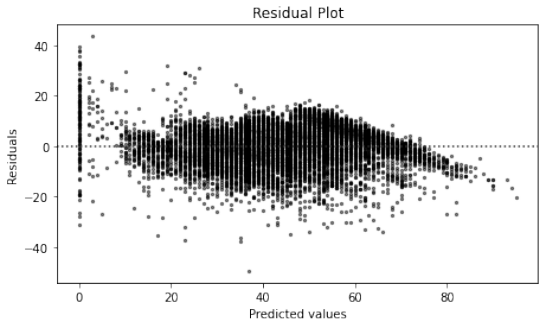
[30]:
```
# Define the model
model2 = Sequential()
model2.add(Dense(256, input_dim=X_train.shape[1], activation='relu'))
```

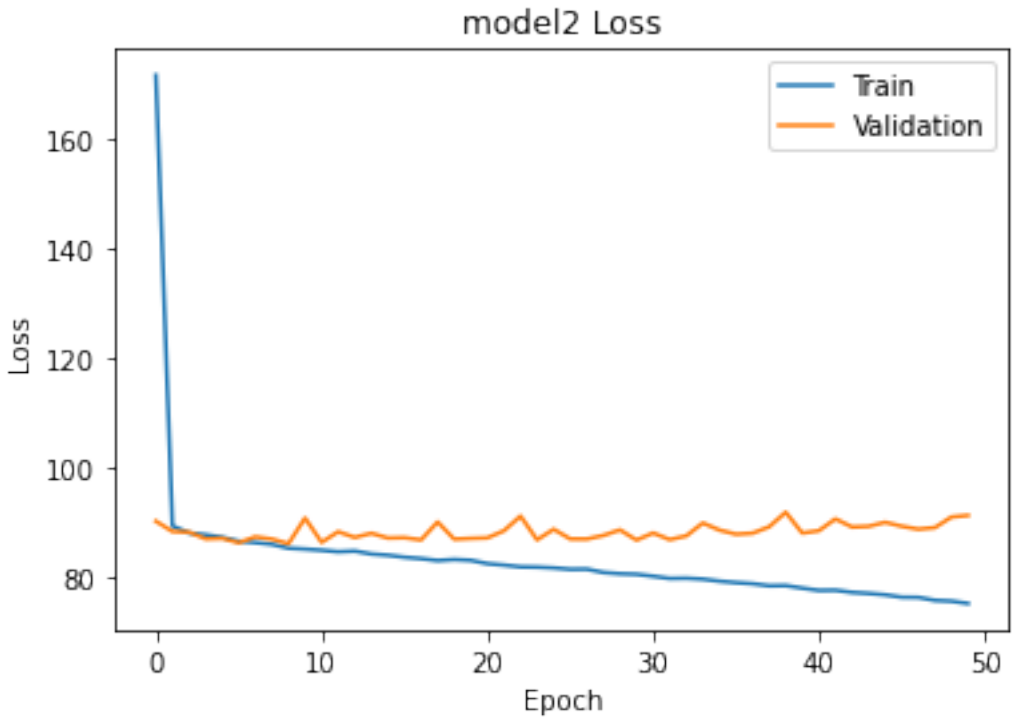
```
model2.add(Dense(128, activation='relu'))
model2.add(Dense(1))

# Compile the model
model2.compile(loss='mean_squared_error', optimizer='adam')

# Fit the model
history2 = model2.fit(X_train, y_train, epochs=50, batch_size=32,␣
 ↪validation_data=(X_valid, y_valid), verbose=0)

# Predict on the training and testing sets
y_train_pred = model2.predict(X_train)
y_test_pred = model2.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.6f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
```

```
axs[1].set_title('Residual Plot')

# Show the plot
plt.show()

# Plot the training and validation loss
plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('model2 Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
1000/1000 [==============================] - 1s 1ms/step
313/313 [==============================] - 0s 1ms/step
Training set MAE: 6.55
Training set MSE: 73.97
Training set R^2: 0.693758
Testing set MAE: 7.18
Testing set MSE: 91.472832
Testing set R^2: 0.626516
```







The results show an improvement in the performance of the model compared to the first neural network, with a higher R2 score on the training set. However, the improvement is not significant. The training set R2 score improved from 0.646 to 0.694, and the testing set R2 score decreased from 0.641 to 0.627. Indicating that adding an extra layer and more neurons improved the performance of the model on the training set but did not translate to a significant improvement in the model's ability to generalize to new, unseen data, as the testing set R2 score decreased. The test MAE increased from 7.02 to 7.18, and the test MSE increased from 87.88 to 91.47. The increase in test MAE and MSE also suggests that the model is overfitting to the training data, as it is performing worse on the testing set.

