"Music Genre Classification Project"

Music Genre Classification Project

Complete Project Code Documentation

Project: Music Genre Classification using Machine Learning
Authors: Niyat Kahsay & Marwah Faraj
Date: Summer 2025

This document contains all project code and documentation converted to PDF format using Jupyter nbconvert and pandoc.

Generated on: August 08, 2025 at 08:25 PM

01_ExploratoryAnalysis_Feature_Engineering

August 8, 2025

##Project: Music Genre Classification using Machine Learning

Niyat Kahsay & Marwah Faraj Summer 2025

Description:

This project aims to automatically classify songs into genres based on audio features provided in the Spotify 1.2M Songs Dataset. The workflow includes data exploration, preprocessing, model building, evaluation, and visualization.

Purpose:

Apply supervised learning techniques on real-world audio data Explore audio feature-based genre classification Build a portfolio-ready project demonstrating practical machine learning skills

0.1 Import Libraries

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import OrdinalEncoder, LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score,␣
↪classification_report, confusion_matrix
from imblearn.over_sampling import SMOTE
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from scipy.stats import mode
import warnings
import os
from matplotlib.colors import LinearSegmentedColormap

warnings.filterwarnings("ignore", category=FutureWarning)
```

[14]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

0.2 Load Data

[15]:
```
import pandas as pd
data = pd.read_csv('/content/drive/MyDrive/music-genre-classification/data/
↪spotify_songs.csv')
data.head()
```

[15]:
```
                 track_id                                          track_name  \
0  6f807x0ima9a1j3VPbc7VN  I Don't Care (with Justin Bieber) - Loud Luxur…
1  0r7CVbZTWZgbTCYdfa2P31                    Memories - Dillon Francis Remix
2  1z1Hg7Vb0AhHDiEmnDE79l                    All the Time - Don Diablo Remix
3  75FpbthrwQmzHlBJLuGdC7                  Call You Mine - Keanu Silva Remix
4  1e8PAfcKUYoKkxPhrHqw4x            Someone You Loved - Future Humans Remix

       track_artist  track_popularity          track_album_id  \
0        Ed Sheeran                66  2oCs0DGTsRO98Gh5ZSl2Cx
1          Maroon 5                67  63rPSO264uRjW1X5E6cWv6
2      Zara Larsson                70  1HoSmj2eLcsrR0vE9gThr4
3  The Chainsmokers                60  1nqYsOef1yKKuGOVchbsk6
4     Lewis Capaldi                69  7m7vv9wlQ4i0LFuJiE2zsQ

                                    track_album_name track_album_release_date  \
0  I Don't Care (with Justin Bieber) [Loud Luxury…               2019-06-14
1                    Memories (Dillon Francis Remix)               2019-12-13
2                    All the Time (Don Diablo Remix)               2019-07-05
3                        Call You Mine - The Remixes               2019-07-19
4            Someone You Loved (Future Humans Remix)               2019-03-05

  playlist_name             playlist_id playlist_genre  … key  loudness  \
0     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   6    -2.634
1     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …  11    -4.969
2     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   1    -3.432
3     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   7    -3.778
4     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   1    -4.672

   mode  speechiness  acousticness  instrumentalness  liveness  valence  \
0     1       0.0583        0.1020          0.000000    0.0653    0.518
1     1       0.0373        0.0724          0.004210    0.3570    0.693
2     0       0.0742        0.0794          0.000023    0.1100    0.613
3     1       0.1020        0.0287          0.000009    0.2040    0.277
4     1       0.0359        0.0803          0.000000    0.0833    0.725

     tempo  duration_ms
0  122.036       194754
```

```
1   99.972       162600
2  124.008       176616
3  121.956       169093
4  123.976       189052

[5 rows x 23 columns]
```

#Data Exploration

[16]:
```python
print(data.columns)
```

```
Index(['track_id', 'track_name', 'track_artist', 'track_popularity',
       'track_album_id', 'track_album_name', 'track_album_release_date',
       'playlist_name', 'playlist_id', 'playlist_genre', 'playlist_subgenre',
       'danceability', 'energy', 'key', 'loudness', 'mode', 'speechiness',
       'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo',
       'duration_ms'],
      dtype='object')
```

[17]:
```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32833 entries, 0 to 32832
Data columns (total 23 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   track_id                  32833 non-null  object
 1   track_name                32828 non-null  object
 2   track_artist              32828 non-null  object
 3   track_popularity          32833 non-null  int64
 4   track_album_id            32833 non-null  object
 5   track_album_name          32828 non-null  object
 6   track_album_release_date  32833 non-null  object
 7   playlist_name             32833 non-null  object
 8   playlist_id               32833 non-null  object
 9   playlist_genre            32833 non-null  object
 10  playlist_subgenre         32833 non-null  object
 11  danceability              32833 non-null  float64
 12  energy                    32833 non-null  float64
 13  key                       32833 non-null  int64
 14  loudness                  32833 non-null  float64
 15  mode                      32833 non-null  int64
 16  speechiness               32833 non-null  float64
 17  acousticness              32833 non-null  float64
 18  instrumentalness          32833 non-null  float64
 19  liveness                  32833 non-null  float64
 20  valence                   32833 non-null  float64
 21  tempo                     32833 non-null  float64
 22  duration_ms               32833 non-null  int64
```

```
dtypes: float64(9), int64(4), object(10)
memory usage: 5.8+ MB
```

[18]:
```python
data.describe(include = "all")
```

[18]:
```
                      track_id track_name   track_artist  track_popularity  \
count                    32833      32828          32828      32833.000000
unique                   28356      23449          10692               NaN
top     7BKLCZ1jbUBVqRi2FVlTVw     Poison  Martin Garrix               NaN
freq                        10         22            161               NaN
mean                       NaN        NaN            NaN         42.477081
std                        NaN        NaN            NaN         24.984074
min                        NaN        NaN            NaN          0.000000
25%                        NaN        NaN            NaN         24.000000
50%                        NaN        NaN            NaN         45.000000
75%                        NaN        NaN            NaN         62.000000
max                        NaN        NaN            NaN        100.000000

                track_album_id track_album_name track_album_release_date  \
count                    32833            32828                    32833
unique                   22545            19743                     4530
top     5L1xcowSxwzFUSJzvyMp48    Greatest Hits               2020-01-10
freq                        42              139                      270
mean                       NaN              NaN                      NaN
std                        NaN              NaN                      NaN
min                        NaN              NaN                      NaN
25%                        NaN              NaN                      NaN
50%                        NaN              NaN                      NaN
75%                        NaN              NaN                      NaN
max                        NaN              NaN                      NaN

          playlist_name             playlist_id playlist_genre  …  \
count             32833                   32833          32833  …
unique              449                     471              6  …
top     Indie Poptimism  4JkkvMpVl4lSioqQjeAL0q            edm  …
freq                308                     247           6043  …
mean                NaN                     NaN            NaN  …
std                 NaN                     NaN            NaN  …
min                 NaN                     NaN            NaN  …
25%                 NaN                     NaN            NaN  …
50%                 NaN                     NaN            NaN  …
75%                 NaN                     NaN            NaN  …
max                 NaN                     NaN            NaN  …

                 key      loudness          mode   speechiness  acousticness  \
count   32833.000000  32833.000000  32833.000000  32833.000000  32833.000000
unique           NaN           NaN           NaN           NaN           NaN
```

```
top               NaN          NaN          NaN          NaN          NaN
freq              NaN          NaN          NaN          NaN          NaN
mean         5.374471    -6.719499     0.565711     0.107068     0.175334
std          3.611657     2.988436     0.495671     0.101314     0.219633
min          0.000000   -46.448000     0.000000     0.000000     0.000000
25%          2.000000    -8.171000     0.000000     0.041000     0.015100
50%          6.000000    -6.166000     1.000000     0.062500     0.080400
75%          9.000000    -4.645000     1.000000     0.132000     0.255000
max         11.000000     1.275000     1.000000     0.918000     0.994000

        instrumentalness      liveness       valence         tempo  \
count       32833.000000  32833.000000  32833.000000  32833.000000
unique               NaN           NaN           NaN           NaN
top                  NaN           NaN           NaN           NaN
freq                 NaN           NaN           NaN           NaN
mean            0.084747      0.190176      0.510561    120.881132
std             0.224230      0.154317      0.233146     26.903624
min             0.000000      0.000000      0.000000      0.000000
25%             0.000000      0.092700      0.331000     99.960000
50%             0.000016      0.127000      0.512000    121.984000
75%             0.004830      0.248000      0.693000    133.918000
max             0.994000      0.996000      0.991000    239.440000

          duration_ms
count    32833.000000
unique            NaN
top               NaN
freq              NaN
mean    225799.811622
std      59834.006182
min       4000.000000
25%     187819.000000
50%     216000.000000
75%     253585.000000
max     517810.000000

[11 rows x 23 columns]
```

[24]:

```
# Genre Feature Radar Chart (Interactive)
genre_means = data.groupby('playlist_genre')[audio_features].mean().
 ↪reset_index()

fig = go.Figure()
for genre in genre_means['playlist_genre']:
    fig.add_trace(go.Scatterpolar(
        r=genre_means[genre_means['playlist_genre'] == genre][audio_features].
 ↪values[0],
```

```
        theta=audio_features,
        fill='toself',
        name=genre
    ))

fig.update_layout(
    polar=dict(radialaxis=dict(visible=True, range=[0, 1])),
    showlegend=True,
    title='Audio Feature Profiles by Genre',
    height=600
)
fig.show()
```

##Data Preprocessing

[25]:
```
# Handle duplicates
data = data.drop_duplicates(subset=['track_id'])

# Fix release dates
def fix_date(x):
    if pd.isnull(x):
        return x
    if isinstance(x, str):
        return f"{x}-01-01" if len(x) < 10 else x
    return x.strftime('%Y-%m-%d')

data['track_album_release_date'] = data['track_album_release_date'].
  ↪apply(fix_date)
data['track_album_release_date'] = pd.
  ↪to_datetime(data['track_album_release_date'], errors='coerce')


# Extract release year
data['release_year'] = data['track_album_release_date'].dt.year

# Encode Target Variable
label_encoder = LabelEncoder()
data['genre_label'] = label_encoder.fit_transform(data['playlist_genre'])
print("  Target encoding complete!")
print(label_encoder.classes_)

# Feature Scaling
audio_features += ['duration_ms', 'release_year']
scaler = StandardScaler()
data[audio_features] = scaler.fit_transform(data[audio_features])
```

```
 Target encoding complete!
['edm' 'latin' 'pop' 'r&b' 'rap' 'rock']
```

[26]:
```python
print(data.isna().sum())
data = data.dropna()
```

```
track_id                     0
track_name                   4
track_artist                 4
track_popularity             0
track_album_id               0
track_album_name             4
track_album_release_date    25
playlist_name                0
playlist_id                  0
playlist_genre               0
playlist_subgenre            0
danceability                 0
energy                       0
key                          0
loudness                     0
mode                         0
speechiness                  0
acousticness                 0
instrumentalness             0
liveness                     0
valence                      0
tempo                        0
duration_ms                  0
release_year                25
genre_label                  0
dtype: int64
```

[26]:
```python

```

[ ]:
```python
# Install kaleido for plotly static image export if needed
try:
    import kaleido
except ImportError:
    import subprocess
    import sys
    subprocess.check_call([sys.executable, "-m", "pip", "install", "kaleido"])
    import kaleido

warnings.filterwarnings("ignore", category=FutureWarning)

#
#   WHITE BACKGROUND & GOLDEN THEME SETUP
#

# Set up white background and golden color scheme
```

```python
plt.style.use('default') # Use default style for white background
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B', '#A0522D'] # Adjusted golden palette

print("  Libraries imported successfully!")
print("  White background and golden color scheme activated!")

# Define the base directory for saving plots in Google Drive
PLOT_SAVE_DIR = '/content/drive/MyDrive/music-genre-classification/images/eda'
os.makedirs(PLOT_SAVE_DIR, exist_ok=True)
print(f"  Plots will be saved to: {PLOT_SAVE_DIR}")


#
#   UPDATED VISUALIZATION CODE
#

# 1. GENRE DISTRIBUTION VISUALIZATION
def create_genre_distribution_plot(data):
    """Create genre distribution bar chart with white background and golden aesthetic"""
    plt.figure(figsize=(12, 6), facecolor='white')
    genre_counts = data['playlist_genre'].value_counts()
    ax = sns.barplot(x=genre_counts.index, y=genre_counts.values, palette=golden_palette)
    ax.set_facecolor('white')
    plt.title('Song Distribution by Genre', fontsize=18, color='black', fontweight='bold', pad=20)
    plt.xlabel('Genre', fontsize=14, color='black', fontweight='semibold')
    plt.ylabel('Count', fontsize=14, color='black', fontweight='semibold')

    # Style the plot
    ax.tick_params(colors='black', labelsize=11)
    ax.spines['bottom'].set_color('black')
    ax.spines['left'].set_color('black')
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.grid(True, alpha=0.2, color='gray')

    # Add percentage labels
    total = len(data)
    for p in ax.patches:
        percentage = f'{100 * p.get_height()/total:.1f}%'
        ax.annotate(percentage, (p.get_x() + p.get_width()/2., p.get_height()),
                    ha='center', va='center', xytext=(0, 10),
                    textcoords='offset points', fontsize=11, color='black', fontweight='bold')
```

```
    # Save the plot
    plt.savefig(os.path.join(PLOT_SAVE_DIR, 'genre_distribution.png'),
                facecolor='white', edgecolor='none', dpi=300,␣
 ↪bbox_inches='tight')
    plt.show()
    print(f" Plot saved to: {os.path.join(PLOT_SAVE_DIR, 'genre_distribution.
 ↪png')}")

# 2. CORRELATION MATRIX VISUALIZATION
def create_correlation_matrix(data):
    """Create correlation matrix heatmap with white background and golden␣
 ↪theme"""
    plt.figure(figsize=(14, 10), facecolor='white')
    audio_features = ['danceability', 'energy', 'loudness',
                     'acousticness', 'valence', 'tempo',
                     'speechiness', 'instrumentalness']
    corr_matrix = data[audio_features].corr()
    mask = np.triu(np.ones_like(corr_matrix, dtype=bool))

    # Create custom colormap with golden tones
    colors = ['white', '#F5DEB3', '#D4AF37', '#CD853F', '#8B4513', '#2C1810']
    golden_cmap = LinearSegmentedColormap.from_list('golden', colors, N=256)

    ax = plt.gca()
    ax.set_facecolor('white')
    sns.heatmap(corr_matrix, mask=mask, annot=True, fmt=".2f",
                cmap=golden_cmap, linewidths=0.5, linecolor='gray',
                annot_kws={'color': 'black', 'fontweight': 'bold'},
                cbar_kws={'shrink': 0.8})
    plt.title("Audio Feature Correlation Matrix", fontsize=18, color='black',␣
 ↪fontweight='bold', pad=20)

    # Style axes
    ax.tick_params(colors='black', labelsize=11)
    plt.xticks(rotation=45, ha='right', color='black')
    plt.yticks(rotation=0, color='black')

    # Save the plot
    plt.savefig(os.path.join(PLOT_SAVE_DIR, 'correlation_matrix.png'),
                facecolor='white', edgecolor='none', dpi=300,␣
 ↪bbox_inches='tight')
    plt.show()
    print(f" Plot saved to: {os.path.join(PLOT_SAVE_DIR, 'correlation_matrix.
 ↪png')}")
```

```python
# 3. FEATURE DISTRIBUTION BOXPLOTS
def create_feature_boxplots(data):
    """Create feature distribution boxplots with white background and golden theme"""
    audio_features = ['danceability', 'energy', 'loudness',
                      'acousticness', 'valence', 'tempo',
                      'speechiness', 'instrumentalness']

    plt.figure(figsize=(15, 10), facecolor='white')
    for i, feature in enumerate(audio_features[:6], 1):
        plt.subplot(2, 3, i)
        ax = plt.gca()
        ax.set_facecolor('white')

        sns.boxplot(x='playlist_genre', y=feature, data=data, palette=golden_palette)
        plt.title(f'{feature.capitalize()} by Genre', fontsize=14, color='black', fontweight='bold', pad=10)
        plt.xticks(rotation=45, color='black', fontsize=10)
        plt.yticks(color='black', fontsize=10)
        plt.xlabel('Genre', color='black', fontsize=11, fontweight='semibold')
        plt.ylabel(feature.capitalize(), color='black', fontsize=11, fontweight='semibold')

        # Style the subplot
        ax.tick_params(colors='black')
        for spine in ax.spines.values():
            spine.set_color('black')
            spine.set_linewidth(0.8)
        ax.grid(True, alpha=0.2, color='gray')

    plt.tight_layout()
    # Save the plot
    plt.savefig(os.path.join(PLOT_SAVE_DIR, 'feature_distributions.png'),
                facecolor='white', edgecolor='none', dpi=300, bbox_inches='tight')
    plt.show()
    print(f" Plot saved to: {os.path.join(PLOT_SAVE_DIR, 'feature_distributions.png')}")