The reason for this might be that the increase in model complexity resulted in the model being able to capture more nuanced patterns in the training data, but it also led to the model overfitting to this data. A possible solution to this issue is to incorporate regularization techniques, such as dropout or L1/L2 regularization, to reduce overfitting and improve generalization performance.

6.3 Neural Network 3: Hyperparameter Tuning

Since the improvement in the last model was not significant, and there was a suspicion of overfitting, as the R2 score on the training set was much higher than the testing set. A hyperparameter tuning process was performed to find the best combination of batch size and number of epochs for the model before implementing early stopping and dropout rates to prevent overfitting.

In the hyperparameter tuning process, a grid search was performed over different values of batch size and number of epochs to find the best combination of hyperparameters. Using grid search can help to avoid the risk of overfitting to the training data and result in a model that generalizes better

to new, unseen data. The grid search used three different batch sizes and three different epoch values, resulting in a total of nine different combinations. The batch size represents the number of training samples that are used in one iteration of the training process. A smaller batch size means the weights of the model are updated more frequently, which can lead to faster convergence but also increase the noise in the gradients. On the other hand, a larger batch size means the weights are updated less frequently, which can lead to more stable gradients but also slow down the convergence. To find the optimal batch size, three are set (32, 64, 128). The number of epochs represents the number of times the entire training dataset is passed through the model during the training process. A smaller number of epochs may result in underfitting, while a larger number of epochs may result in overfitting. Therefore, 3 epoch values were set (50, 100, 200). The mean squared error was used as the scoring metric for the grid search.

[31]:
```
import logging
logging.getLogger('tensorflow').disabled = True

# Define the model
def create_model(optimizer='adam'):
    model = Sequential()
    model.add(Dense(256, input_dim=X_train.shape[1], activation='relu'))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer=optimizer)
    return model

# Create the KerasRegressor object
model3 = KerasRegressor(build_fn=create_model, verbose=0)

# Define the hyperparameters to tune
params = {
    'batch_size': [32, 64, 128],
    'epochs': [50, 100, 200],
}

# Create the GridSearchCV object
grid = GridSearchCV(estimator=model3, param_grid=params, cv=3,␣
 ↪scoring='neg_mean_squared_error')

# Fit the model
grid_result = grid.fit(X_train, y_train)

# Print the best parameters
print(f'Best parameters: {grid_result.best_params_}')
```

```
/var/folders/j8/lz7rfdrn2fj94rmkk03433dr0000gn/T/ipykernel_21414/3880811622.py:1
4: DeprecationWarning: KerasRegressor is deprecated, use Sci-Keras
(https://github.com/adriangb/scikeras) instead. See
https://www.adriangb.com/scikeras/stable/migration.html for help migrating.
  model3 = KerasRegressor(build_fn=create_model, verbose=0)
```

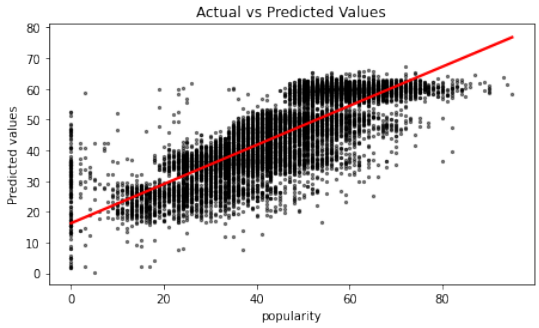
```
Best parameters: {'batch_size': 128, 'epochs': 50}
```

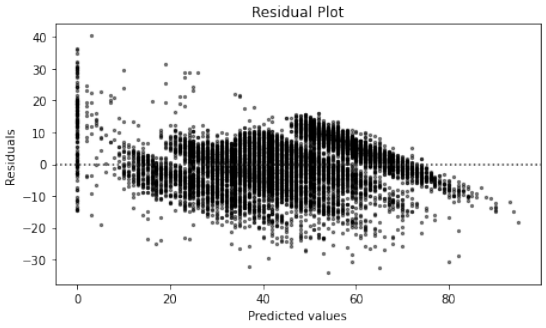
After the ideal combination was found, the model was trained on this combination to see if there was still overfitting.