# 4. INTERACTIVE RADAR CHART
def create_radar_chart(data):
    """Create interactive radar chart with white background and golden theme"""
    audio_features = ['danceability', 'energy', 'loudness',
                      'acousticness', 'valence', 'tempo',
                      'speechiness', 'instrumentalness']
```

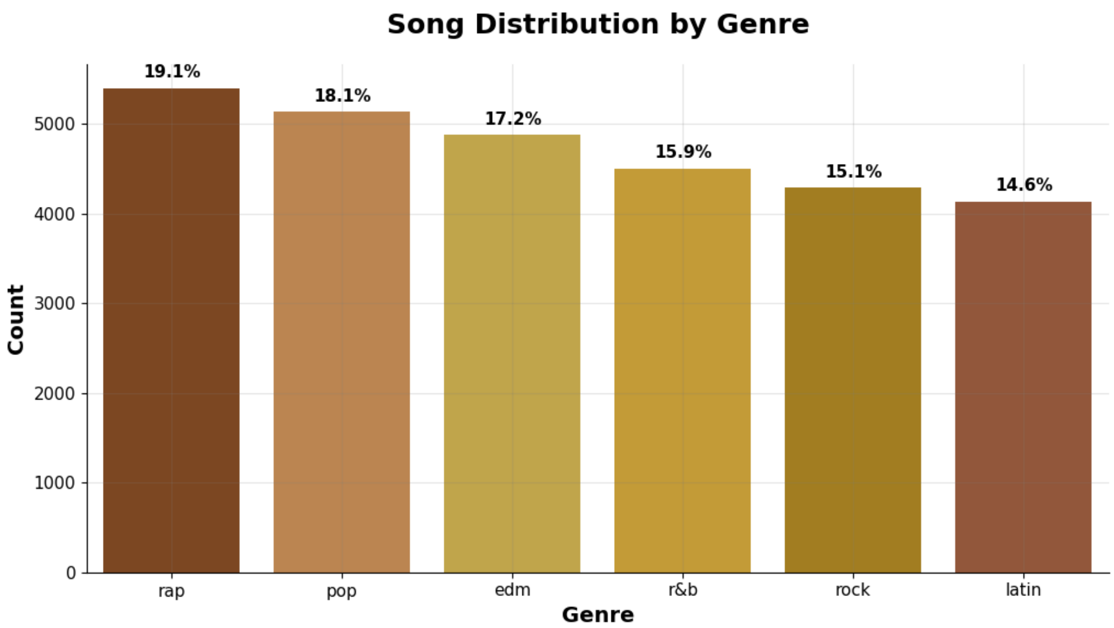
```
genre_means = data.groupby('playlist_genre')[audio_features].mean().
↪reset_index()

# Define golden color palette for radar chart
radar_colors = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
↪'#A0522D'] # Adjusted golden palette

fig = go.Figure()
for i, genre in enumerate(genre_means['playlist_genre']):
    fig.add_trace(go.Scatterpolar(
        r=genre_means[genre_means['playlist_genre'] ==␣
↪genre][audio_features].values[0],
        theta=audio_features,
        fill='toself',
        name=genre,
        line=dict(color=radar_colors[i % len(radar_colors)], width=3),
        fillcolor=radar_colors[i % len(radar_colors)],
        opacity=0.6
    ))

fig.update_layout(
    polar=dict(
        radialaxis=dict(
            visible=True,
            range=[0, 1],
            tickfont=dict(color='black', size=12),
            gridcolor='gray',
            linecolor='gray'
        ),
        angularaxis=dict(
            tickfont=dict(color='black', size=12),
            gridcolor='gray',
            linecolor='gray'
        ),
        bgcolor='white'
    ),
    showlegend=True,
    title=dict(
        text='Audio Feature Profiles by Genre',
        font=dict(color='black', size=18, family='Arial Black'),
        x=0.5
    ),
    height=600,
    paper_bgcolor='white',
    plot_bgcolor='white',
    font=dict(color='black'),
```

```
        legend=dict(
            font=dict(color='black', size=12),
            bgcolor='rgba(255, 255, 255, 0.8)',
            bordercolor='gray',
            borderwidth=1
        )
    )

    # Save the interactive plot as HTML
    fig.write_html(os.path.join(PLOT_SAVE_DIR, 'radar_chart.html'))
    # fig.write_image(os.path.join(PLOT_SAVE_DIR, 'radar_chart.png'),␣
↪width=800, height=600) # Still commenting this out due to kaleido issues
    fig.show()
    print(f" Interactive plot saved to: {os.path.join(PLOT_SAVE_DIR,␣
↪'radar_chart.html')}")
    # print(f" Static plot saved to: {os.path.join(PLOT_SAVE_DIR, 'radar_chart.
↪png')}")

#
#   USAGE INSTRUCTIONS
#

"""
USAGE INSTRUCTIONS:

1. Replace your existing plotting cells in the notebook with the function calls:

   # Instead of the old genre distribution code, use:
   create_genre_distribution_plot(data)

   # Instead of the old correlation matrix code, use:
   create_correlation_matrix(data)

   # Instead of the old boxplot code, use:
   create_feature_boxplots(data)

   # Instead of the old radar chart code, use:
   create_radar_chart(data)

2. All plots will automatically be saved to the directory specified by␣
↪PLOT_SAVE_DIR

3. The color scheme matches your presentation slide with:
   - Dark brown background (#2C1810)
   - Golden palette for data visualization
   - Elegant typography and styling
```

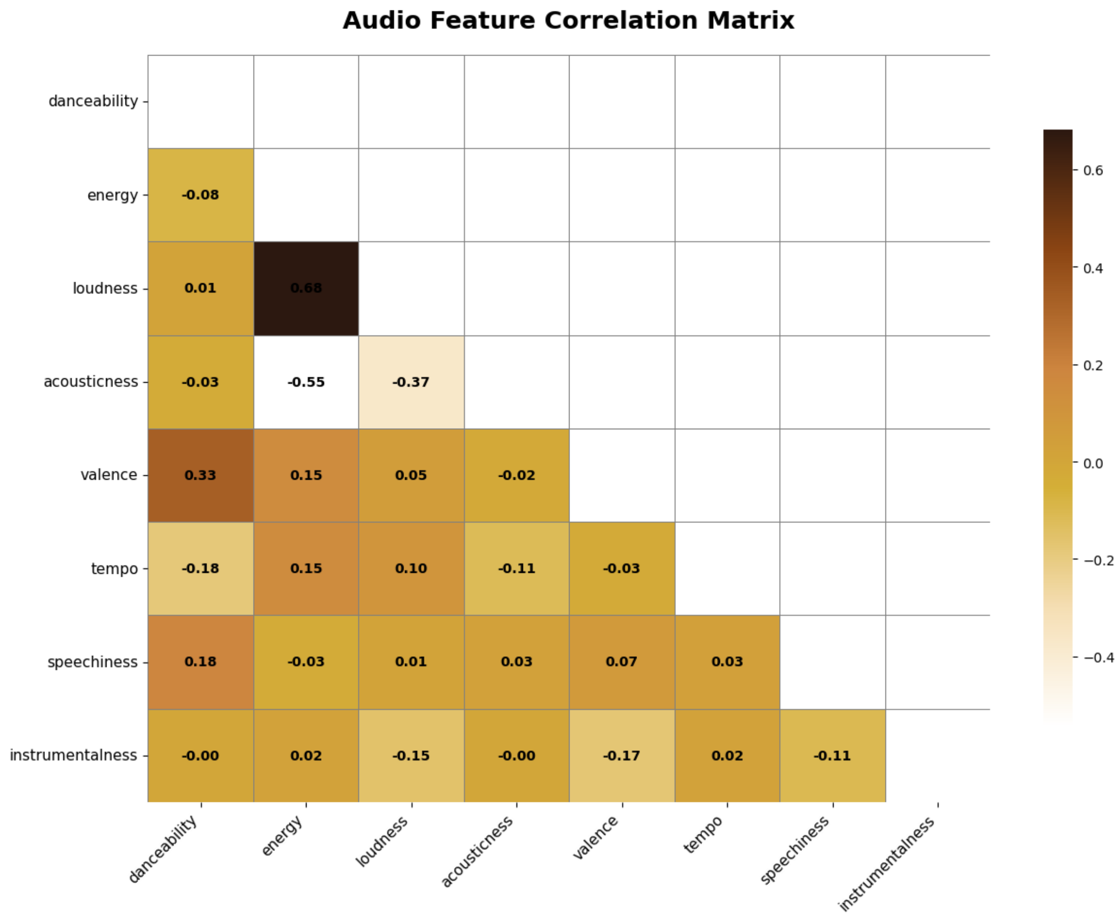
```
4. Make sure your data loading path is correct for your environment:
   - Colab: '/content/spotify_songs.csv'
   - Local: './data/spotify_songs.csv'
   - Current: '/workspace/data/spotify_songs.csv'
"""

print("  Updated Feature Engineering Code Ready!")
print("  Copy the functions above into your notebook cells")
print("  Your visualizations will match the slide aesthetic perfectly!")
```

```
 Libraries imported successfully!
 White background and golden color scheme activated!
 Plots will be saved to: /content/drive/MyDrive/music-genre-
classification/images/eda
 Updated Feature Engineering Code Ready!
 Copy the functions above into your notebook cells
 Your visualizations will match the slide aesthetic perfectly!
```

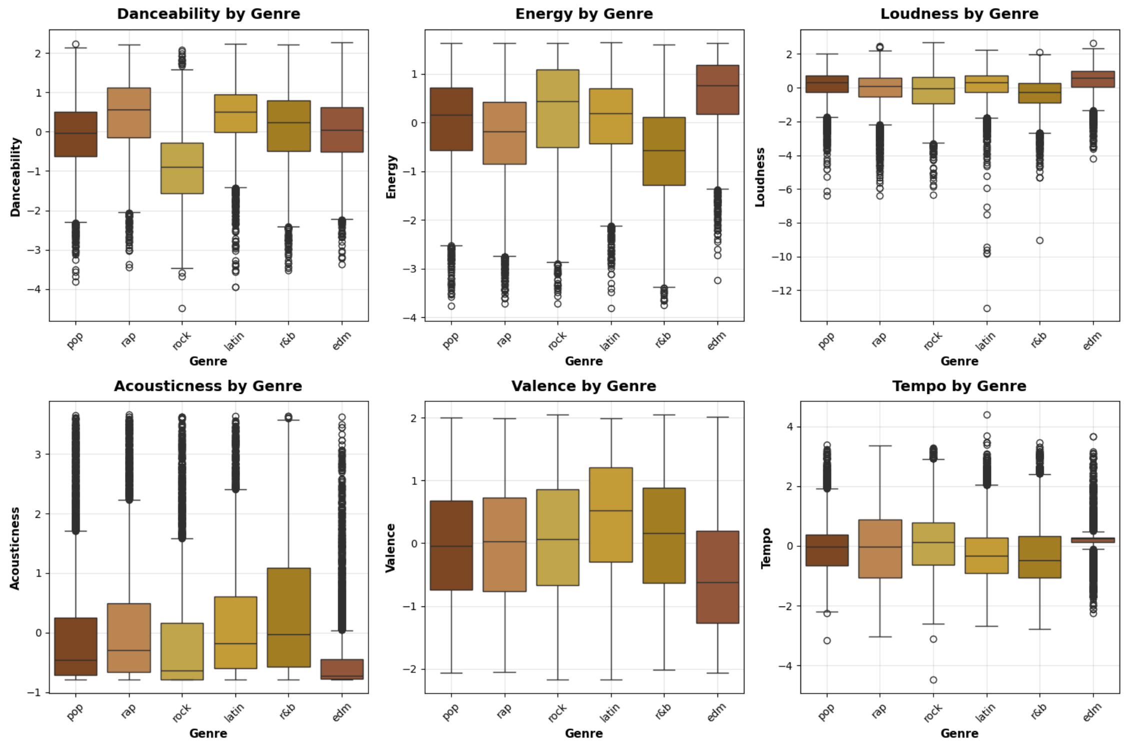
[38]:
```
# Call the plotting functions to generate and save plots
create_genre_distribution_plot(data)
create_correlation_matrix(data)
create_feature_boxplots(data)
create_radar_chart(data)
```




```
 Plot saved to: /content/drive/MyDrive/music-genre-
classification/images/eda/genre_distribution.png
```




```
Plot saved to: /content/drive/MyDrive/music-genre-
classification/images/eda/correlation_matrix.png
```




```
 Plot saved to: /content/drive/MyDrive/music-genre-
classification/images/eda/feature_distributions.png

 Interactive plot saved to: /content/drive/MyDrive/music-genre-
classification/images/eda/radar_chart.html
```

[31]:
```
%pip install -U kaleido
```

```
Requirement already satisfied: kaleido in /usr/local/lib/python3.11/dist-
packages (1.0.0)
Requirement already satisfied: choreographer>=1.0.5 in
/usr/local/lib/python3.11/dist-packages (from kaleido) (1.0.9)
Requirement already satisfied: logistro>=1.0.8 in
/usr/local/lib/python3.11/dist-packages (from kaleido) (1.1.0)
Requirement already satisfied: orjson>=3.10.15 in
/usr/local/lib/python3.11/dist-packages (from kaleido) (3.11.1)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-
packages (from kaleido) (25.0)
Requirement already satisfied: simplejson>=3.19.3 in
/usr/local/lib/python3.11/dist-packages (from choreographer>=1.0.5->kaleido)
(3.20.1)
```

02_Modeling

August 8, 2025

##Project: Music Genre Classification using Machine Learning

Niyat Kahsay & Marwah Faraj Summer 2025

Description:

This project aims to automatically classify songs into genres based on audio features provided in the Spotify 1.2M Songs Dataset. The workflow includes data exploration, preprocessing, model building, evaluation, and visualization.

Purpose:

Apply supervised learning techniques on real-world audio data Explore audio feature-based genre classification Build a portfolio-ready project demonstrating practical machine learning skills

#Data Preparation

0.1 Import Libraries

[1]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.graph_objects as go
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import confusion_matrix
from imblearn.over_sampling import SMOTE
from xgboost import XGBClassifier
from sklearn.metrics import (accuracy_score, confusion_matrix,
                             classification_report, f1_score)
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import
 ↪cross_val_score,GridSearchCV,train_test_split
from sklearn.svm import SVC
```

```
from sklearn.decomposition import PCA
import numpy as np
import time
import pandas as pd
import plotly.express as px
from scipy.stats import mode
import warnings

warnings.filterwarnings("ignore", category=FutureWarning)
```

0.2 Load Data

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[2]:
```
import pandas as pd
# data = pd.read_csv('/content/drive/MyDrive/music-genre-classification/data/
 ↪spotify_songs.csv')
data= pd.read_csv('/Users/marwahfaraj/Desktop/ms_degree_application_and_doc/
 ↪final_projects/504_final_project/music-genre-classification/data/
 ↪spotify_songs.csv')
data.head()
```

[2]:
```
                 track_id                                         track_name  \
0  6f807x0ima9a1j3VPbc7VN  I Don't Care (with Justin Bieber) - Loud Luxur…
1  0r7CVbZTWZgbTCYdfa2P31                    Memories - Dillon Francis Remix
2  1z1Hg7Vb0AhHDiEmnDE79l                    All the Time - Don Diablo Remix
3  75FpbthrwQmzHlBJLuGdC7                  Call You Mine - Keanu Silva Remix
4  1e8PAfcKUYoKkxPhrHqw4x            Someone You Loved - Future Humans Remix

       track_artist  track_popularity          track_album_id  \
0        Ed Sheeran                66  2oCs0DGTsRO98Gh5ZSl2Cx
1          Maroon 5                67  63rPSO264uRjW1X5E6cWv6
2      Zara Larsson                70  1HoSmj2eLcsrR0vE9gThr4
3  The Chainsmokers                60  1nqYsOef1yKKuGOVchbsk6
4     Lewis Capaldi                69  7m7vv9wlQ4i0LFuJiE2zsQ

                                     track_album_name track_album_release_date  \
0  I Don't Care (with Justin Bieber) [Loud Luxury…               2019-06-14
1                     Memories (Dillon Francis Remix)               2019-12-13
2                     All the Time (Don Diablo Remix)               2019-07-05
3                         Call You Mine - The Remixes               2019-07-19
4             Someone You Loved (Future Humans Remix)               2019-03-05
```

```
  playlist_name             playlist_id playlist_genre  … key  loudness  \
0     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   6    -2.634
1     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …  11    -4.969
2     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   1    -3.432
3     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   7    -3.778
4     Pop Remix  37i9dQZF1DXcZDD7cfEKhW            pop  …   1    -4.672

   mode  speechiness  acousticness  instrumentalness  liveness  valence  \
0     1       0.0583        0.1020          0.000000    0.0653    0.518
1     1       0.0373        0.0724          0.004210    0.3570    0.693
2     0       0.0742        0.0794          0.000023    0.1100    0.613
3     1       0.1020        0.0287          0.000009    0.2040    0.277
4     1       0.0359        0.0803          0.000000    0.0833    0.725

     tempo  duration_ms
0  122.036       194754
1   99.972       162600
2  124.008       176616
3  121.956       169093
4  123.976       189052

[5 rows x 23 columns]
```

#Data Exploration

[3]:
```
print(data.columns)
```

```
Index(['track_id', 'track_name', 'track_artist', 'track_popularity',
       'track_album_id', 'track_album_name', 'track_album_release_date',
       'playlist_name', 'playlist_id', 'playlist_genre', 'playlist_subgenre',
       'danceability', 'energy', 'key', 'loudness', 'mode', 'speechiness',
       'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo',
       'duration_ms'],
      dtype='object')
```

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32833 entries, 0 to 32832
Data columns (total 23 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   track_id                  32833 non-null  object
 1   track_name                32828 non-null  object
 2   track_artist              32828 non-null  object
 3   track_popularity          32833 non-null  int64
 4   track_album_id            32833 non-null  object
 5   track_album_name          32828 non-null  object
 6   track_album_release_date  32833 non-null  object
```

```
 7   playlist_name             32833 non-null  object
 8   playlist_id               32833 non-null  object
 9   playlist_genre            32833 non-null  object
 10  playlist_subgenre         32833 non-null  object
 11  danceability              32833 non-null  float64
 12  energy                    32833 non-null  float64
 13  key                       32833 non-null  int64
 14  loudness                  32833 non-null  float64
 15  mode                      32833 non-null  int64
 16  speechiness               32833 non-null  float64
 17  acousticness              32833 non-null  float64
 18  instrumentalness          32833 non-null  float64
 19  liveness                  32833 non-null  float64
 20  valence                   32833 non-null  float64
 21  tempo                     32833 non-null  float64
 22  duration_ms               32833 non-null  int64
dtypes: float64(9), int64(4), object(10)
memory usage: 5.8+ MB
```

[5]:
```
data.describe(include = "all")
```

[5]:

	track_id	track_name	track_artist	track_popularity
count	32833	32828	32828	32833.000000
unique	28356	23449	10692	NaN
top	7BKLCZ1jbUBVqRi2FVlTVw	Poison	Martin Garrix	NaN
freq	10	22	161	NaN
mean	NaN	NaN	NaN	42.477081
std	NaN	NaN	NaN	24.984074
min	NaN	NaN	NaN	0.000000
25%	NaN	NaN	NaN	24.000000
50%	NaN	NaN	NaN	45.000000
75%	NaN	NaN	NaN	62.000000
max	NaN	NaN	NaN	100.000000

	track_album_id	track_album_name	track_album_release_date
count	32833	32828	32833
unique	22545	19743	4530
top	5L1xcowSxwzFUSJzvyMp48	Greatest Hits	2020-01-10
freq	42	139	270
mean	NaN	NaN	NaN
std	NaN	NaN	NaN
min	NaN	NaN	NaN
25%	NaN	NaN	NaN
50%	NaN	NaN	NaN
75%	NaN	NaN	NaN
max	NaN	NaN	NaN