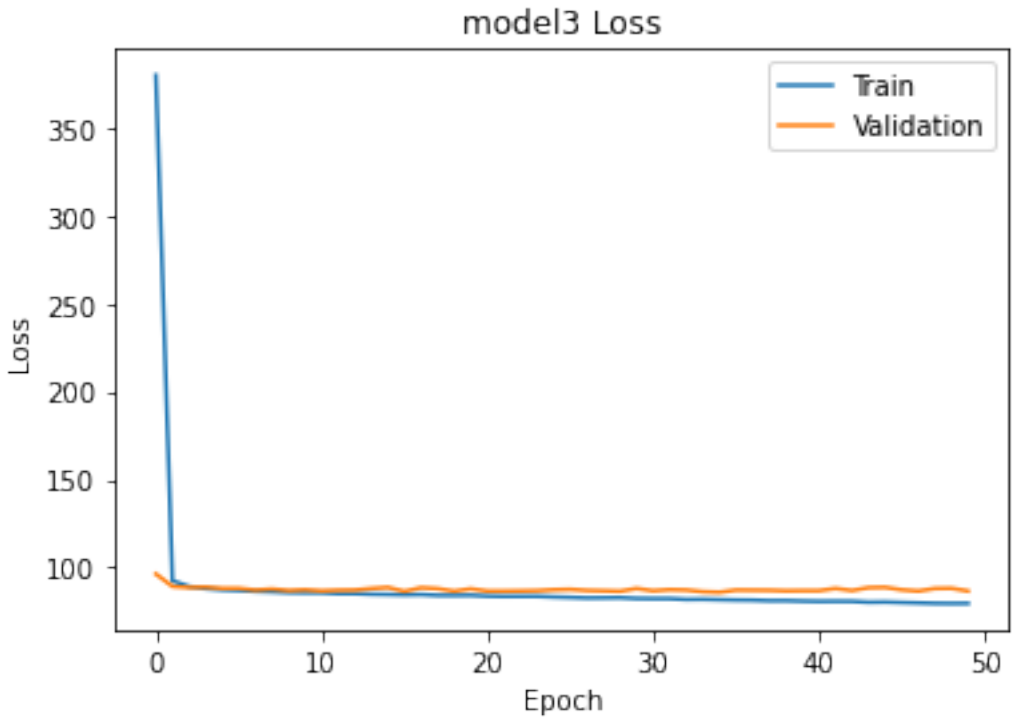
[34]:
```
# Define the model architecture
model3 = Sequential([
    Dense(256, input_shape=(X_train.shape[1],), activation='relu'),
    Dense(128, activation='relu'),
    Dense(1)
])

# Compile the model
model3.compile(optimizer='adam', loss='mse')

# Train the model
history4 = model3.fit(X_train, y_train, epochs=50, batch_size=128,␣
 ↪validation_data=(X_valid, y_valid),
                    verbose=0)

# Predict on the training and testing sets
y_train_pred = model3.predict(X_train)
y_test_pred = model3.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.2f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
```

```
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
axs[1].set_title('Residual Plot')

# Show the plot
plt.show()

# Plot the training and validation loss
plt.plot(history4.history['loss'])
plt.plot(history4.history['val_loss'])
plt.title('model3 Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
1000/1000 [==============================] - 1s 1ms/step
313/313 [==============================] - 1s 2ms/step
Training set MAE: 6.73
Training set MSE: 77.83
Training set R^2: 0.677760
Testing set MAE: 7.06
Testing set MSE: 88.21
Testing set R^2: 0.639852
```







The resulting model had a training set R2 score of 0.677760 and a testing set R2 score of 0.639852. Although the R2 score did not improve significantly from the first model, the testing set R2 score did not decrease either, indicating that the overfitting issue was resolved. The training set MAE, MSE, and R2 score showed improvements compared to the first model.

The hyperparameter tuning process helped find the best combination of batch size and number of epochs for the model, which helped prevent overfitting. The resulting model showed improvement in performance compared to the first model, and the loss plot showed that early stopping or dropout rates may be necessary to prevent further overfitting.