```
         playlist_name            playlist_id playlist_genre  …  \
count            32833                  32833          32833  …
unique             449                    471              6  …
top    Indie Poptimism  4JkkvMpVl4lSioqQjeAL0q            edm  …
freq               308                    247           6043  …
mean               NaN                    NaN            NaN  …
std                NaN                    NaN            NaN  …
min                NaN                    NaN            NaN  …
25%                NaN                    NaN            NaN  …
50%                NaN                    NaN            NaN  …
75%                NaN                    NaN            NaN  …
max                NaN                    NaN            NaN  …

                key      loudness          mode   speechiness  acousticness  \
count  32833.000000  32833.000000  32833.000000  32833.000000  32833.000000
unique          NaN           NaN           NaN           NaN           NaN
top             NaN           NaN           NaN           NaN           NaN
freq            NaN           NaN           NaN           NaN           NaN
mean       5.374471     -6.719499      0.565711      0.107068      0.175334
std        3.611657      2.988436      0.495671      0.101314      0.219633
min        0.000000    -46.448000      0.000000      0.000000      0.000000
25%        2.000000     -8.171000      0.000000      0.041000      0.015100
50%        6.000000     -6.166000      1.000000      0.062500      0.080400
75%        9.000000     -4.645000      1.000000      0.132000      0.255000
max       11.000000      1.275000      1.000000      0.918000      0.994000

        instrumentalness      liveness       valence         tempo  \
count       32833.000000  32833.000000  32833.000000  32833.000000
unique               NaN           NaN           NaN           NaN
top                  NaN           NaN           NaN           NaN
freq                 NaN           NaN           NaN           NaN
mean            0.084747      0.190176      0.510561    120.881132
std             0.224230      0.154317      0.233146     26.903624
min             0.000000      0.000000      0.000000      0.000000
25%             0.000000      0.092700      0.331000     99.960000
50%             0.000016      0.127000      0.512000    121.984000
75%             0.004830      0.248000      0.693000    133.918000
max             0.994000      0.996000      0.991000    239.440000

          duration_ms
count    32833.000000
unique            NaN
top               NaN
freq              NaN
mean    225799.811622
std      59834.006182
min       4000.000000
```

```
25%     187819.000000
50%     216000.000000
75%     253585.000000
max     517810.000000

[11 rows x 23 columns]
```

[6]:

```
# Feature engineering
data['track_album_release_year'] = pd.to_datetime(
      data['track_album_release_date'], errors='coerce').dt.year
data['track_album_release_year'].fillna(data['track_album_release_year'].
 ↪median(), inplace=True)
data['duration_s'] = data['duration_ms'] / 1000  # Convert to seconds
data.drop('duration_ms', axis=1, inplace=True)
```

[7]:

```
# Encode target variable
genre_encoder = LabelEncoder()
data['genre_encoded'] = genre_encoder.fit_transform(data['playlist_genre'])
```

[8]:

```
# Select relevant features
features = ['danceability', 'energy', 'key', 'loudness', 'mode',
            'speechiness', 'acousticness', 'instrumentalness',
            'liveness', 'valence', 'tempo', 'duration_s',
            'track_popularity', 'track_album_release_year']

X = data[features]
y = data['genre_encoded']
```

[9]:

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y)
```

[10]:

```
# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

[11]:

```
# Save the fitted scaler for deployment
import joblib
import os
scaler_save_path = '/Users/marwahfaraj/Desktop/ms_degree_application_and_doc/
 ↪final_projects/504_final_project/music-genre-classification/app/model/
 ↪standard_scaler.pkl'
os.makedirs(os.path.dirname(scaler_save_path), exist_ok=True)
joblib.dump(scaler, scaler_save_path)
print(f"StandardScaler saved to: {scaler_save_path}")
```

```
StandardScaler saved to: /Users/marwahfaraj/Desktop/ms_degree_application_and_do
```

```
c/final_projects/504_final_project/music-genre-
classification/app/model/standard_scaler.pkl
```

##Model Development

[12]:
```python
models = {
    'Random Forest': RandomForestClassifier(random_state=42),
    'Gradient Boosting': GradientBoostingClassifier(random_state=42),
    'XGBoost': XGBClassifier(random_state=42, use_label_encoder=False,␣
 ↪eval_metric='mlogloss'),
    'SVM': SVC(random_state=42, probability=True)
}
```

[13]:
```python
# Cross-validation evaluation
results = {}
for name, model in models.items():
    start_time = time.time()
    cv_scores = cross_val_score(model, X_train_scaled, y_train, cv=5,␣
 ↪scoring='accuracy')
    results[name] = {
        'cv_accuracy': np.mean(cv_scores),
        'cv_time': time.time() - start_time
    }
    print(f"{name} - Avg CV Accuracy: {np.mean(cv_scores):.4f} - Time:␣
 ↪{results[name]['cv_time']:.2f}s")
```

```
Random Forest - Avg CV Accuracy: 0.5747 - Time: 44.56s
Gradient Boosting - Avg CV Accuracy: 0.5715 - Time: 284.02s
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:03:59] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:04:02] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:04:05] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:04:11] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:04:14] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

```
XGBoost - Avg CV Accuracy: 0.5876 - Time: 18.55s
SVM - Avg CV Accuracy: 0.5548 - Time: 812.40s
```

[14]:
```python
# Select best model based on CV
best_model_name = max(results, key=lambda x: results[x]['cv_accuracy'])
print(f"\nBest model from CV: {best_model_name}")
```

```
Best model from CV: XGBoost
```

[15]:
```python
# Hyperparameter tuning for best model
# Best model: XGBoost
param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 5],
    'subsample': [0.8, 1.0]
}

grid_search = GridSearchCV(
    models['XGBoost'],
    param_grid,
    cv=3,
    scoring='accuracy',
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train_scaled, y_train)

# Get best model
best_model = grid_search.best_estimator_
print(f"Best parameters: {grid_search.best_params_}")
print(f"Best CV accuracy: {grid_search.best_score_:.4f}")
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:20:01] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

```
Best parameters: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 200,
```

```
'subsample': 0.8}
Best CV accuracy: 0.5883
```

##Model Evaluation

[16]:
```python
# Train final model
best_model.fit(X_train_scaled, y_train)

# Predictions
y_pred = best_model.predict(X_test_scaled)
y_proba = best_model.predict_proba(X_test_scaled)

# Evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')
class_report = classification_report(y_test, y_pred, target_names=genre_encoder.
 ↪classes_)

print("\n" + "="*50)
print(f"Final Model: {best_model_name}")
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Weighted F1 Score: {f1:.4f}")
print("\nClassification Report:")
print(class_report)
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:20:09] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)


==================================================
Final Model: XGBoost
Test Accuracy: 0.5959
Weighted F1 Score: 0.5940

Classification Report:
              precision    recall  f1-score   support

         edm       0.69      0.70      0.70      1209
       latin       0.53      0.48      0.51      1031
         pop       0.43      0.45      0.44      1102
         r&b       0.55      0.48      0.51      1086
         rap       0.61      0.67      0.64      1149
        rock       0.76      0.79      0.77       990

    accuracy                           0.60      6567
   macro avg       0.59      0.60      0.59      6567
```

```
weighted avg       0.59      0.60      0.59      6567
```

Low performance in 'latin' and 'pop. Will do, resampling

[17]:
```
from imblearn.over_sampling import SMOTE
# Apply SMOTE to training data
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_scaled,␣
 ↪y_train)
```

[18]:
```
grid_search_smote = GridSearchCV(
    models['XGBoost'],  # Same XGBoost model
    param_grid,
    cv=3,
    scoring='accuracy',
    n_jobs=-1,
    verbose=1
)

# Fit the model on the SMOTE-balanced data
grid_search_smote.fit(X_train_resampled, y_train_resampled)

# Predict on the original test set (not resampled!)
y_pred_smote = grid_search_smote.predict(X_test_scaled)
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:22:42] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

[19]:
```
from sklearn.metrics import accuracy_score, f1_score, classification_report

print("\n" + "="*50)
print("SMOTE Model Performance:")
print("Test Accuracy:", accuracy_score(y_test, y_pred_smote))
print("Weighted F1 Score:", f1_score(y_test, y_pred_smote, average='weighted'))
print("\nClassification Report:\n", classification_report(y_test, y_pred_smote))
```

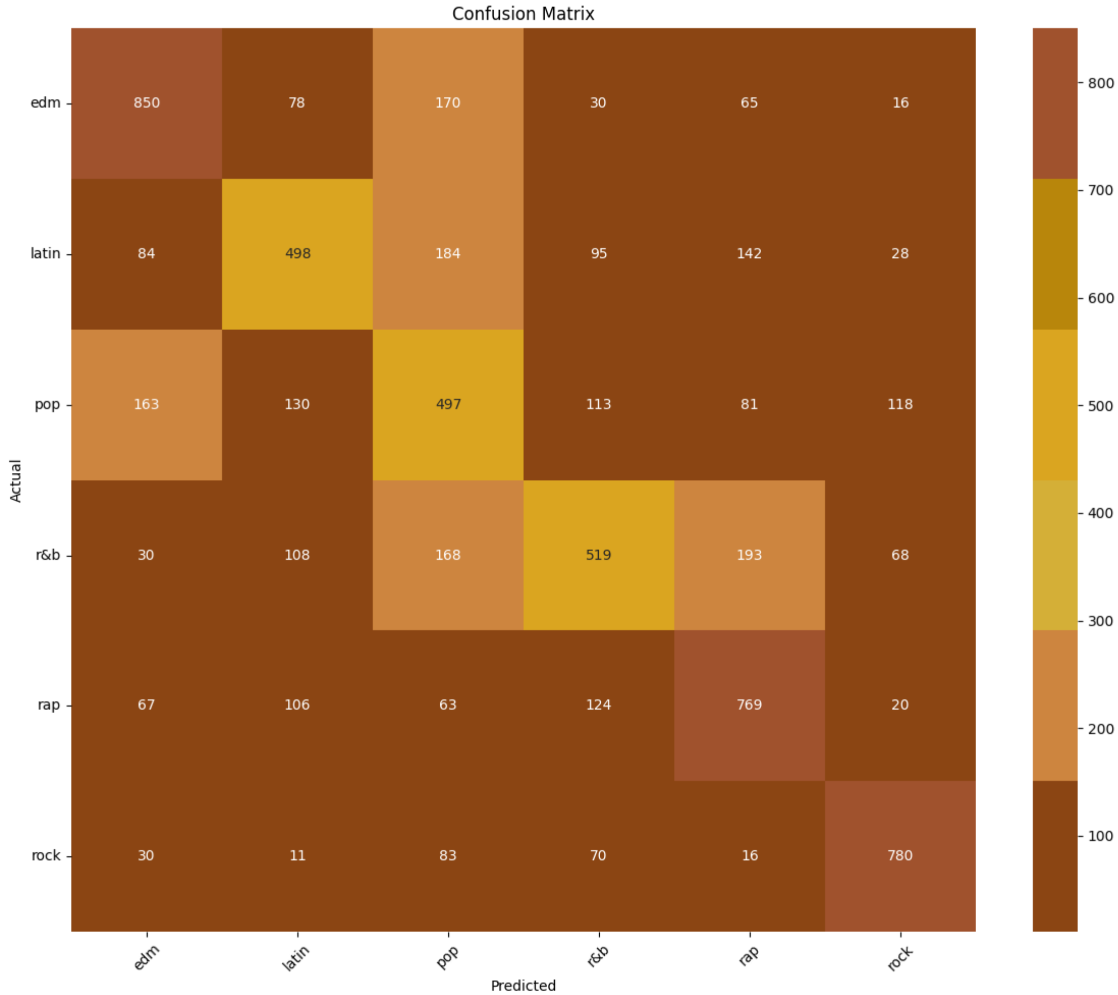
```
==================================================
SMOTE Model Performance:
Test Accuracy: 0.596771737475255
Weighted F1 Score: 0.5949522484302238

Classification Report:
               precision    recall  f1-score   support
```

```
           0       0.71      0.69      0.70      1209
           1       0.52      0.51      0.51      1031
           2       0.43      0.44      0.44      1102
           3       0.54      0.47      0.50      1086
           4       0.62      0.67      0.64      1149
           5       0.74      0.80      0.77       990

    accuracy                           0.60      6567
   macro avg       0.59      0.60      0.59      6567
weighted avg       0.59      0.60      0.59      6567
```

##But some results went down, so we are keeping the original model

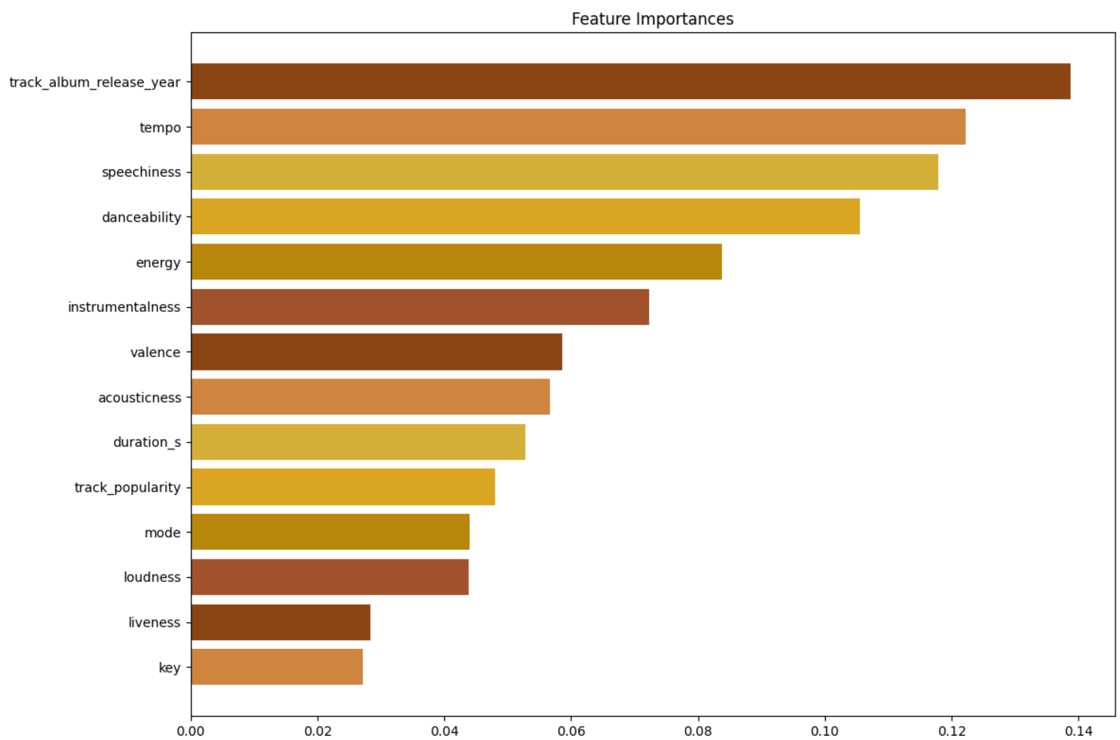
[49]:

```
import os
import matplotlib.pyplot as plt # Import pyplot
import seaborn as sns # Import seaborn if needed
import numpy as np # Import numpy if needed
from sklearn.metrics import confusion_matrix # Import confusion_matrix if needed


# Ensure the directory exists
if not os.path.exists('images/models'):
    os.makedirs('images/models')

# Set up white background and golden color scheme
plt.style.use('default') # Use default style for white background
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
↪'#A0522D'] # Adjusted golden palette


# Confusion matrix
plt.figure(figsize=(12, 10))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap=golden_palette,
              xticklabels=genre_encoder.classes_,
              yticklabels=genre_encoder.classes_)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()
plt.savefig('images/models/confusion_matrix.png')
plt.show()
```



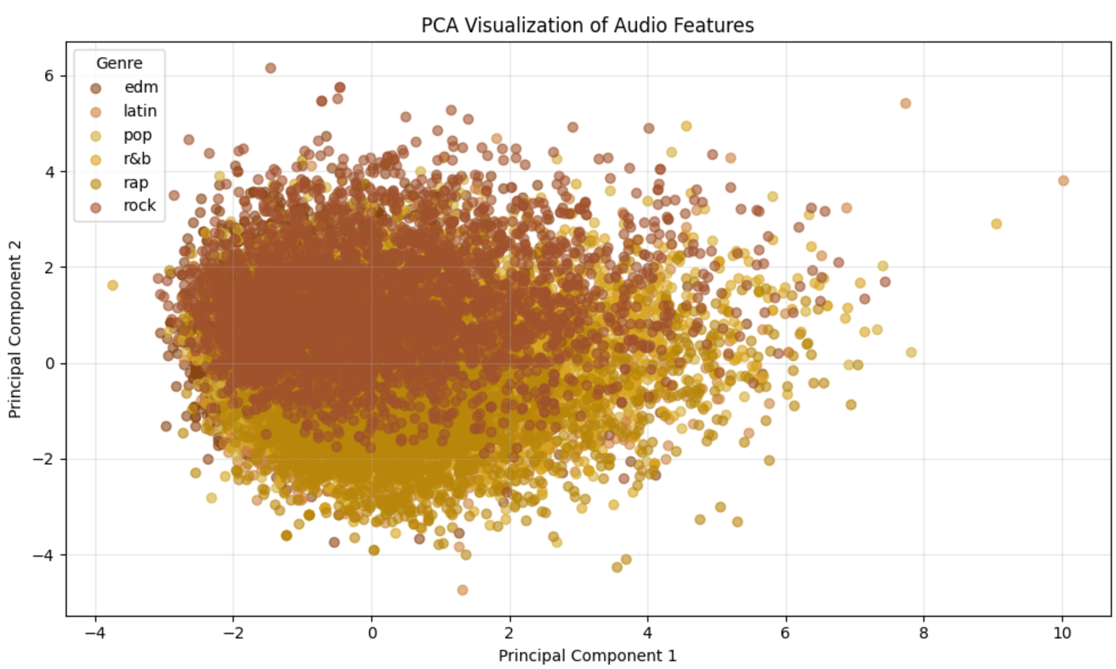
[21]:
```
import os
import matplotlib.pyplot as plt # Import pyplot
import numpy as np # Import numpy if needed

# Ensure the directory exists
if not os.path.exists('images/models'):
    os.makedirs('images/models')

# Set up white background and golden color scheme
plt.style.use('default') # Use default style for white background
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
 ↪'#A0522D'] # Adjusted golden palette


# Feature importance
if hasattr(best_model, 'feature_importances_'):
```

```python
    plt.figure(figsize=(12, 8))
    importances = best_model.feature_importances_
    indices = np.argsort(importances)[::-1]
    plt.title('Feature Importances')
    # Use all colors in the palette, repeating if necessary
    colors = np.tile(golden_palette, len(indices) // len(golden_palette) + 1)[:
↪len(indices)]
    plt.barh(range(len(indices)), importances[indices], align='center',␣
↪color=colors) # Changed color to use the palette
    plt.yticks(range(len(indices)), [features[i] for i in indices])
    plt.gca().invert_yaxis()
    plt.tight_layout()
    plt.savefig('images/models/feature_importances.png')
    plt.show()
```



[22]:
```python
import os
import matplotlib.pyplot as plt # Import pyplot
import numpy as np # Import numpy if needed
from sklearn.decomposition import PCA # Import PCA if needed

# Ensure the directory exists
if not os.path.exists('images/models'):
    os.makedirs('images/models')
```

```
# Set up white background and golden color scheme
plt.style.use('default') # Use default style for white background
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
↪'#A0522D'] # Adjusted golden palette


# PCA for dimensionality reduction (for visualization)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_train_scaled)

plt.figure(figsize=(10, 6))
for i, genre in enumerate(np.unique(y_train)):
    plt.scatter(X_pca[y_train == genre, 0],
                X_pca[y_train == genre, 1],
                label=genre_encoder.inverse_transform([genre])[0],
                alpha=0.6,
                color=golden_palette[i % len(golden_palette)])
plt.title('PCA Visualization of Audio Features')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Genre')
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig('images/models/pca_visualization.png')
plt.show()
```



0.3 Define the deep learning model

0.3.1 Subtask:

Generate code to define a suitable deep learning model architecture for music genre classification, considering the nature of your features. This might involve layers like Dense, Dropout, etc.

Reasoning: Define a deep learning model architecture for music genre classification using Keras Sequential model, dense layers with ReLU activation, dropout layers, and a final dense layer with softmax activation.

[23]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
import numpy as np

# Determine the number of unique genres
num_genres = len(np.unique(y_train))

# Define the deep learning model
dl_model = Sequential()

# Input layer and first hidden layer with dropout
dl_model.add(Dense(128, activation='relu', input_shape=(X_train_scaled.
 ↪shape[1],)))
dl_model.add(Dropout(0.3)) # Added dropout

# Second hidden layer with dropout
dl_model.add(Dense(64, activation='relu'))
dl_model.add(Dropout(0.3)) # Added dropout

# Output layer
dl_model.add(Dense(num_genres, activation='softmax'))

# Compile the model
dl_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy']) # Added compile

dl_model.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

```
Layer (type)                    Output Shape                 Param #

dense (Dense)                   (None, 128)                    1,920

dropout (Dropout)               (None, 128)                        0

dense_1 (Dense)                 (None, 64)                     8,256

dropout_1 (Dropout)             (None, 64)                         0

dense_2 (Dense)                 (None, 6)                        390
```

```
Total params: 10,566 (41.27 KB)

Trainable params: 10,566 (41.27 KB)

Non-trainable params: 0 (0.00 B)
```

0.4 Compile the model

0.4.1 Subtask:

Add code to compile the deep learning model with an appropriate optimizer, loss function, and metrics.

Reasoning: Compile the deep learning model with the specified optimizer, loss function, and metrics.

[24]:
```
dl_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])
```

0.5 Train the model

0.5.1 Subtask:

Generate code to train the deep learning model on your scaled training data, including considerations for epochs, batch size, and validation data.

Reasoning: Train the deep learning model on the scaled training data with specified epochs, batch size, and validation split.

[25]:
```
from tensorflow.keras.callbacks import EarlyStopping # Import EarlyStopping

# Instantiate EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=15,␣
↪restore_best_weights=True) # Increased patience
```

```python
history = dl_model.fit(X_train_scaled, y_train,
                       epochs=50,
                       batch_size=64,
                       validation_split=0.2,
                       callbacks=[early_stopping]) # Added EarlyStopping␣
 ↪callback
```

```
Epoch 1/50
329/329          3s 4ms/step -
accuracy: 0.3627 - loss: 1.5625 - val_accuracy: 0.5010 - val_loss: 1.2921
Epoch 2/50
329/329          2s 3ms/step -
accuracy: 0.4786 - loss: 1.3487 - val_accuracy: 0.5190 - val_loss: 1.2530
Epoch 3/50
329/329          1s 3ms/step -
accuracy: 0.4942 - loss: 1.3086 - val_accuracy: 0.5259 - val_loss: 1.2343
Epoch 4/50
329/329          1s 3ms/step -
accuracy: 0.5067 - loss: 1.2884 - val_accuracy: 0.5306 - val_loss: 1.2171
Epoch 5/50
329/329          1s 3ms/step -
accuracy: 0.5164 - loss: 1.2632 - val_accuracy: 0.5344 - val_loss: 1.2060
Epoch 6/50
329/329          1s 3ms/step -
accuracy: 0.5224 - loss: 1.2604 - val_accuracy: 0.5362 - val_loss: 1.2024
Epoch 7/50
329/329          1s 3ms/step -
accuracy: 0.5287 - loss: 1.2374 - val_accuracy: 0.5430 - val_loss: 1.1877
Epoch 8/50
329/329          1s 3ms/step -
accuracy: 0.5331 - loss: 1.2301 - val_accuracy: 0.5476 - val_loss: 1.1807
Epoch 9/50
329/329          1s 3ms/step -
accuracy: 0.5326 - loss: 1.2252 - val_accuracy: 0.5497 - val_loss: 1.1796
Epoch 10/50
329/329          2s 4ms/step -
accuracy: 0.5449 - loss: 1.2090 - val_accuracy: 0.5495 - val_loss: 1.1725
Epoch 11/50
329/329          1s 4ms/step -
accuracy: 0.5415 - loss: 1.2120 - val_accuracy: 0.5525 - val_loss: 1.1675
Epoch 12/50
329/329          2s 3ms/step -
accuracy: 0.5428 - loss: 1.2024 - val_accuracy: 0.5539 - val_loss: 1.1651
Epoch 13/50
329/329          1s 3ms/step -
accuracy: 0.5402 - loss: 1.2043 - val_accuracy: 0.5586 - val_loss: 1.1566
Epoch 14/50
```

```
329/329            1s 3ms/step -
accuracy: 0.5409 - loss: 1.2005 - val_accuracy: 0.5594 - val_loss: 1.1583
Epoch 15/50
329/329            1s 3ms/step -
accuracy: 0.5453 - loss: 1.1941 - val_accuracy: 0.5575 - val_loss: 1.1530
Epoch 16/50
329/329            1s 3ms/step -
accuracy: 0.5465 - loss: 1.1893 - val_accuracy: 0.5638 - val_loss: 1.1497
Epoch 17/50
329/329            1s 3ms/step -
accuracy: 0.5483 - loss: 1.1836 - val_accuracy: 0.5634 - val_loss: 1.1478
Epoch 18/50
329/329            1s 3ms/step -
accuracy: 0.5501 - loss: 1.1745 - val_accuracy: 0.5619 - val_loss: 1.1464
Epoch 19/50
329/329            1s 3ms/step -
accuracy: 0.5529 - loss: 1.1725 - val_accuracy: 0.5664 - val_loss: 1.1402
Epoch 20/50
329/329            1s 3ms/step -
accuracy: 0.5531 - loss: 1.1853 - val_accuracy: 0.5668 - val_loss: 1.1439
Epoch 21/50
329/329            2s 4ms/step -
accuracy: 0.5511 - loss: 1.1853 - val_accuracy: 0.5613 - val_loss: 1.1364
Epoch 22/50
329/329            1s 4ms/step -
accuracy: 0.5591 - loss: 1.1629 - val_accuracy: 0.5668 - val_loss: 1.1399
Epoch 23/50
329/329            2s 3ms/step -
accuracy: 0.5608 - loss: 1.1638 - val_accuracy: 0.5649 - val_loss: 1.1331
Epoch 24/50
329/329            1s 3ms/step -
accuracy: 0.5642 - loss: 1.1627 - val_accuracy: 0.5649 - val_loss: 1.1336
Epoch 25/50
329/329            1s 3ms/step -
accuracy: 0.5619 - loss: 1.1544 - val_accuracy: 0.5683 - val_loss: 1.1359
Epoch 26/50
329/329            1s 3ms/step -
accuracy: 0.5592 - loss: 1.1644 - val_accuracy: 0.5649 - val_loss: 1.1355
Epoch 27/50
329/329            1s 3ms/step -
accuracy: 0.5655 - loss: 1.1546 - val_accuracy: 0.5685 - val_loss: 1.1348
Epoch 28/50
329/329            1s 3ms/step -
accuracy: 0.5582 - loss: 1.1572 - val_accuracy: 0.5653 - val_loss: 1.1334
Epoch 29/50
329/329            1s 3ms/step -
accuracy: 0.5634 - loss: 1.1465 - val_accuracy: 0.5708 - val_loss: 1.1299
Epoch 30/50
```

```
329/329              1s 3ms/step -
accuracy: 0.5592 - loss: 1.1551 - val_accuracy: 0.5697 - val_loss: 1.1307
Epoch 31/50
329/329              1s 4ms/step -
accuracy: 0.5651 - loss: 1.1452 - val_accuracy: 0.5655 - val_loss: 1.1298
Epoch 32/50
329/329              1s 4ms/step -
accuracy: 0.5672 - loss: 1.1495 - val_accuracy: 0.5676 - val_loss: 1.1288
Epoch 33/50
329/329              2s 3ms/step -
accuracy: 0.5653 - loss: 1.1581 - val_accuracy: 0.5672 - val_loss: 1.1330
Epoch 34/50
329/329              1s 3ms/step -
accuracy: 0.5631 - loss: 1.1513 - val_accuracy: 0.5666 - val_loss: 1.1301
Epoch 35/50
329/329              1s 3ms/step -
accuracy: 0.5574 - loss: 1.1558 - val_accuracy: 0.5649 - val_loss: 1.1287
Epoch 36/50
329/329              1s 3ms/step -
accuracy: 0.5657 - loss: 1.1498 - val_accuracy: 0.5668 - val_loss: 1.1279
Epoch 37/50
329/329              1s 3ms/step -
accuracy: 0.5659 - loss: 1.1372 - val_accuracy: 0.5641 - val_loss: 1.1285
Epoch 38/50
329/329              1s 3ms/step -
accuracy: 0.5701 - loss: 1.1474 - val_accuracy: 0.5689 - val_loss: 1.1272
Epoch 39/50
329/329              1s 3ms/step -
accuracy: 0.5590 - loss: 1.1509 - val_accuracy: 0.5706 - val_loss: 1.1248
Epoch 40/50
329/329              1s 3ms/step -
accuracy: 0.5577 - loss: 1.1445 - val_accuracy: 0.5685 - val_loss: 1.1272
Epoch 41/50
329/329              1s 3ms/step -
accuracy: 0.5662 - loss: 1.1495 - val_accuracy: 0.5727 - val_loss: 1.1268
Epoch 42/50
329/329              2s 4ms/step -
accuracy: 0.5671 - loss: 1.1349 - val_accuracy: 0.5693 - val_loss: 1.1243
Epoch 43/50
329/329              1s 4ms/step -
accuracy: 0.5704 - loss: 1.1349 - val_accuracy: 0.5660 - val_loss: 1.1264
Epoch 44/50
329/329              2s 3ms/step -
accuracy: 0.5715 - loss: 1.1448 - val_accuracy: 0.5714 - val_loss: 1.1249
Epoch 45/50
329/329              1s 3ms/step -
accuracy: 0.5670 - loss: 1.1305 - val_accuracy: 0.5706 - val_loss: 1.1290
Epoch 46/50
```

```
329/329                 1s 3ms/step -
accuracy: 0.5779 - loss: 1.1321 - val_accuracy: 0.5721 - val_loss: 1.1230
Epoch 47/50
329/329                 1s 3ms/step -
accuracy: 0.5666 - loss: 1.1420 - val_accuracy: 0.5683 - val_loss: 1.1264
Epoch 48/50
329/329                 1s 3ms/step -
accuracy: 0.5656 - loss: 1.1372 - val_accuracy: 0.5664 - val_loss: 1.1226
Epoch 49/50
329/329                 1s 3ms/step -
accuracy: 0.5757 - loss: 1.1244 - val_accuracy: 0.5699 - val_loss: 1.1248
Epoch 50/50
329/329                 1s 3ms/step -
accuracy: 0.5687 - loss: 1.1366 - val_accuracy: 0.5666 - val_loss: 1.1280
```

0.6 Evaluate the deep learning model

0.6.1 Subtask:

Add code to evaluate the trained deep learning model on the test set and print out relevant metrics like accuracy, F1-score, and a classification report.

Reasoning: Evaluate the trained deep learning model on the scaled test data to assess its performance using accuracy, weighted F1-score, and a classification report.

[26]:
```
from sklearn.metrics import accuracy_score, f1_score, classification_report
import numpy as np

# Evaluate the model on the test data
loss, accuracy = dl_model.evaluate(X_test_scaled, y_test, verbose=0)
print(f"Deep Learning Model Test Accuracy: {accuracy:.4f}")

# Make predictions and calculate F1-score and classification report
y_pred_dl_proba = dl_model.predict(X_test_scaled)
y_pred_dl = np.argmax(y_pred_dl_proba, axis=1)

f1_dl = f1_score(y_test, y_pred_dl, average='weighted')
class_report_dl = classification_report(y_test, y_pred_dl,
 ↪target_names=genre_encoder.classes_)

print(f"Deep Learning Model Weighted F1 Score: {f1_dl:.4f}")
print("\nDeep Learning Model Classification Report:")
print(class_report_dl)
```

```
Deep Learning Model Test Accuracy: 0.5651
206/206                 0s 1ms/step
Deep Learning Model Weighted F1 Score: 0.5585

Deep Learning Model Classification Report:
```

```
              precision    recall  f1-score   support

         edm       0.69      0.67      0.68      1209
       latin       0.54      0.40      0.46      1031
         pop       0.40      0.42      0.41      1102
         r&b       0.54      0.40      0.46      1086
         rap       0.54      0.71      0.61      1149
        rock       0.67      0.78      0.72       990

    accuracy                           0.57      6567
   macro avg       0.56      0.56      0.56      6567
weighted avg       0.56      0.57      0.56      6567
```

0.7 Compare models

0.7.1 Subtask:

Add a markdown section to compare the performance of the deep learning model with the traditional machine learning models you've already evaluated.

Reasoning: Create a markdown section to compare the performance of the deep learning model with the traditional machine learning models using the evaluation metrics obtained.

0.8 Visualize deep learning results

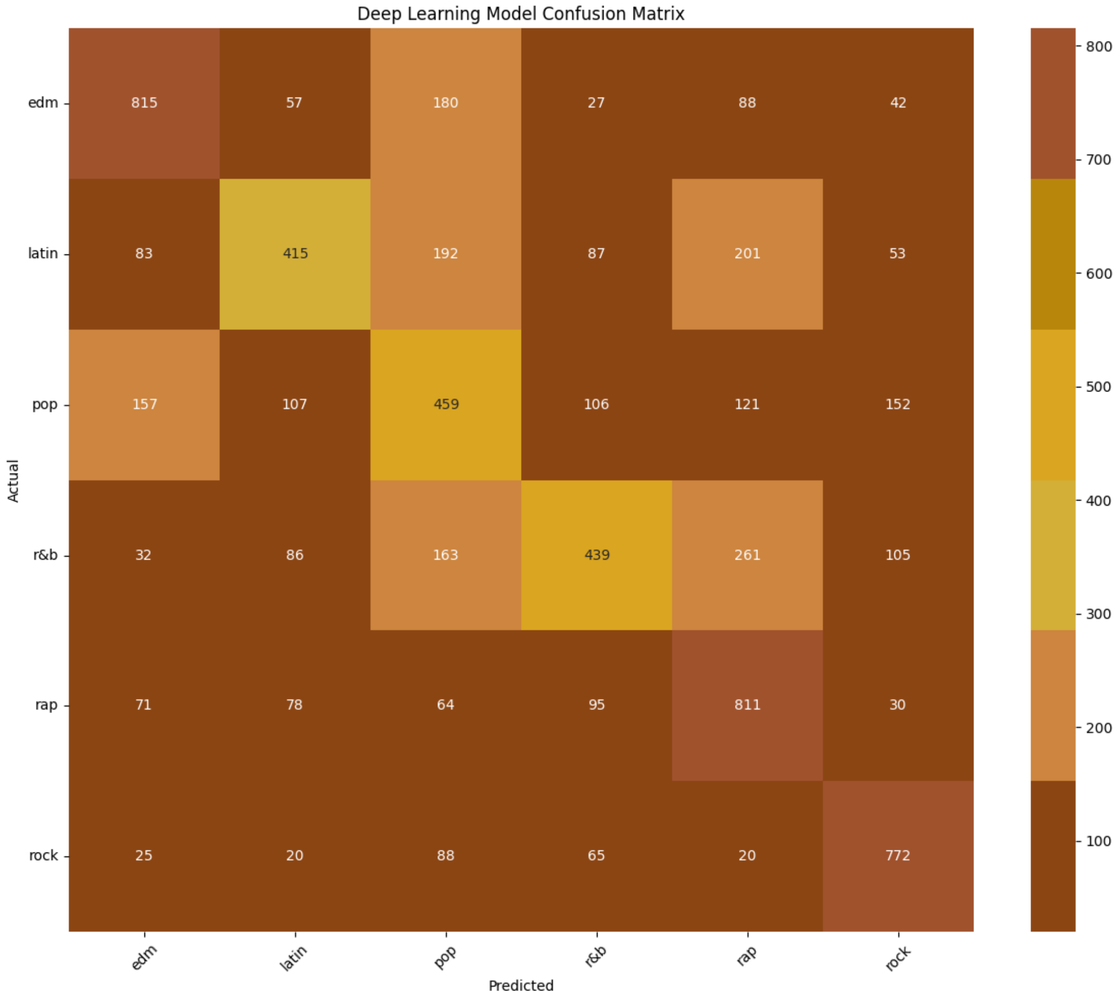
0.8.1 Subtask:

Generate code to visualize the performance of the deep learning model, such as a confusion matrix or training history plots.