6.4 Neural Network 4: Early Stopping

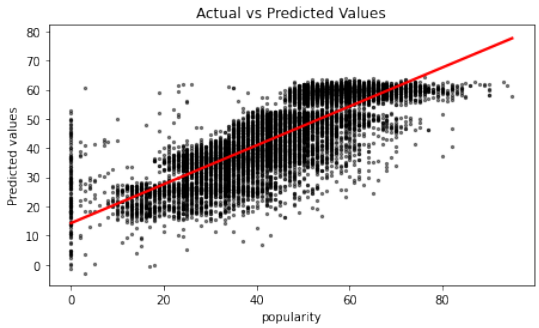
In this model, early stopping was used to prevent overfitting. Early stopping is a regularization technique that stops the training process before the model starts to overfit. The early stopping callback was defined with a patience of 10, which means that training will stop if the validation loss does not improve for 10 consecutive epochs. This ensures that the model does not continue to learn from the noise in the data and that it stops training at the optimal point where it can generalize well to unseen data. By using early stopping, we can improve the performance of the model on unseen data and prevent overfitting, leading to a better generalization of the model.

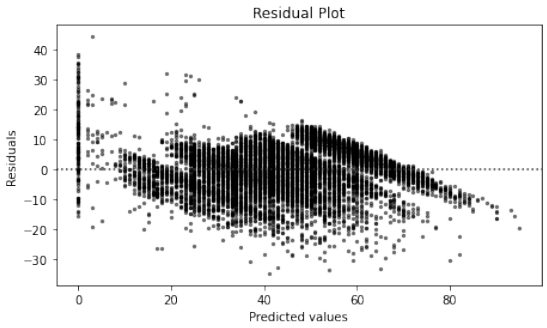
[38]:
```
# Define the model architecture
model4 = Sequential([
    Dense(256, input_shape=(X_train.shape[1],), activation='relu'),
    Dense(128, activation='relu'),
    Dense(1)
```

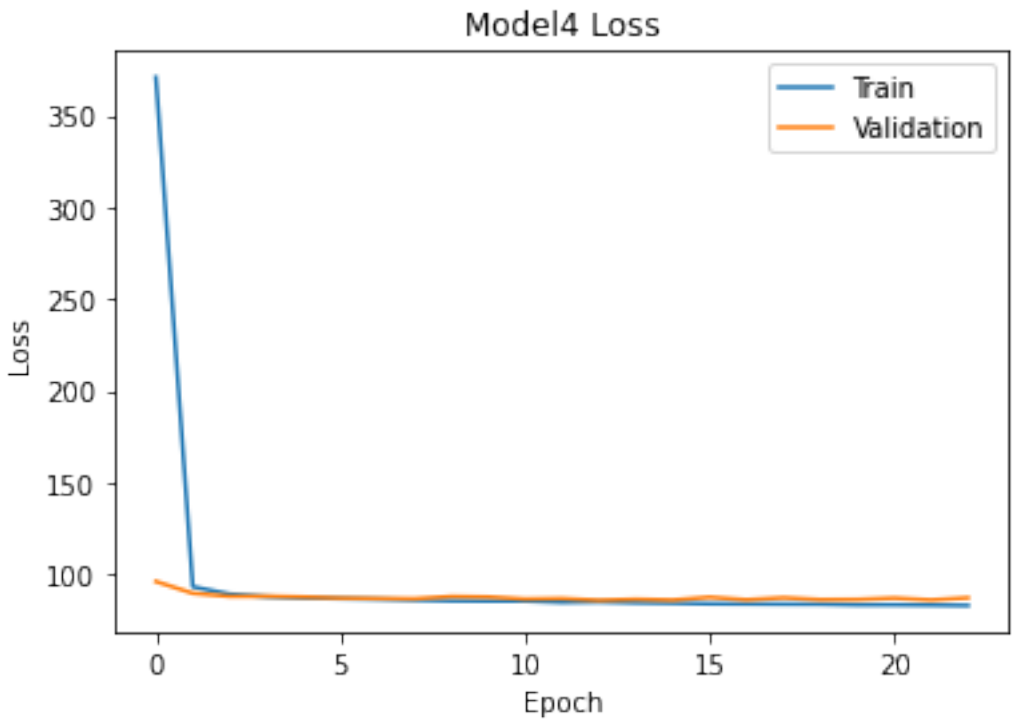
```
])

# Compile the model
model4.compile(optimizer='adam', loss='mse')

# Define early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=10)

# Train the model
history4 = model4.fit(X_train, y_train, epochs=50, batch_size=128,␣
 ↪validation_data=(X_valid, y_valid),
                    callbacks=[early_stopping], verbose=0)

# Predict on the training and testing sets
y_train_pred = model4.predict(X_train)
y_test_pred = model4.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.2f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
```

```
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
axs[1].set_title('Residual Plot')

# Show the plot
plt.show()

# Plot the training and validation loss
plt.plot(history4.history['loss'])
plt.plot(history4.history['val_loss'])
plt.title('Model4 Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
1000/1000 [==============================] - 1s 1ms/step
313/313 [==============================] - 0s 1ms/step
Training set MAE: 6.93
Training set MSE: 82.81
Training set R^2: 0.657154
Testing set MAE: 7.05
Testing set MSE: 88.21
Testing set R^2: 0.639855
```