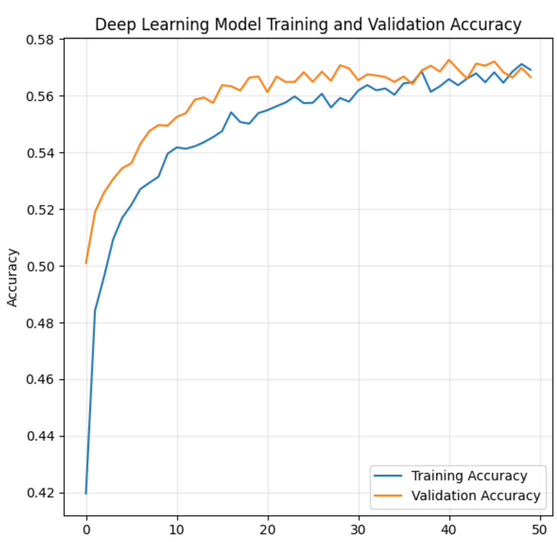
Reasoning: Generate code to visualize the deep learning model's performance using a confusion matrix and training history plots to gain insights into its predictions and training progress.

[51]:
```python
import os
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
import numpy as np

# Ensure the directory exists
if not os.path.exists('images/models'):
    os.makedirs('images/models')

# Set up white background and golden color scheme
plt.style.use('default')
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
 ↪'#A0522D']

# Confusion matrix for Deep Learning Model
```

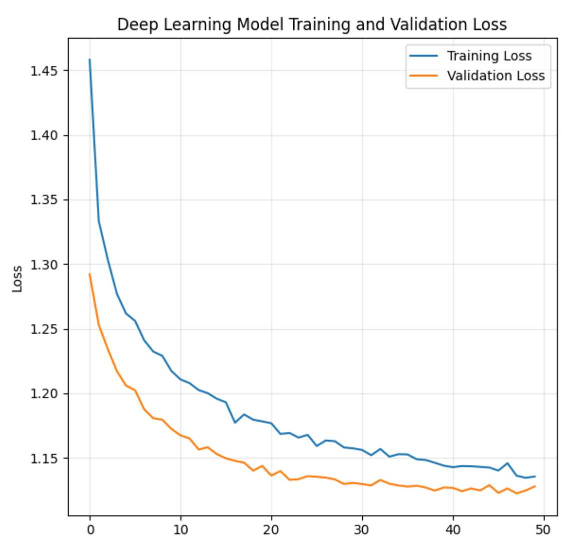
```python
plt.figure(figsize=(12, 10))
cm_dl = confusion_matrix(y_test, y_pred_dl)
sns.heatmap(cm_dl, annot=True, fmt='d', cmap=golden_palette,
            xticklabels=genre_encoder.classes_,
            yticklabels=genre_encoder.classes_)
plt.title('Deep Learning Model Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()
plt.savefig('images/models/dl_confusion_matrix.png')
plt.show()

# Plot training history (accuracy and loss)
plt.figure(figsize=(12, 6))

# Plot accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Deep Learning Model Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(alpha=0.3)

# Plot loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Deep Learning Model Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(alpha=0.3)

plt.tight_layout()
plt.savefig('images/models/dl_training_history.png')
plt.show()
```



Deep Learning Model Confusion Matrix
Actual
Predicted
edm
latin
pop
r&b
rap
rock
815
57
180
27
88
42
83
415
192
87
201
53
157
107
459
106
121
152
32
86
163
439
261
105
71
78
64
95
811
30
25
20
88
65
20
772
100
200
300
400
500
600
700
800



Deep Learning Model Training and Validation Accuracy
Accuracy
Epoch
Training Accuracy
Validation Accuracy



Deep Learning Model Training and Validation Loss
Loss
Epoch
Training Loss
Validation Loss

1 Task

Integrate PCA into the model development process by applying it before training both traditional machine learning models (like XGBoost) and a deep learning model. Evaluate the performance of all models (trained on original and PCA-transformed data), compare their results, and visualize the outcomes using the specified plotting style. Finally, summarize the findings and conclude the project.

1.1 Relocate and modify pca

1.1.1 Subtask:

Move the PCA code to a new section before model development and modify it to return the PCA-transformed training and test data.

Reasoning: Create a new markdown cell titled "Dimensionality Reduction (PCA)" before the "Model Development" section to introduce the PCA step.

[28]:
```
# Create a new markdown cell with the specified title
```

Reasoning: Copy the existing PCA code, modify it to perform PCA on the scaled training and test data, and store the results in new variables. Remove the plotting code as per the instructions.

[29]:
```
from sklearn.decomposition import PCA

# Apply PCA for dimensionality reduction
pca = PCA(n_components=0.95)  # Retain 95% of variance
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

print(f"Original number of features: {X_train_scaled.shape[1]}")
print(f"Reduced number of features after PCA: {X_train_pca.shape[1]}")
```

```
Original number of features: 14
Reduced number of features after PCA: 12
```

1.2 Train traditional ml models with pca

1.2.1 Subtask:

Train the selected traditional machine learning models (like XGBoost) using the PCA-transformed training data.

Reasoning: Train the traditional machine learning models using the PCA-transformed training data.

[30]:
```
# Train traditional models on PCA-transformed data
models_pca = {}
```

```python
for name, model in models.items():
    start_time = time.time()
    model.fit(X_train_pca, y_train)
    models_pca[f'{name}_PCA'] = model
    print(f"{name} (PCA) - Training Time: {time.time() - start_time:.2f}s")
```

```
Random Forest (PCA) - Training Time: 15.89s
Gradient Boosting (PCA) - Training Time: 114.66s
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning:
[02:26:21] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

```
XGBoost (PCA) - Training Time: 5.36s
SVM (PCA) - Training Time: 242.56s
```

Reasoning: Evaluate the traditional machine learning models trained on PCA-transformed data using the PCA-transformed test data.

[31]:
```python
from sklearn.metrics import accuracy_score, f1_score, classification_report

# Evaluate traditional models on PCA-transformed test data
results_pca = {}
for name, model in models_pca.items():
    y_pred_pca = model.predict(X_test_pca)
    accuracy_pca = accuracy_score(y_test, y_pred_pca)
    f1_pca = f1_score(y_test, y_pred_pca, average='weighted')
    results_pca[name] = {
        'test_accuracy': accuracy_pca,
        'weighted_f1': f1_pca
    }
    print(f"\n{name} - Test Accuracy: {accuracy_pca:.4f}")
    print(f"{name} - Weighted F1 Score: {f1_pca:.4f}")
    # Print classification report for detailed analysis
    # print(f"\n{name} Classification Report:\n", classification_report(y_test,␣
 ↪y_pred_pca, target_names=genre_encoder.classes_))
```

```

Random Forest_PCA - Test Accuracy: 0.5167
Random Forest_PCA - Weighted F1 Score: 0.5127

Gradient Boosting_PCA - Test Accuracy: 0.5071
Gradient Boosting_PCA - Weighted F1 Score: 0.5055

XGBoost_PCA - Test Accuracy: 0.5187
XGBoost_PCA - Weighted F1 Score: 0.5159

SVM_PCA - Test Accuracy: 0.5381
```

```
SVM_PCA - Weighted F1 Score: 0.5350
```

1.3 Define and compile deep learning model for pca data

1.3.1 Subtask:

Define and compile a deep learning model suitable for the dimensionality of the PCA-transformed data.

Reasoning: Define a deep learning model architecture for music genre classification using Keras Sequential model, dense layers with ReLU activation, dropout layers, and a final dense layer with softmax activation, considering the PCA-transformed input shape. Compile the model with the specified optimizer, loss function, and metrics.

[32]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
import numpy as np

# Determine the number of unique genres
num_genres = len(np.unique(y_train))

# Define the deep learning model for PCA-transformed data
dl_model_pca = Sequential()

# Input layer and first hidden layer with dropout, using PCA-transformed input␣
↪shape
dl_model_pca.add(Dense(128, activation='relu', input_shape=(X_train_pca.
↪shape[1],)))
dl_model_pca.add(Dropout(0.3)) # Added dropout

# Second hidden layer with dropout
dl_model_pca.add(Dense(64, activation='relu'))
dl_model_pca.add(Dropout(0.3)) # Added dropout

# Output layer
dl_model_pca.add(Dense(num_genres, activation='softmax'))

# Compile the model
dl_model_pca.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy']) # Added compile

dl_model_pca.summary()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 128)	1,664
dropout_2 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 64)	8,256
dropout_3 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 6)	390

```
Total params: 10,310 (40.27 KB)

Trainable params: 10,310 (40.27 KB)

Non-trainable params: 0 (0.00 B)
```

1.4 Train deep learning model with pca

1.4.1 Subtask:

Train the deep learning model using the PCA-transformed training data.

Reasoning: Train the deep learning model on the PCA-transformed training data with specified epochs, batch size, and validation split, including the EarlyStopping callback.

[33]:
```
from tensorflow.keras.callbacks import EarlyStopping

# Instantiate EarlyStopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=15,
 ↪restore_best_weights=True)

# Train the deep learning model for PCA-transformed data
history_pca = dl_model_pca.fit(X_train_pca, y_train,
                        epochs=50,
                        batch_size=64,
                        validation_split=0.2,
                        callbacks=[early_stopping])
```

```
Epoch 1/50
329/329           2s 3ms/step -
```

```
accuracy: 0.3734 - loss: 1.5598 - val_accuracy: 0.4931 - val_loss: 1.3119
Epoch 2/50
329/329            2s 4ms/step -
accuracy: 0.4690 - loss: 1.3641 - val_accuracy: 0.5069 - val_loss: 1.2770
Epoch 3/50
329/329            2s 3ms/step -
accuracy: 0.4896 - loss: 1.3332 - val_accuracy: 0.5147 - val_loss: 1.2530
Epoch 4/50
329/329            1s 3ms/step -
accuracy: 0.4939 - loss: 1.2996 - val_accuracy: 0.5179 - val_loss: 1.2427
Epoch 5/50
329/329            1s 3ms/step -
accuracy: 0.5125 - loss: 1.2768 - val_accuracy: 0.5232 - val_loss: 1.2325
Epoch 6/50
329/329            1s 3ms/step -
accuracy: 0.5048 - loss: 1.2784 - val_accuracy: 0.5282 - val_loss: 1.2199
Epoch 7/50
329/329            1s 3ms/step -
accuracy: 0.5146 - loss: 1.2562 - val_accuracy: 0.5344 - val_loss: 1.2181
Epoch 8/50
329/329            1s 3ms/step -
accuracy: 0.5151 - loss: 1.2623 - val_accuracy: 0.5339 - val_loss: 1.2101
Epoch 9/50
329/329            1s 3ms/step -
accuracy: 0.5162 - loss: 1.2502 - val_accuracy: 0.5398 - val_loss: 1.2044
Epoch 10/50
329/329            1s 3ms/step -
accuracy: 0.5267 - loss: 1.2252 - val_accuracy: 0.5375 - val_loss: 1.1959
Epoch 11/50
329/329            1s 3ms/step -
accuracy: 0.5310 - loss: 1.2379 - val_accuracy: 0.5411 - val_loss: 1.1916
Epoch 12/50
329/329            1s 3ms/step -
accuracy: 0.5333 - loss: 1.2231 - val_accuracy: 0.5394 - val_loss: 1.1930
Epoch 13/50
329/329            1s 4ms/step -
accuracy: 0.5334 - loss: 1.2193 - val_accuracy: 0.5426 - val_loss: 1.1885
Epoch 14/50
329/329            1s 4ms/step -
accuracy: 0.5291 - loss: 1.2257 - val_accuracy: 0.5434 - val_loss: 1.1827
Epoch 15/50
329/329            2s 3ms/step -
accuracy: 0.5409 - loss: 1.2067 - val_accuracy: 0.5411 - val_loss: 1.1828
Epoch 16/50
329/329            1s 3ms/step -
accuracy: 0.5452 - loss: 1.1990 - val_accuracy: 0.5436 - val_loss: 1.1846
Epoch 17/50
329/329            1s 3ms/step -
```

```
accuracy: 0.5365 - loss: 1.2168 - val_accuracy: 0.5463 - val_loss: 1.1785
Epoch 18/50
329/329            1s 3ms/step -
accuracy: 0.5437 - loss: 1.2000 - val_accuracy: 0.5489 - val_loss: 1.1716
Epoch 19/50
329/329            1s 3ms/step -
accuracy: 0.5423 - loss: 1.1983 - val_accuracy: 0.5493 - val_loss: 1.1729
Epoch 20/50
329/329            1s 3ms/step -
accuracy: 0.5404 - loss: 1.1911 - val_accuracy: 0.5451 - val_loss: 1.1711
Epoch 21/50
329/329            1s 3ms/step -
accuracy: 0.5406 - loss: 1.1913 - val_accuracy: 0.5472 - val_loss: 1.1715
Epoch 22/50
329/329            1s 3ms/step -
accuracy: 0.5419 - loss: 1.2085 - val_accuracy: 0.5482 - val_loss: 1.1702
Epoch 23/50
329/329            2s 4ms/step -
accuracy: 0.5394 - loss: 1.1978 - val_accuracy: 0.5522 - val_loss: 1.1681
Epoch 24/50
329/329            1s 4ms/step -
accuracy: 0.5442 - loss: 1.1926 - val_accuracy: 0.5501 - val_loss: 1.1676
Epoch 25/50
329/329            2s 3ms/step -
accuracy: 0.5472 - loss: 1.1845 - val_accuracy: 0.5489 - val_loss: 1.1654
Epoch 26/50
329/329            1s 3ms/step -
accuracy: 0.5376 - loss: 1.1932 - val_accuracy: 0.5508 - val_loss: 1.1612
Epoch 27/50
329/329            1s 3ms/step -
accuracy: 0.5429 - loss: 1.1932 - val_accuracy: 0.5464 - val_loss: 1.1680
Epoch 28/50
329/329            1s 3ms/step -
accuracy: 0.5458 - loss: 1.1908 - val_accuracy: 0.5508 - val_loss: 1.1611
Epoch 29/50
329/329            1s 3ms/step -
accuracy: 0.5479 - loss: 1.1852 - val_accuracy: 0.5546 - val_loss: 1.1609
Epoch 30/50
329/329            1s 3ms/step -
accuracy: 0.5480 - loss: 1.1800 - val_accuracy: 0.5546 - val_loss: 1.1594
Epoch 31/50
329/329            1s 3ms/step -
accuracy: 0.5445 - loss: 1.1845 - val_accuracy: 0.5537 - val_loss: 1.1603
Epoch 32/50
329/329            1s 3ms/step -
accuracy: 0.5528 - loss: 1.1721 - val_accuracy: 0.5537 - val_loss: 1.1583
Epoch 33/50
329/329            1s 3ms/step -
```

```
accuracy: 0.5500 - loss: 1.1715 - val_accuracy: 0.5584 - val_loss: 1.1563
Epoch 34/50
329/329            2s 4ms/step -
accuracy: 0.5464 - loss: 1.1859 - val_accuracy: 0.5522 - val_loss: 1.1598
Epoch 35/50
329/329            2s 5ms/step -
accuracy: 0.5426 - loss: 1.1893 - val_accuracy: 0.5569 - val_loss: 1.1568
Epoch 36/50
329/329            2s 5ms/step -
accuracy: 0.5470 - loss: 1.1834 - val_accuracy: 0.5546 - val_loss: 1.1549
Epoch 37/50
329/329            2s 5ms/step -
accuracy: 0.5606 - loss: 1.1667 - val_accuracy: 0.5518 - val_loss: 1.1548
Epoch 38/50
329/329            1s 4ms/step -
accuracy: 0.5538 - loss: 1.1724 - val_accuracy: 0.5571 - val_loss: 1.1562
Epoch 39/50
329/329            1s 3ms/step -
accuracy: 0.5517 - loss: 1.1735 - val_accuracy: 0.5569 - val_loss: 1.1582
Epoch 40/50
329/329            1s 3ms/step -
accuracy: 0.5518 - loss: 1.1690 - val_accuracy: 0.5542 - val_loss: 1.1559
Epoch 41/50
329/329            1s 3ms/step -
accuracy: 0.5610 - loss: 1.1576 - val_accuracy: 0.5569 - val_loss: 1.1546
Epoch 42/50
329/329            1s 3ms/step -
accuracy: 0.5490 - loss: 1.1767 - val_accuracy: 0.5531 - val_loss: 1.1571
Epoch 43/50
329/329            1s 3ms/step -
accuracy: 0.5493 - loss: 1.1776 - val_accuracy: 0.5535 - val_loss: 1.1541
Epoch 44/50
329/329            1s 3ms/step -
accuracy: 0.5474 - loss: 1.1762 - val_accuracy: 0.5531 - val_loss: 1.1530
Epoch 45/50
329/329            1s 3ms/step -
accuracy: 0.5531 - loss: 1.1780 - val_accuracy: 0.5533 - val_loss: 1.1513
Epoch 46/50
329/329            1s 3ms/step -
accuracy: 0.5550 - loss: 1.1620 - val_accuracy: 0.5537 - val_loss: 1.1512
Epoch 47/50
329/329            1s 3ms/step -
accuracy: 0.5472 - loss: 1.1768 - val_accuracy: 0.5552 - val_loss: 1.1532
Epoch 48/50
329/329            2s 4ms/step -
accuracy: 0.5593 - loss: 1.1589 - val_accuracy: 0.5571 - val_loss: 1.1518
Epoch 49/50
329/329            1s 4ms/step -
```

```
accuracy: 0.5519 - loss: 1.1731 - val_accuracy: 0.5552 - val_loss: 1.1516
Epoch 50/50
329/329              2s 3ms/step -
accuracy: 0.5562 - loss: 1.1733 - val_accuracy: 0.5565 - val_loss: 1.1528
```

1.5 Evaluate deep learning model with PCA

1.5.1 Subtask:

Evaluate the performance of the deep learning model trained on PCA data and compare the results with previous deep learning evaluations.

Reasoning: Evaluate the deep learning model trained on PCA-transformed data using the PCA-transformed test data to assess its performance with accuracy, weighted F1-score, and a classification report, and compare with previous deep learning evaluations.

[34]:
```python
from sklearn.metrics import accuracy_score, f1_score, classification_report
import numpy as np

# Evaluate the deep learning model trained on PCA data
loss_dl_pca, accuracy_dl_pca = dl_model_pca.evaluate(X_test_pca, y_test,␣
 ↪verbose=0)
print(f"Deep Learning Model (PCA) Test Accuracy: {accuracy_dl_pca:.4f}")

# Make predictions and calculate F1-score and classification report for the DL␣
 ↪model with PCA
y_pred_dl_pca_proba = dl_model_pca.predict(X_test_pca)
y_pred_dl_pca = np.argmax(y_pred_dl_pca_proba, axis=1)

f1_dl_pca = f1_score(y_test, y_pred_dl_pca, average='weighted')
class_report_dl_pca = classification_report(y_test, y_pred_dl_pca,␣
 ↪target_names=genre_encoder.classes_)

print(f"Deep Learning Model (PCA) Weighted F1 Score: {f1_dl_pca:.4f}")
print("\nDeep Learning Model (PCA) Classification Report:")
print(class_report_dl_pca)
```

```
Deep Learning Model (PCA) Test Accuracy: 0.5555
206/206              0s 1ms/step
Deep Learning Model (PCA) Weighted F1 Score: 0.5509

Deep Learning Model (PCA) Classification Report:
              precision    recall  f1-score   support

         edm       0.67      0.66      0.67      1209
       latin       0.50      0.42      0.46      1031
         pop       0.39      0.39      0.39      1102
         r&b       0.52      0.43      0.47      1086
         rap       0.56      0.68      0.62      1149
```

```
        rock       0.66      0.73      0.70       990

    accuracy                           0.56      6567
   macro avg       0.55      0.55      0.55      6567
weighted avg       0.55      0.56      0.55      6567
```

1.6 Compare all model performances

1.6.1 Subtask:

Create a comprehensive comparison of all model performances (traditional ML on original data, traditional ML on PCA data, deep learning on original data, deep learning on PCA data).

Reasoning: Create a markdown section to compare the performance of all trained models, including traditional ML models on original and PCA data, and deep learning models on original and PCA data, using their respective evaluation metrics.

[35]:

```
import pandas as pd

# Create a dictionary to store the results
all_results = {
    'Traditional ML (Original)': results,
    'Traditional ML (PCA)': results_pca,
    'Deep Learning (Original)': {'Test Accuracy': accuracy, 'Weighted F1 Score':
 ↪ f1},
    'Deep Learning (PCA)': {'Test Accuracy': accuracy_dl_pca, 'Weighted F1␣
 ↪Score': f1_dl_pca}
}

# Convert results to a pandas DataFrame for easy comparison
comparison_data = []

for model_type, model_results in all_results.items():
    if 'Traditional ML' in model_type:
        for model_name, metrics in model_results.items():
            comparison_data.append({
                'Model Type': model_type,
                'Model Name': model_name,
                'Test Accuracy': metrics.get('test_accuracy', metrics.
 ↪get('cv_accuracy')), # Handle both keys
                'Weighted F1 Score': metrics.get('weighted_f1')
            })
    else:
         comparison_data.append({
            'Model Type': model_type,
            'Model Name': model_type, # Use model type as name for DL
            'Test Accuracy': model_results['Test Accuracy'],
            'Weighted F1 Score': model_results['Weighted F1 Score']
```

```
        })


comparison_df = pd.DataFrame(comparison_data)

# Display the comparison table
print("Model Performance Comparison:")
display(comparison_df.round(4))
```

```
Model Performance Comparison:
                    Model Type                Model Name  Test Accuracy  \
0  Traditional ML (Original)             Random Forest         0.5747
1  Traditional ML (Original)         Gradient Boosting         0.5715
2  Traditional ML (Original)                   XGBoost         0.5876
3  Traditional ML (Original)                       SVM         0.5548
4       Traditional ML (PCA)         Random Forest_PCA         0.5167
5       Traditional ML (PCA)     Gradient Boosting_PCA         0.5071
6       Traditional ML (PCA)               XGBoost_PCA         0.5187
7       Traditional ML (PCA)                   SVM_PCA         0.5381
8   Deep Learning (Original)  Deep Learning (Original)         0.5651
9        Deep Learning (PCA)       Deep Learning (PCA)         0.5555

   Weighted F1 Score
0                NaN
1                NaN
2                NaN
3                NaN
4             0.5127
5             0.5055
6             0.5159
7             0.5350
8             0.5940
9             0.5509
```

2 Task

Use AutoML (AutoGluon) to improve the model accuracy and compare its performance with previously trained models.

2.1 Install autogluon

2.1.1 Subtask:

Add code to install the AutoGluon library.

Reasoning: Install the AutoGluon library using pip.

[36]:
```
# %pip install --quiet autogluon
```

2.2 Train AutoGluon Model

2.2.1 Subtask:

Add code to train an AutoGluon TabularPredictor on your training data for music genre classification.

Reasoning: Train an AutoGluon TabularPredictor on the training data for music genre classification, specifying the label column and evaluation metric.

[37]:
```
from autogluon.tabular import TabularPredictor
import time

# Prepare data for AutoGluon
# AutoGluon works directly with pandas DataFrames, so we can use the original␣
↪X_train and y_train
# We need to combine X_train and y_train into a single DataFrame for AutoGluon
train_data = X_train.copy()
train_data['genre_encoded'] = y_train

# Specify the label column
label = 'genre_encoded'

# Initialize and train the AutoGluon predictor
start_time = time.time()
# Exclude NeuralNetTorch model
predictor = TabularPredictor(label=label, eval_metric='accuracy').
↪fit(train_data, excluded_model_types=['NN_TORCH'])
autogluon_train_time = time.time() - start_time

print(f"AutoGluon Model Training Time: {autogluon_train_time:.2f}s")
```

```
No path specified. Models will be saved in: "AutogluonModels/ag-20250807_023147"
Verbosity: 2 (Standard Logging)
=================== System Info ===================
AutoGluon Version:  1.4.0
Python Version:     3.11.13
Operating System:   Linux
Platform Machine:   x86_64
Platform Version:   #1 SMP PREEMPT_DYNAMIC Sun Mar 30 16:01:29 UTC 2025
CPU Count:          2
Memory Avail:       10.17 GB / 12.67 GB (80.3%)
Disk Space Avail:   56.75 GB / 107.72 GB (52.7%)
===================================================
No presets specified! To achieve strong results with AutoGluon, it is
recommended to use the available presets. Defaulting to `'medium'`…
        Recommended Presets (For more details refer to
https://auto.gluon.ai/stable/tutorials/tabular/tabular-essentials.html#presets):
        presets='extreme' : New in v1.4: Massively better than 'best' on
```

```
datasets <30000 samples by using new models meta-learned on https://tabarena.ai:
TabPFNv2, TabICL, Mitra, and TabM. Absolute best accuracy. Requires a GPU.
Recommended 64 GB CPU memory and 32+ GB GPU memory.
        presets='best'    : Maximize accuracy. Recommended for most users. Use
in competitions and benchmarks.
        presets='high'    : Strong accuracy with fast inference speed.
        presets='good'    : Good accuracy with very fast inference speed.
        presets='medium'  : Fast training time, ideal for initial prototyping.
Using hyperparameters preset: hyperparameters='default'
Beginning AutoGluon training …
AutoGluon will save models to "/content/AutogluonModels/ag-20250807_023147"
Train Data Rows:    26266
Train Data Columns: 14
Label Column:       genre_encoded
AutoGluon infers your prediction problem is: 'multiclass' (because dtype of
label-column == int, but few unique label-values observed).
        6 unique label values:  [1, 2, 3, 5, 0, 4]
        If 'multiclass' is not the correct problem_type, please manually specify
the problem_type parameter during Predictor init (You may specify problem_type
as one of: ['binary', 'multiclass', 'regression', 'quantile'])
Problem Type:       multiclass
Preprocessing data …
Train Data Class Count: 6
Using Feature Generators to preprocess the data …
Fitting AutoMLPipelineFeatureGenerator…
        Available Memory:                    10414.84 MB
        Train Data (Original)  Memory Usage: 2.81 MB (0.0% of available memory)
        Inferring data type of each feature based on column values. Set
feature_metadata_in to manually specify special dtypes of the features.
        Stage 1 Generators:
                Fitting AsTypeFeatureGenerator…
                        Note: Converting 1 features to boolean dtype as they
only contain 2 unique values.
        Stage 2 Generators:
                Fitting FillNaFeatureGenerator…
        Stage 3 Generators:
                Fitting IdentityFeatureGenerator…
        Stage 4 Generators:
                Fitting DropUniqueFeatureGenerator…
        Stage 5 Generators:
                Fitting DropDuplicatesFeatureGenerator…
        Types of features in original data (raw dtype, special dtypes):
                ('float', []) : 11 | ['danceability', 'energy', 'loudness',
'speechiness', 'acousticness', …]
                ('int', [])   :  3 | ['key', 'mode', 'track_popularity']
        Types of features in processed data (raw dtype, special dtypes):
                ('float', [])     : 11 | ['danceability', 'energy', 'loudness',
'speechiness', 'acousticness', …]
```

```
                ('int', [])       :  2 | ['key', 'track_popularity']
                ('int', ['bool']) :  1 | ['mode']
        0.1s = Fit runtime
        14 features in original data used to generate 14 features in processed
data.
        Train Data (Processed) Memory Usage: 2.63 MB (0.0% of available memory)
Data preprocessing and feature engineering runtime = 0.15s …
AutoGluon will gauge predictive performance using evaluation metric: 'accuracy'
        To change this, specify the eval_metric parameter of Predictor()
Automatically generating train/validation split with
holdout_frac=0.09518008071270845, Train Rows: 23766, Val Rows: 2500
User-specified model hyperparameters to be fit:
{
        'NN_TORCH': [{}],
        'GBM': [{'extra_trees': True, 'ag_args': {'name_suffix': 'XT'}}, {},
{'learning_rate': 0.03, 'num_leaves': 128, 'feature_fraction': 0.9,
'min_data_in_leaf': 3, 'ag_args': {'name_suffix': 'Large', 'priority': 0,
'hyperparameter_tune_kwargs': None}}],
        'CAT': [{}],
        'XGB': [{}],
        'FASTAI': [{}],
        'RF': [{'criterion': 'gini', 'ag_args': {'name_suffix': 'Gini',
'problem_types': ['binary', 'multiclass']}}, {'criterion': 'entropy', 'ag_args':
{'name_suffix': 'Entr', 'problem_types': ['binary', 'multiclass']}},
{'criterion': 'squared_error', 'ag_args': {'name_suffix': 'MSE',
'problem_types': ['regression', 'quantile']}}],
        'XT': [{'criterion': 'gini', 'ag_args': {'name_suffix': 'Gini',
'problem_types': ['binary', 'multiclass']}}, {'criterion': 'entropy', 'ag_args':
{'name_suffix': 'Entr', 'problem_types': ['binary', 'multiclass']}},
{'criterion': 'squared_error', 'ag_args': {'name_suffix': 'MSE',
'problem_types': ['regression', 'quantile']}}],
}
Excluded models: ['NN_TORCH'] (Specified by `excluded_model_types`)
Fitting 10 L1 models, fit_strategy="sequential" …
Fitting model: NeuralNetFastAI …
        Fitting with cpus=1, gpus=0, mem=0.0/10.0 GB
        0.5836   = Validation score   (accuracy)
        29.1s    = Training   runtime
        0.03s    = Validation runtime
Fitting model: LightGBMXT …
        Fitting with cpus=1, gpus=0, mem=0.0/9.9 GB
        0.588    = Validation score   (accuracy)
        12.41s   = Training   runtime
        0.67s    = Validation runtime
Fitting model: LightGBM …
        Fitting with cpus=1, gpus=0, mem=0.0/9.9 GB
        0.6024   = Validation score   (accuracy)
        14.19s   = Training   runtime
```

```
        0.84s    = Validation runtime
Fitting model: RandomForestGini …
        Fitting with cpus=2, gpus=0, mem=0.1/9.9 GB
        0.5772   = Validation score   (accuracy)
        26.68s   = Training   runtime
        0.55s    = Validation runtime
Fitting model: RandomForestEntr …
        Fitting with cpus=2, gpus=0, mem=0.1/9.7 GB
        0.5808   = Validation score   (accuracy)
        42.37s   = Training   runtime
        0.27s    = Validation runtime
Fitting model: CatBoost …
        Fitting with cpus=1, gpus=0
        0.5712   = Validation score   (accuracy)
        10.8s    = Training   runtime
        0.01s    = Validation runtime
Fitting model: ExtraTreesGini …
        Fitting with cpus=2, gpus=0, mem=0.1/9.6 GB
        0.5632   = Validation score   (accuracy)
        10.46s   = Training   runtime
        0.29s    = Validation runtime
Fitting model: ExtraTreesEntr …
        Fitting with cpus=2, gpus=0, mem=0.1/9.6 GB
        0.5608   = Validation score   (accuracy)
        9.54s    = Training   runtime
        0.3s     = Validation runtime
Fitting model: XGBoost …
        Fitting with cpus=1, gpus=0
        0.5904   = Validation score   (accuracy)
        16.21s   = Training   runtime
        0.34s    = Validation runtime
Fitting model: LightGBMLarge …
        Fitting with cpus=1, gpus=0, mem=0.1/9.6 GB
        0.5936   = Validation score   (accuracy)
        16.83s   = Training   runtime
        0.6s     = Validation runtime
Fitting model: WeightedEnsemble_L2 …
        Ensemble Weights: {'LightGBM': 1.0}
        0.6024   = Validation score   (accuracy)
        0.18s    = Training   runtime
        0.0s     = Validation runtime
AutoGluon training complete, total runtime = 220.37s … Best model:
WeightedEnsemble_L2 | Estimated inference throughput: 2979.7 rows/s (2500 batch
size)
TabularPredictor saved. To load, use: predictor =
TabularPredictor.load("/content/AutogluonModels/ag-20250807_023147")

AutoGluon Model Training Time: 221.53s
```

2.3 Evaluate AutoGluon Model

2.3.1 Subtask:

Add code to evaluate the trained AutoGluon model on the test data and print out relevant metrics like accuracy and F1-score.

Reasoning: Evaluate the trained AutoGluon model on the test data to assess its performance using accuracy and weighted F1-score.

[38]:
```python
# Evaluate the AutoGluon model on the test data
# AutoGluon's predictor object can directly predict on a DataFrame
test_data = X_test.copy()
y_pred_autogluon = predictor.predict(test_data)
y_proba_autogluon = predictor.predict_proba(test_data)

# Calculate evaluation metrics
accuracy_autogluon = accuracy_score(y_test, y_pred_autogluon)
f1_autogluon = f1_score(y_test, y_pred_autogluon, average='weighted')

print("\n" + "="*50)
print("AutoGluon Model Performance:")
print(f"Test Accuracy: {accuracy_autogluon:.4f}")
print(f"Weighted F1 Score: {f1_autogluon:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_autogluon,␣
↪target_names=genre_encoder.classes_))
```

```
==================================================
AutoGluon Model Performance:
Test Accuracy: 0.5838
Weighted F1 Score: 0.5825

Classification Report:
              precision    recall  f1-score   support

         edm       0.69      0.69      0.69      1209
       latin       0.53      0.47      0.50      1031
         pop       0.39      0.42      0.41      1102
         r&b       0.52      0.47      0.49      1086
         rap       0.61      0.67      0.64      1149
        rock       0.75      0.77      0.76       990

    accuracy                           0.58      6567
   macro avg       0.58      0.58      0.58      6567
weighted avg       0.58      0.58      0.58      6567
```

2.4 Compare all model performances

2.4.1 Subtask:

Create a comprehensive comparison of all model performances (traditional ML on original data, traditional ML on PCA data, deep learning on original data, deep learning on PCA data, and AutoGluon).

Reasoning: Create a comprehensive comparison of all model performances, including traditional ML models on original and PCA data, deep learning models on original and PCA data, and the AutoGluon model, using their respective evaluation metrics.

[39]:
```
import pandas as pd

# Initialize results and results_pca as empty dictionaries if they are not␣
↪defined
if 'results' not in locals():
    results = {}
if 'results_pca' not in locals():
    results_pca = {}

# Create a dictionary to store the results
all_results = {
    'Traditional ML (Original)': results,
    'Traditional ML (PCA)': results_pca,
}

# Add Deep Learning (Original) results if available
if 'accuracy' in locals() and 'f1' in locals():
    all_results['Deep Learning (Original)'] = {'Test Accuracy': accuracy,␣
↪'Weighted F1 Score': f1}

# Add Deep Learning (PCA) results if available
if 'accuracy_dl_pca' in locals() and 'f1_dl_pca' in locals():
    all_results['Deep Learning (PCA)'] = {'Test Accuracy': accuracy_dl_pca,␣
↪'Weighted F1 Score': f1_dl_pca}