The results show that the model with early stopping performed slightly worse on the testing set compared to the previous model, with a testing set MAE of 7.05 and R^2 of 0.639855. However, it is important to note that the early stopping technique was applied to prevent overfitting and improve generalization performance, rather than solely optimizing for the testing set performance.

6.5 Neural Network 5: Using Dropout

In this model, dropout regularization was used to experiment with regularization to see if it can reduce the error further. Dropout is a regularization technique used to prevent overfitting in neural networks. It works by randomly dropping out a certain percentage of neurons during training to prevent them from being too dependent on each other.

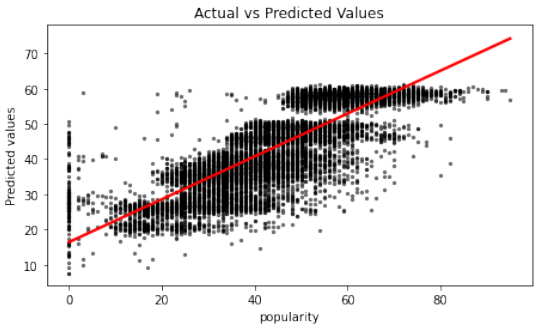
The model architecture for this model is the same as the previous one, except that two dropout layers were added with a dropout rate of 0.5. After compiling the model, early stopping was again applied with a patience of 10, and the model was trained for 50 epochs with a batch size of 128.

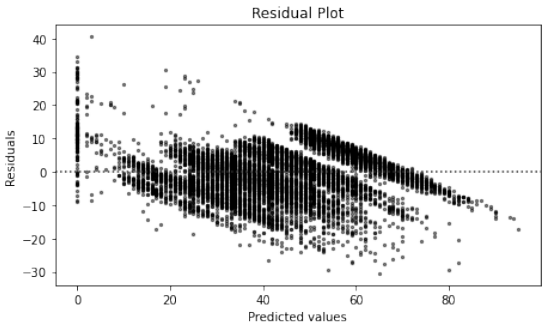
[40]:
```
from sklearn.metrics import r2_score

# Define the model architecture
model5 = Sequential([
    Dense(256, input_shape=(X_train.shape[1],), activation='relu'),
    Dropout(0.5),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(1)
```