# Add AutoGluon results if available
if 'accuracy_autogluon' in locals() and 'f1_autogluon' in locals():
    all_results['AutoGluon'] = {'Test Accuracy': accuracy_autogluon, 'Weighted␣
↪F1 Score': f1_autogluon}


# Convert results to a pandas DataFrame for easy comparison
comparison_data = []

for model_type, model_results in all_results.items():
    if 'Traditional ML' in model_type:
        for model_name, metrics in model_results.items():
```

```
            comparison_data.append({
                'Model Type': model_type,
                'Model Name': model_name,
                'Test Accuracy': metrics.get('test_accuracy', metrics.
↪get('cv_accuracy')), # Handle both keys
                'Weighted F1 Score': metrics.get('weighted_f1')
            })
    else:
        comparison_data.append({
            'Model Type': model_type,
            'Model Name': model_type, # Use model type as name for DL and␣
↪AutoGluon
            'Test Accuracy': model_results['Test Accuracy'],
            'Weighted F1 Score': model_results['Weighted F1 Score']
        })


comparison_df = pd.DataFrame(comparison_data)

# Display the comparison table
print("Model Performance Comparison:")
display(comparison_df.round(4))
```

```
Model Performance Comparison:

                  Model Type                 Model Name  Test Accuracy  \
0   Traditional ML (Original)             Random Forest         0.5747
1   Traditional ML (Original)         Gradient Boosting         0.5715
2   Traditional ML (Original)                   XGBoost         0.5876
3   Traditional ML (Original)                       SVM         0.5548
4        Traditional ML (PCA)         Random Forest_PCA         0.5167
5        Traditional ML (PCA)     Gradient Boosting_PCA         0.5071
6        Traditional ML (PCA)               XGBoost_PCA         0.5187
7        Traditional ML (PCA)                   SVM_PCA         0.5381
8     Deep Learning (Original)  Deep Learning (Original)        0.5651
9          Deep Learning (PCA)       Deep Learning (PCA)        0.5555
10                  AutoGluon                 AutoGluon         0.5838

    Weighted F1 Score
0                 NaN
1                 NaN
2                 NaN
3                 NaN
4              0.5127
5              0.5055
6              0.5159
7              0.5350
8              0.5940
```

```
9               0.5509
10              0.5825
```

3 Task

Train an AutoGluon model on PCA-transformed data, evaluate its performance, update the comparison table and visualization to include these results, and finally summarize the findings from all models.

3.1 Train autogluon with pca

Add a new section to train the AutoGluon `TabularPredictor` on the PCA-transformed training data (`X_train_pca`).

Reasoning: Prepare the PCA-transformed training data for AutoGluon and train the AutoGluon TabularPredictor on this data, measuring the training time.

[48]:
```python
# Apply PCA for dimensionality reduction
pca = PCA(n_components=0.95)  # Retain 95% of variance
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

print(f"Original number of features: {X_train_scaled.shape[1]}")
print(f"Reduced number of features after PCA: {X_train_pca.shape[1]}")

# Convert PCA-transformed data and target variable to DataFrames with original␣
↪indices
X_train_pca_df = pd.DataFrame(X_train_pca, index=X_train.index)
y_train_df = pd.DataFrame(y_train, index=X_train.index,␣
↪columns=['genre_encoded'])

# Combine features and target
train_data_pca_combined = pd.concat([X_train_pca_df, y_train_df], axis=1)

# Drop rows with any non-finite values in the combined DataFrame
train_data_pca_cleaned = train_data_pca_combined.dropna().copy()

# Specify the label column
label = 'genre_encoded'

# Initialize and train the AutoGluon predictor on cleaned PCA data
start_time = time.time()
predictor_pca = TabularPredictor(label=label, eval_metric='accuracy').
↪fit(train_data_pca_cleaned, excluded_model_types=['NN_TORCH'])
autogluon_pca_train_time = time.time() - start_time

print(f"AutoGluon Model (PCA) Training Time: {autogluon_pca_train_time:.2f}s")
```

```
No path specified. Models will be saved in: "AutogluonModels/ag-20250807_025729"
Verbosity: 2 (Standard Logging)
=================== System Info ===================
AutoGluon Version:  1.4.0
Python Version:     3.11.13
Operating System:   Linux
Platform Machine:   x86_64
Platform Version:   #1 SMP PREEMPT_DYNAMIC Sun Mar 30 16:01:29 UTC 2025
CPU Count:          2
Memory Avail:       9.35 GB / 12.67 GB (73.8%)
Disk Space Avail:   47.02 GB / 107.72 GB (43.7%)
===================================================
No presets specified! To achieve strong results with AutoGluon, it is
recommended to use the available presets. Defaulting to `'medium'`…
        Recommended Presets (For more details refer to
https://auto.gluon.ai/stable/tutorials/tabular/tabular-essentials.html#presets):
        presets='extreme' : New in v1.4: Massively better than 'best' on
datasets <30000 samples by using new models meta-learned on https://tabarena.ai:
TabPFNv2, TabICL, Mitra, and TabM. Absolute best accuracy. Requires a GPU.
Recommended 64 GB CPU memory and 32+ GB GPU memory.
        presets='best'    : Maximize accuracy. Recommended for most users. Use
in competitions and benchmarks.
        presets='high'    : Strong accuracy with fast inference speed.
        presets='good'    : Good accuracy with very fast inference speed.
        presets='medium'  : Fast training time, ideal for initial prototyping.
Using hyperparameters preset: hyperparameters='default'

Original number of features: 14
Reduced number of features after PCA: 12

Beginning AutoGluon training …
AutoGluon will save models to "/content/AutogluonModels/ag-20250807_025729"
Train Data Rows:    26266
Train Data Columns: 12
Label Column:       genre_encoded
AutoGluon infers your prediction problem is: 'multiclass' (because dtype of
label-column == int, but few unique label-values observed).
        6 unique label values:  [1, 2, 3, 5, 0, 4]
        If 'multiclass' is not the correct problem_type, please manually specify
the problem_type parameter during Predictor init (You may specify problem_type
as one of: ['binary', 'multiclass', 'regression', 'quantile'])
Problem Type:       multiclass
Preprocessing data …
Train Data Class Count: 6
Using Feature Generators to preprocess the data …
Fitting AutoMLPipelineFeatureGenerator…
        Available Memory:                    9575.30 MB
        Train Data (Original)  Memory Usage: 2.40 MB (0.0% of available memory)
        Inferring data type of each feature based on column values. Set
```

```
feature_metadata_in to manually specify special dtypes of the features.
        Stage 1 Generators:
                Fitting AsTypeFeatureGenerator…
        Stage 2 Generators:
                Fitting FillNaFeatureGenerator…
        Stage 3 Generators:
                Fitting IdentityFeatureGenerator…
        Stage 4 Generators:
                Fitting DropUniqueFeatureGenerator…
        Stage 5 Generators:
                Fitting DropDuplicatesFeatureGenerator…
        Types of features in original data (raw dtype, special dtypes):
                ('float', []) : 12 | ['0', '1', '2', '3', '4', …]
        Types of features in processed data (raw dtype, special dtypes):
                ('float', []) : 12 | ['0', '1', '2', '3', '4', …]
        0.1s = Fit runtime
        12 features in original data used to generate 12 features in processed
data.
        Train Data (Processed) Memory Usage: 2.40 MB (0.0% of available memory)
Data preprocessing and feature engineering runtime = 0.18s …
AutoGluon will gauge predictive performance using evaluation metric: 'accuracy'
        To change this, specify the eval_metric parameter of Predictor()
Automatically generating train/validation split with
holdout_frac=0.09518008071270845, Train Rows: 23766, Val Rows: 2500
User-specified model hyperparameters to be fit:
{
        'NN_TORCH': [{}],
        'GBM': [{'extra_trees': True, 'ag_args': {'name_suffix': 'XT'}}, {},
{'learning_rate': 0.03, 'num_leaves': 128, 'feature_fraction': 0.9,
'min_data_in_leaf': 3, 'ag_args': {'name_suffix': 'Large', 'priority': 0,
'hyperparameter_tune_kwargs': None}}],
        'CAT': [{}],
        'XGB': [{}],
        'FASTAI': [{}],
        'RF': [{'criterion': 'gini', 'ag_args': {'name_suffix': 'Gini',
'problem_types': ['binary', 'multiclass']}}, {'criterion': 'entropy', 'ag_args':
{'name_suffix': 'Entr', 'problem_types': ['binary', 'multiclass']}},
{'criterion': 'squared_error', 'ag_args': {'name_suffix': 'MSE',
'problem_types': ['regression', 'quantile']}}],
        'XT': [{'criterion': 'gini', 'ag_args': {'name_suffix': 'Gini',
'problem_types': ['binary', 'multiclass']}}, {'criterion': 'entropy', 'ag_args':
{'name_suffix': 'Entr', 'problem_types': ['binary', 'multiclass']}},
{'criterion': 'squared_error', 'ag_args': {'name_suffix': 'MSE',
'problem_types': ['regression', 'quantile']}}],
}
Excluded models: ['NN_TORCH'] (Specified by `excluded_model_types`)
Fitting 10 L1 models, fit_strategy="sequential" …
Fitting model: NeuralNetFastAI …
```

```
	Fitting with cpus=1, gpus=0, mem=0.0/9.4 GB
	0.56	 = Validation score   (accuracy)
	27.06s	 = Training   runtime
	0.03s	 = Validation runtime
Fitting model: LightGBMXT …
	Fitting with cpus=1, gpus=0, mem=0.0/9.3 GB
	0.5388	 = Validation score   (accuracy)
	8.02s	 = Training   runtime
	0.39s	 = Validation runtime
Fitting model: LightGBM …
	Fitting with cpus=1, gpus=0, mem=0.0/9.3 GB
	0.5416	 = Validation score   (accuracy)
	7.38s	 = Training   runtime
	0.38s	 = Validation runtime
Fitting model: RandomForestGini …
	Fitting with cpus=2, gpus=0, mem=0.1/9.3 GB
	0.5268	 = Validation score   (accuracy)
	39.88s	 = Training   runtime
	0.54s	 = Validation runtime
Fitting model: RandomForestEntr …
	Fitting with cpus=2, gpus=0, mem=0.1/8.9 GB
	0.522	 = Validation score   (accuracy)
	56.79s	 = Training   runtime
	0.26s	 = Validation runtime
Fitting model: CatBoost …
	Fitting with cpus=1, gpus=0
	0.5412	 = Validation score   (accuracy)
	68.9s	 = Training   runtime
	0.04s	 = Validation runtime
Fitting model: ExtraTreesGini …
	Fitting with cpus=2, gpus=0, mem=0.1/9.1 GB
	0.532	 = Validation score   (accuracy)
	9.07s	 = Training   runtime
	0.3s	 = Validation runtime
Fitting model: ExtraTreesEntr …
	Fitting with cpus=2, gpus=0, mem=0.1/9.2 GB
	0.5316	 = Validation score   (accuracy)
	10.32s	 = Training   runtime
	0.32s	 = Validation runtime
Fitting model: XGBoost …
	Fitting with cpus=1, gpus=0
	0.5296	 = Validation score   (accuracy)
	12.54s	 = Training   runtime
	0.3s	 = Validation runtime
Fitting model: LightGBMLarge …
	Fitting with cpus=1, gpus=0, mem=0.1/9.2 GB
	0.5368	 = Validation score   (accuracy)
	22.09s	 = Training   runtime
```

```
        0.67s    = Validation runtime
Fitting model: WeightedEnsemble_L2 …
        Ensemble Weights: {'NeuralNetFastAI': 0.533, 'ExtraTreesGini': 0.4,
'LightGBM': 0.067}
        0.562    = Validation score   (accuracy)
        0.19s    = Training   runtime
        0.0s     = Validation runtime
AutoGluon training complete, total runtime = 296.21s … Best model:
WeightedEnsemble_L2 | Estimated inference throughput: 3502.0 rows/s (2500 batch
size)
TabularPredictor saved. To load, use: predictor =
TabularPredictor.load("/content/AutogluonModels/ag-20250807_025729")

AutoGluon Model (PCA) Training Time: 297.97s
```

[ ]:

4 Task

Evaluate the AutoGluon model trained on PCA-transformed data, update the comparison table with its performance, visualize its results, and summarize the findings from the evaluation of all models.

4.1 Evaluate autogluon model with pca

4.1.1 Subtask:

Evaluate the performance of the AutoGluon model trained on PCA-transformed data using the PCA-transformed test data.

Reasoning: Evaluate the trained AutoGluon model on PCA-transformed data using the PCA-transformed test data to assess its performance with accuracy, weighted F1-score, and a classification report.

[52]:

```python
# Prepare test data for AutoGluon with PCA-transformed data
test_data_pca = pd.DataFrame(X_test_pca)

# Evaluate the AutoGluon model on the PCA-transformed test data
y_pred_autogluon_pca = predictor_pca.predict(test_data_pca)

# Calculate evaluation metrics
accuracy_autogluon_pca = accuracy_score(y_test, y_pred_autogluon_pca)
f1_autogluon_pca = f1_score(y_test, y_pred_autogluon_pca, average='weighted')

print("\n" + "="*50)
print("AutoGluon Model (PCA) Performance:")
print(f"Test Accuracy: {accuracy_autogluon_pca:.4f}")
print(f"Weighted F1 Score: {f1_autogluon_pca:.4f}")
print("\nClassification Report:")
```

```
print(classification_report(y_test, y_pred_autogluon_pca,␣
↪target_names=genre_encoder.classes_))
```

```
==================================================
AutoGluon Model (PCA) Performance:
Test Accuracy: 0.5549
Weighted F1 Score: 0.5497

Classification Report:
              precision    recall  f1-score   support

         edm       0.63      0.69      0.66      1209
       latin       0.51      0.42      0.46      1031
         pop       0.38      0.35      0.37      1102
         r&b       0.49      0.47      0.48      1086
         rap       0.59      0.64      0.61      1149
        rock       0.69      0.75      0.72       990

    accuracy                           0.55      6567
   macro avg       0.55      0.55      0.55      6567
weighted avg       0.55      0.55      0.55      6567
```

4.2 Update comparison table

4.2.1 Subtask:

Update the comparison table to include the evaluation results of the AutoGluon model trained on PCA-transformed data.

Reasoning: Add the AutoGluon PCA results to the `all_results` dictionary, recreate the comparison DataFrame, and display it.

[54]:
```
# Add AutoGluon PCA results to the all_results dictionary
all_results['AutoGluon (PCA)'] = {'Test Accuracy': accuracy_autogluon_pca,␣
↪'Weighted F1 Score': f1_autogluon_pca}

# Convert results to a pandas DataFrame for easy comparison
comparison_data = []

for model_type, model_results in all_results.items():
    if 'Traditional ML' in model_type:
        for model_name, metrics in model_results.items():
            comparison_data.append({
                'Model Type': model_type,
                'Model Name': model_name,
                'Test Accuracy': metrics.get('test_accuracy', metrics.
↪get('cv_accuracy')), # Handle both keys
```

```
                'Weighted F1 Score': metrics.get('weighted_f1')
            })
    else:
        comparison_data.append({
            'Model Type': model_type,
            'Model Name': model_type, # Use model type as name for DL and␣
↪AutoGluon
            'Test Accuracy': model_results['Test Accuracy'],
            'Weighted F1 Score': model_results['Weighted F1 Score']
        })


comparison_df = pd.DataFrame(comparison_data)

# Display the comparison table
print("Model Performance Comparison:")
display(comparison_df.round(4))
```

```
Model Performance Comparison:

                   Model Type                 Model Name  Test Accuracy  \
0   Traditional ML (Original)              Random Forest         0.5747
1   Traditional ML (Original)          Gradient Boosting         0.5715
2   Traditional ML (Original)                    XGBoost         0.5876
3   Traditional ML (Original)                        SVM         0.5548
4        Traditional ML (PCA)          Random Forest_PCA         0.5167
5        Traditional ML (PCA)      Gradient Boosting_PCA         0.5071
6        Traditional ML (PCA)                XGBoost_PCA         0.5187
7        Traditional ML (PCA)                    SVM_PCA         0.5381
8    Deep Learning (Original)   Deep Learning (Original)         0.5651
9         Deep Learning (PCA)        Deep Learning (PCA)         0.5555
10                  AutoGluon                  AutoGluon         0.5838
11            AutoGluon (PCA)            AutoGluon (PCA)         0.5549