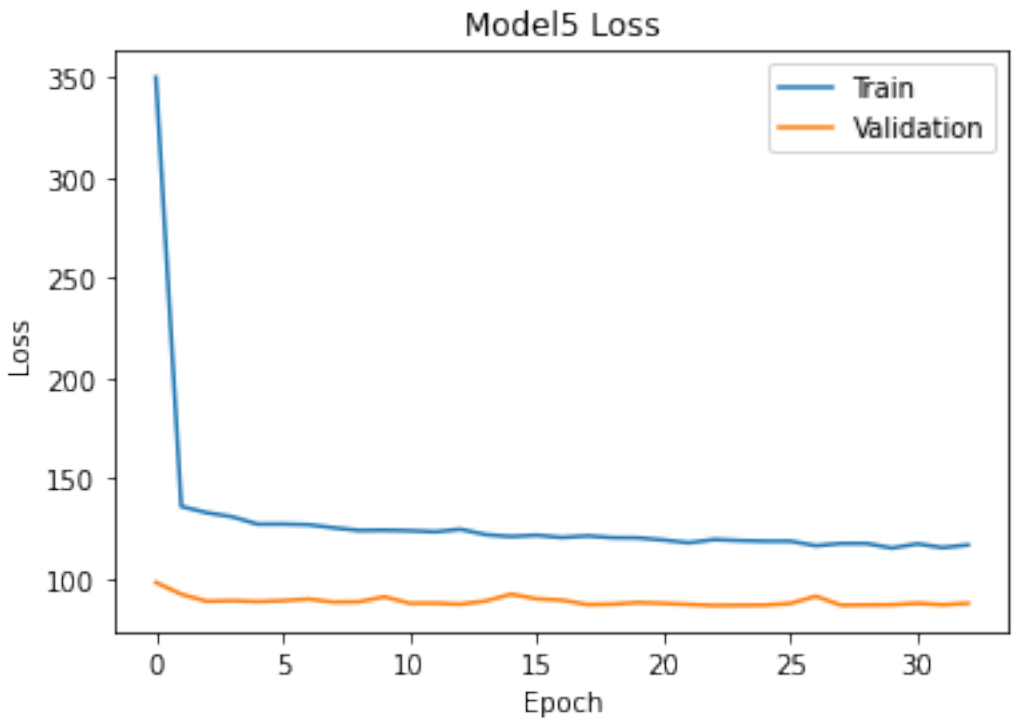
```
])

# Compile the model
model5.compile(loss='mse', metrics=['mae'])

# Define early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=10)

# Train the model
history5 = model5.fit(X_train, y_train, epochs=50, batch_size=128,␣
 ↪validation_data=(X_valid, y_valid),
                    callbacks=[early_stopping], verbose=0)

# Predict on the training and testing sets
y_train_pred = model5.predict(X_train)
y_test_pred = model5.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.2f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
```

```
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
axs[1].set_title('Residual Plot')

# Show the plot
plt.show()

# Plot the training and validation loss
plt.plot(history5.history['loss'])
plt.plot(history5.history['val_loss'])
plt.title('Model5 Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
1000/1000 [==============================] - 1s 1ms/step
313/313 [==============================] - 0s 1ms/step
Training set MAE: 6.98
Training set MSE: 86.32
Training set R^2: 0.642604
Testing set MAE: 7.01
Testing set MSE: 88.99
Testing set R^2: 0.636659
```







The results show that the model with dropout regularization performed slightly worse than the previous model without dropout regularization. The testing set R^2 is slightly lower, indicating that the model is explaining less of the variance in the testing data.

6.6 Neural Network 6: Reducing the Dropout Rate

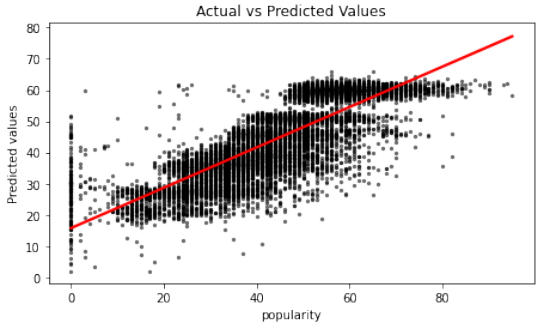
In this model, the dropout rate was reduced in order to experiment with regularization and potentially reduce the error further. The previous model had too much dropout, leading to underfitting and a higher error. By reducing the dropout rate, the model may be been able to learn more specific features and fit the data better.

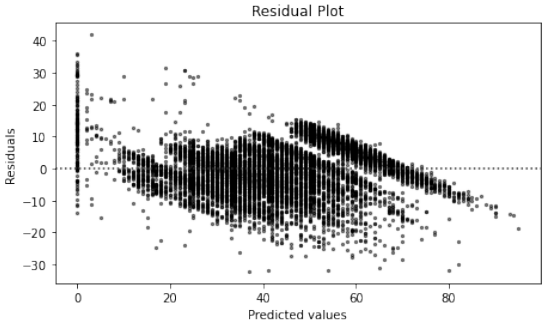
[42]:
```
# Define the model architecture
model6 = Sequential([
    Dense(256, input_shape=(X_train.shape[1],), activation='relu'),
    Dropout(0.2),
    Dense(128, activation='relu'),
    Dropout(0.2),
    Dense(1)
])

# Compile the model
model6.compile(loss='mse', metrics=['mae'])

# Define early stopping callback
```

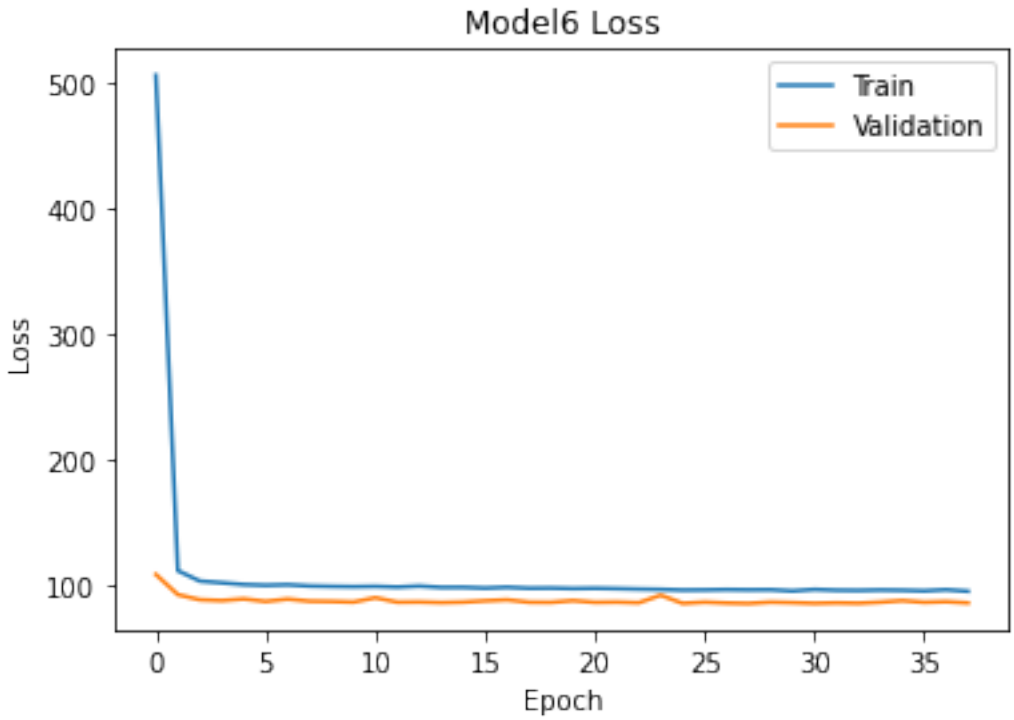
```
early_stopping = EarlyStopping(monitor='val_loss', patience=10)

# Train the model
history6 = model6.fit(X_train, y_train, epochs=50, batch_size=256,␣
 ↪validation_data=(X_valid, y_valid),
                    callbacks=[early_stopping], verbose=0)

# Predict on the training and testing sets
y_train_pred = model6.predict(X_train)
y_test_pred = model6.predict(X_test)

# Calculate the MAE, MSE, and R^2 for the training set
train_mae = mean_absolute_error(y_train, y_train_pred)
train_mse = mean_squared_error(y_train, y_train_pred)
train_r2 = r2_score(y_train, y_train_pred)

# Calculate the MAE, MSE, and R^2 for the testing set
test_mae = mean_absolute_error(y_test, y_test_pred)
test_mse = mean_squared_error(y_test, y_test_pred)
test_r2 = r2_score(y_test, y_test_pred)

# Print the scores
print(f'Training set MAE: {train_mae:.2f}')
print(f'Training set MSE: {train_mse:.2f}')
print(f'Training set R^2: {train_r2:.6f}')

print(f'Testing set MAE: {test_mae:.2f}')
print(f'Testing set MSE: {test_mse:.2f}')
print(f'Testing set R^2: {test_r2:.6f}')

# Create scatterplot of actual vs. predicted values
fig, axs = plt.subplots(ncols=2, figsize=(16,4))
axs[0].scatter(y_test, y_test_pred, alpha=0.5, color='black', s=5)
axs[0].set_xlabel('Actual values')
axs[0].set_ylabel('Predicted values')
axs[0].set_title('Actual vs Predicted Values')
axs[0].plot(color='black', linestyle='--')
sns.regplot(x=y_test, y=y_test_pred, scatter=False, color='red', ax=axs[0])

# Create residual plot
sns.residplot(x=y_test, y=y_test_pred, ax=axs[1], color='black',␣
 ↪scatter_kws={'alpha':0.5, 's':5})
axs[1].set_xlabel('Predicted values')
axs[1].set_ylabel('Residuals')
axs[1].set_title('Residual Plot')

# Show the plot
```

```
plt.show()

# Plot the training and validation loss
plt.plot(history6.history['loss'])
plt.plot(history6.history['val_loss'])
plt.title('Model6 Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
```

```
1000/1000 [==============================] - 1s 1ms/step
313/313 [==============================] - 0s 1ms/step
Training set MAE: 6.91
Training set MSE: 81.98
Training set R^2: 0.660567
Testing set MAE: 7.03
Testing set MSE: 87.44
Testing set R^2: 0.642986
```







The results of the model show a slightly lower MAE and MSE for both the training and testing sets compared to the previous model, indicating a better fit to the data. The R^2 score also increased slightly for both sets. Thus, this model has proved to be the best model with the highest accuracy and lower MAE and MSE.

7 Evaluation

The primary goal of this report was to analyze which factors affect the popularity of a song, and to what extent as well as whether music genre influences a song's popularity. Initial modeling with linear regression and random forest, suggests that there is a linear relationship between popularity (independent variable) and the other corresponding variables (dependent variables). However the results of these machine learning models indicate slight overfitting to the training data, which were improved through deep learning models and the employment of regularization techniques.

1. **Linear Regression:** MSE = 91.68 | R^2 = 62.5678%

2. **Random Forest Regressor:** MSE = 87.94 | R^2 = 64.0921%

To improve the accuracy of predicitions, artificial neural networks were employed. The advanced structure of deep learning models can capture a greater amount of complexities and hidden layers within the data. Therefore, neural networks were iteratively adjusted and improved to enhance their performance.

3. **Neural Network 1 (simple architecture):** MSE = 87.88 | R^2 = 64.1169%

4. **Neural Network 2 (increasing layers and number of neurons):** MSE = 91.47 | R^2 =

62.6516%

5. **Neural Network 3 (hyperparameter tuning):** MSE = 88.21 | R^2 = 63.9852%

6. **Neural Network 4 (early stopping):** MSE = 88.21 | R^2 = 63.9855%

7. **Neural Network 5 (dropout):** MSE = 88.99 | R^2 = 63.6659%

8. **Neural Network 6 (reducing dropout):** MSE = 87.44 | R^2 = 64.2986%

After comparing the simple regressor models to the neural networks, it was discovered that the neural networks enhanced performance. However, the second neural network with an extra layer and more neurons showed improvement in the R^2 score on the training set but did not perform as well on the testing set compared to the first neural network. This suggests that the added complexity did not necessarily lead to better generalization to new data. Moreover, the fifth neural networl with dropout did not improve the model performance as compared to the first neural network, suggesting that the dropout rate may have been too high and led to underfitting of the data. Although, the performance did not improve significantly, it can be concluded that the final neural network with a more complex architecture, increased hidden layers and neurons, tuned hyperparameters,and reduced drop out provided the best performance as it was able to capture more complex patterns within the data. Since this model was trained with the optimal number of epochs, batch size and early stopping, the model successfully avoided overfitting and significant underfitting.

8 Conclusion

This study reveals that identifying songs with low popularity is achievable with basic machine learning models. However, utilizing simple machine learning ensemble models for predicting popularity yielded unsatisfactory results. To improve model accuracy, artificial neural networks were developed, which demonstrated marginally better performance compared to simpler models. Nonetheless, the moderate enhancement indicates that the assumption that neural networks surpass traditional machine learning models is predominantly reliant on the characteristics of the data, such as quality and size. Despite the limitations, the findings suggest that the integration of optimized artificial neural networks can enhance the accuracy of models in predicting the popularity of songs.

9 References

Dataset found here: https://www.kaggle.com/datasets/vicsuperman/prediction-of-music-genre

Géron, A. (2023) Hands-on machine learning with scikit-learn, keras and tensorflow: Concepts, tools, and techniques to build Intelligent Systems. 3rd edn. Sebastapol, CA: O'Reilly.

Alexis, M. (2023) What is the Spotify popularity index?, Two Story Melody. Available at: https://twostorymelody.com/spotify-popularity-index/#:~:text=The%20benefits%20of%20a%20high,add%20it%20to%20different%20playlists.

Di, W. (2018) Deep Learning Essentials: Your hands-on guide to the fundamentals of Deep Learning and neural network modeling. Birmingham: Packt.