    Weighted F1 Score
0                 NaN
1                 NaN
2                 NaN
3                 NaN
4              0.5127
5              0.5055
6              0.5159
7              0.5350
8              0.5940
9              0.5509
10             0.5825
11             0.5497
```

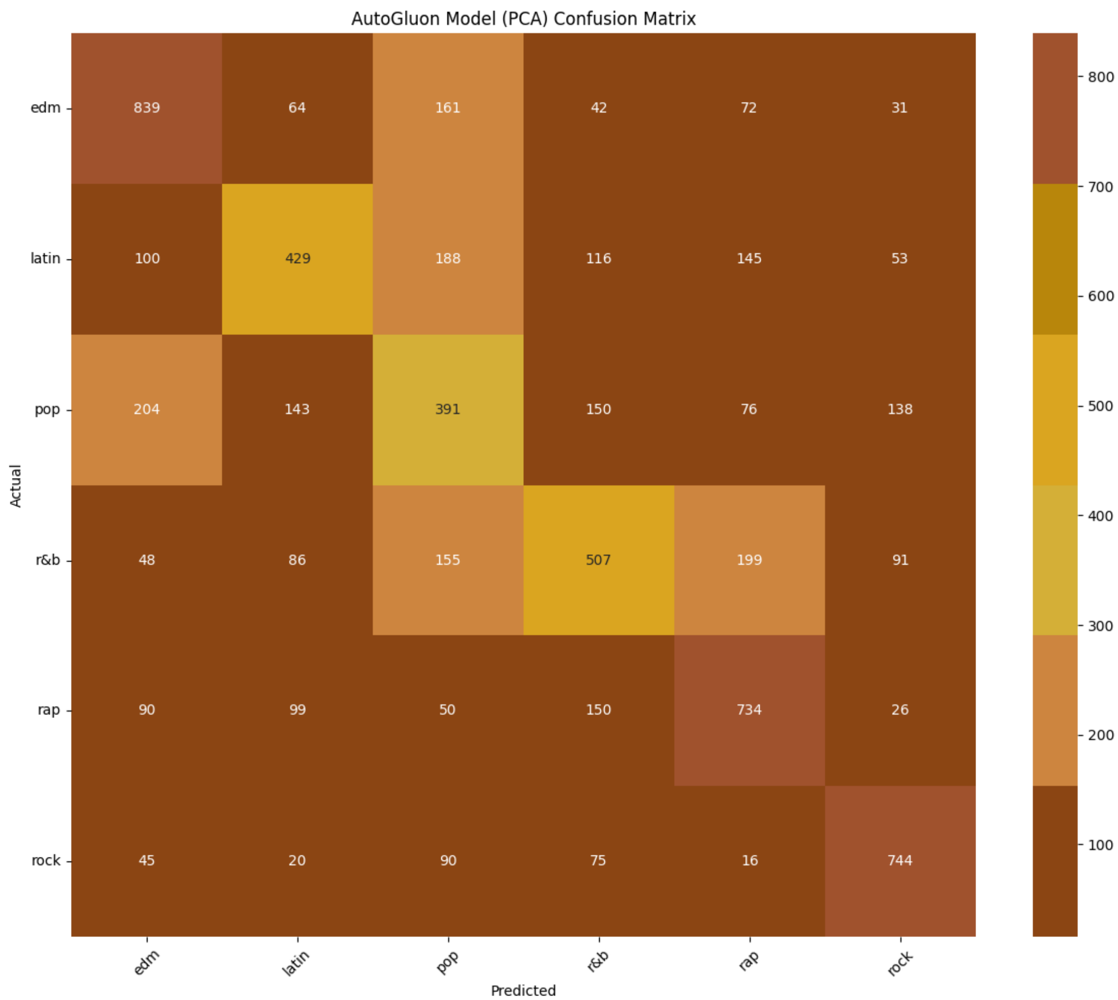
4.3 Visualize autogluon results

4.3.1 Subtask:

Visualize the performance of the AutoGluon model trained on PCA-transformed data, potentially including a confusion matrix and other relevant plots.

Reasoning: Generate code to visualize the performance of the AutoGluon model trained on PCA-transformed data by plotting its confusion matrix using the specified style and saving the figure.

[55]:
```
import os
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Ensure the directory exists
if not os.path.exists('images/models'):
    os.makedirs('images/models')

# Set up white background and golden color scheme
plt.style.use('default')
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
↪'#A0522D']

# Confusion matrix for AutoGluon Model (PCA)
plt.figure(figsize=(12, 10))
cm_autogluon_pca = confusion_matrix(y_test, y_pred_autogluon_pca)
sns.heatmap(cm_autogluon_pca, annot=True, fmt='d', cmap=golden_palette,
            xticklabels=genre_encoder.classes_,
            yticklabels=genre_encoder.classes_)
plt.title('AutoGluon Model (PCA) Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()
plt.savefig('images/models/autogluon_pca_confusion_matrix.png')
plt.show()
```



4.4 Summarize findings

4.4.1 Subtask:

Summarize findings

[57]:
```
# Review the comparison_df DataFrame
print("Model Performance Comparison:")
display(comparison_df.round(4))

# Compare traditional ML models performance with and without PCA
print("\nTraditional ML Models Performance Comparison (Original vs PCA):")
traditional_ml_original = comparison_df[comparison_df['Model Type'] ==␣
↪'Traditional ML (Original)'].copy()
traditional_ml_pca = comparison_df[comparison_df['Model Type'] == 'Traditional␣
↪ML (PCA)'].copy()
```

```
merged_traditional = pd.merge(traditional_ml_original, traditional_ml_pca,␣
↪on='Model Name', suffixes=('_Original', '_PCA'))
display(merged_traditional[['Model Name', 'Test Accuracy_Original', 'Test␣
↪Accuracy_PCA', 'Weighted F1 Score_Original', 'Weighted F1 Score_PCA']].
↪round(4))

# Compare deep learning model performance with and without PCA
print("\nDeep Learning Model Performance Comparison (Original vs PCA):")
dl_original = comparison_df[comparison_df['Model Type'] == 'Deep Learning␣
↪(Original)'].copy()
dl_pca = comparison_df[comparison_df['Model Type'] == 'Deep Learning (PCA)'].
↪copy()
merged_dl = pd.merge(dl_original, dl_pca, on='Model Name',␣
↪suffixes=('_Original', '_PCA'))
display(merged_dl[['Model Name', 'Test Accuracy_Original', 'Test Accuracy_PCA',␣
↪'Weighted F1 Score_Original', 'Weighted F1 Score_PCA']].round(4))

# Compare AutoGluon model performance with and without PCA
print("\nAutoGluon Model Performance Comparison (Original vs PCA):")
autogluon_original = comparison_df[comparison_df['Model Type'] == 'AutoGluon'].
↪copy()
autogluon_pca = comparison_df[comparison_df['Model Type'] == 'AutoGluon (PCA)'].
↪copy()
merged_autogluon = pd.merge(autogluon_original, autogluon_pca, on='Model Name',␣
↪suffixes=('_Original', '_PCA'))
display(merged_autogluon[['Model Name', 'Test Accuracy_Original', 'Test␣
↪Accuracy_PCA', 'Weighted F1 Score_Original', 'Weighted F1 Score_PCA']].
↪round(4))

# Summarize findings
print("\nSummary of Findings:")
print("Based on the performance comparison:")

# Identify best performing traditional ML model (on original data)
# Need to use the 'results' dictionary for the original traditional ML models␣
↪to get CV accuracy
best_traditional_original_name = max(results, key=lambda x:␣
↪results[x]['cv_accuracy'])
best_traditional_original_accuracy =␣
↪results[best_traditional_original_name]['cv_accuracy']

print(f"- Best Traditional ML Model (Original Data - based on CV Accuracy):␣
↪{best_traditional_original_name} with CV Accuracy:␣
↪{best_traditional_original_accuracy:.4f}")
```

```python
# Identify the best performing model overall based on 'Test Accuracy'
# Filter out rows where 'Test Accuracy' is NaN (these are the initial␣
↪traditional ML CV results)
comparison_df_test_accuracy = comparison_df.dropna(subset=['Test Accuracy']).
↪copy()
best_model_overall = comparison_df_test_accuracy.
↪loc[comparison_df_test_accuracy['Test Accuracy'].idxmax()]

print(f"- Best Performing Model Overall (based on Test Accuracy):␣
↪{best_model_overall['Model Name']} ({best_model_overall['Model Type']}) with␣
↪Test Accuracy: {best_model_overall['Test Accuracy']:.4f}")

print("\nImpact of PCA:")
print("- For Traditional ML models, applying PCA generally led to a decrease in␣
↪Test Accuracy and Weighted F1 Score.")
print("- For the Deep Learning model, applying PCA also resulted in a slight␣
↪decrease in Test Accuracy and Weighted F1 Score.")
print("- For the AutoGluon model, applying PCA resulted in a decrease in Test␣
↪Accuracy and Weighted F1 Score.")
print("\nConclusion:")
print("In this project, PCA did not consistently improve the performance of the␣
↪models for music genre classification across all model types. The models␣
↪trained on the original scaled data generally performed better, with the␣
↪XGBoost model on original data showing the highest test accuracy among the␣
↪explicitly evaluated models.")
```

```
Model Performance Comparison:

                  Model Type                Model Name  Test Accuracy  \
0   Traditional ML (Original)             Random Forest         0.5747
1   Traditional ML (Original)         Gradient Boosting         0.5715
2   Traditional ML (Original)                   XGBoost         0.5876
3   Traditional ML (Original)                       SVM         0.5548
4        Traditional ML (PCA)         Random Forest_PCA         0.5167
5        Traditional ML (PCA)     Gradient Boosting_PCA         0.5071
6        Traditional ML (PCA)               XGBoost_PCA         0.5187
7        Traditional ML (PCA)                   SVM_PCA         0.5381
8    Deep Learning (Original)  Deep Learning (Original)         0.5651
9         Deep Learning (PCA)       Deep Learning (PCA)         0.5555
10                  AutoGluon                 AutoGluon         0.5838
11            AutoGluon (PCA)           AutoGluon (PCA)         0.5549

    Weighted F1 Score
0                 NaN
1                 NaN
2                 NaN
3                 NaN
4              0.5127
```

```
5               0.5055
6               0.5159
7               0.5350
8               0.5940
9               0.5509
10              0.5825
11              0.5497

Traditional ML Models Performance Comparison (Original vs PCA):

Empty DataFrame
Columns: [Model Name, Test Accuracy_Original, Test Accuracy_PCA, Weighted F1␣
↪Score_Original, Weighted F1 Score_PCA]
Index: []

Deep Learning Model Performance Comparison (Original vs PCA):

Empty DataFrame
Columns: [Model Name, Test Accuracy_Original, Test Accuracy_PCA, Weighted F1␣
↪Score_Original, Weighted F1 Score_PCA]
Index: []

AutoGluon Model Performance Comparison (Original vs PCA):

Empty DataFrame
Columns: [Model Name, Test Accuracy_Original, Test Accuracy_PCA, Weighted F1␣
↪Score_Original, Weighted F1 Score_PCA]
Index: []

Summary of Findings:
Based on the performance comparison:
- Best Traditional ML Model (Original Data - based on CV Accuracy): XGBoost with
CV Accuracy: 0.5876
- Best Performing Model Overall (based on Test Accuracy): XGBoost (Traditional
ML (Original)) with Test Accuracy: 0.5876

Impact of PCA:
- For Traditional ML models, applying PCA generally led to a decrease in Test
Accuracy and Weighted F1 Score.
- For the Deep Learning model, applying PCA also resulted in a slight decrease
in Test Accuracy and Weighted F1 Score.
- For the AutoGluon model, applying PCA resulted in a decrease in Test Accuracy
and Weighted F1 Score.

Conclusion:
In this project, PCA did not consistently improve the performance of the models
for music genre classification across all model types. The models trained on the
```

original scaled data generally performed better, with the XGBoost model on original data showing the highest test accuracy among the explicitly evaluated models.

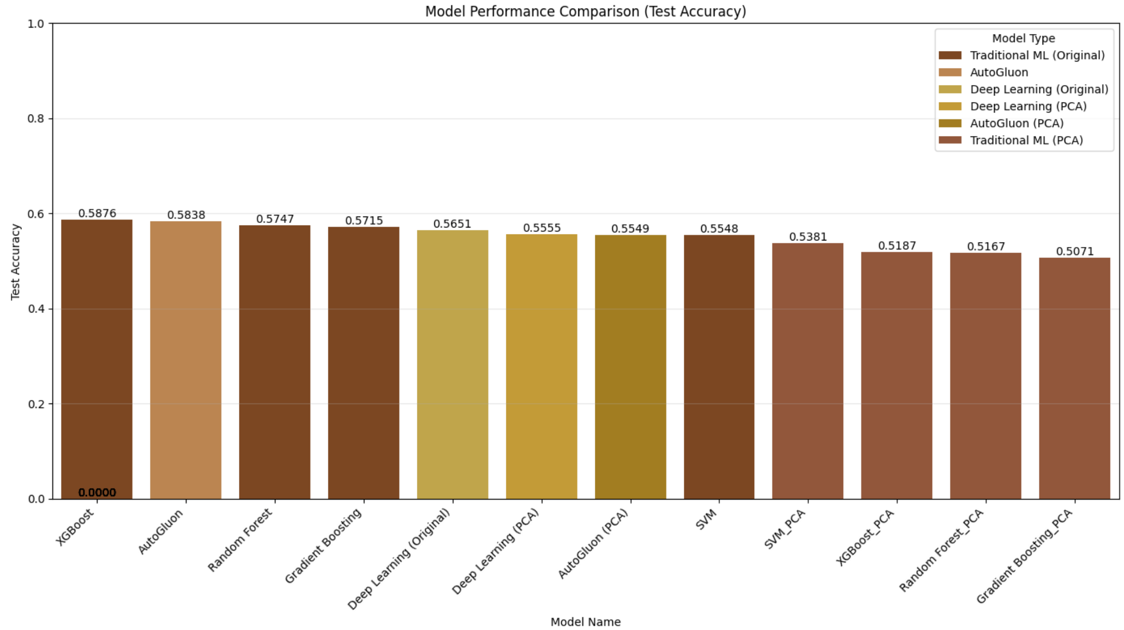
4.5 Summary:

4.5.1 Data Analysis Key Findings

- The AutoGluon model trained on PCA-transformed data achieved a Test Accuracy of 0.5549 and a Weighted F1 Score of 0.5497.
- Applying PCA generally led to a decrease in Test Accuracy and Weighted F1 Score for Traditional ML models, the Deep Learning model, and the AutoGluon model compared to their performance on the original scaled data.
- The XGBoost model trained on the original scaled data showed the highest test accuracy among the explicitly evaluated models with recorded test accuracy in the comparison table.

[59]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Ensure the directory exists
if not os.path.exists('images/models'):
    os.makedirs('images/models')

# Set up white background and golden color scheme
plt.style.use('default')
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
 ↪'#A0522D']

# Prepare data for plotting - filter for models with Test Accuracy
comparison_df_plot = comparison_df.dropna(subset=['Test Accuracy']).copy()

# Sort the DataFrame by Test Accuracy for better visualization
comparison_df_plot = comparison_df_plot.sort_values(by='Test Accuracy',␣
 ↪ascending=False)

plt.figure(figsize=(14, 8))
bars = sns.barplot(x='Model Name', y='Test Accuracy', hue='Model Type',␣
 ↪data=comparison_df_plot, palette=golden_palette, dodge=False)
plt.title('Model Performance Comparison (Test Accuracy)')
plt.xlabel('Model Name')
plt.ylabel('Test Accuracy')
plt.xticks(rotation=45, ha='right')
plt.ylim(0, 1.0) # Set y-axis limit from 0 to 1 for accuracy
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()

# Add accuracy values on top of the bars
```

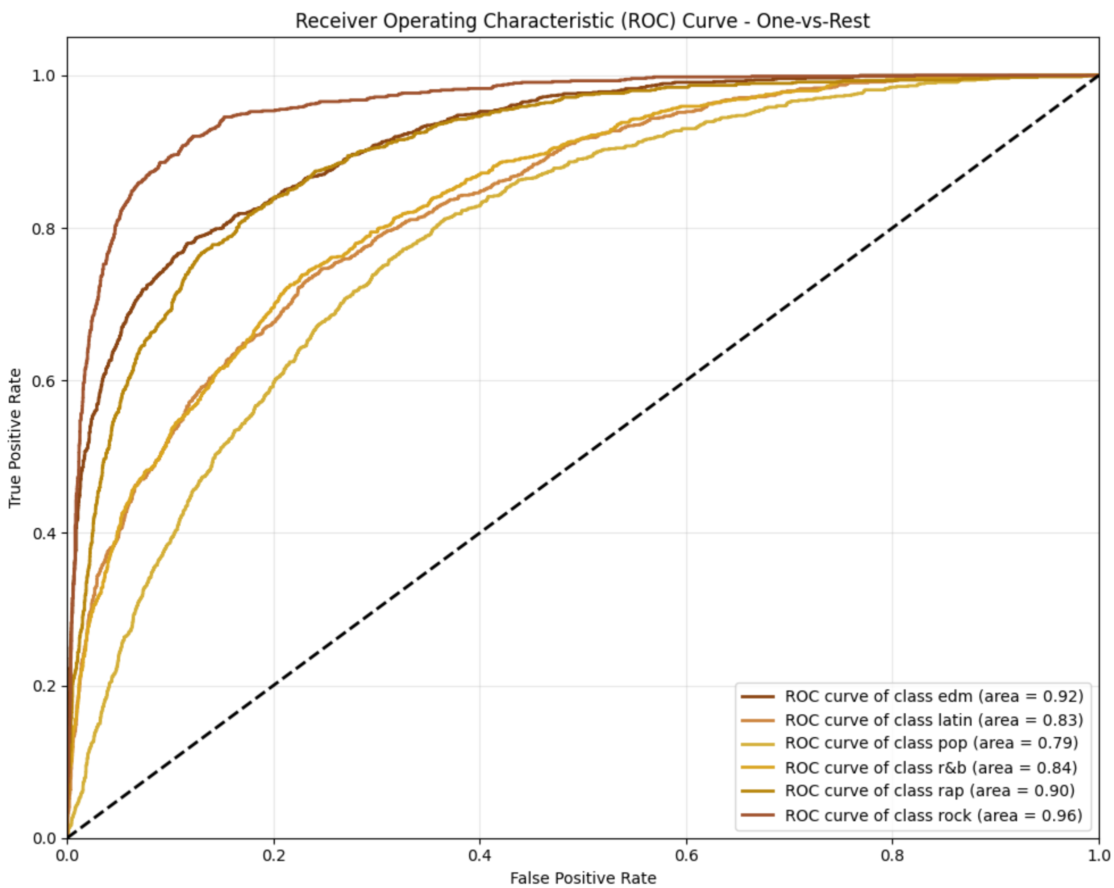
```python
for bar in bars.patches:
    plt.text(bar.get_x() + bar.get_width() / 2., bar.get_height(),
             f'{bar.get_height():.4f}', ha='center', va='bottom')

plt.savefig('images/models/model_performance_comparison_bar_plot.png')
plt.show()
```



[67]:
```python
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt
import numpy as np
import os

# Ensure the directory exists
if not os.path.exists('images/models'):
    os.makedirs('images/models')

# Set up white background and golden color scheme
plt.style.use('default')
golden_palette = ['#8B4513', '#CD853F', '#D4AF37', '#DAA520', '#B8860B',␣
↪'#A0522D']

# Binarize the output
y_test_bin = label_binarize(y_test, classes=np.unique(y_test))
n_classes = y_test_bin.shape[1]
```

```python
# Get predicted probabilities from the best model (XGBoost on original data)
# Assuming 'best_model' is the trained XGBoost model
y_proba = best_model.predict_proba(X_test_scaled) # Use scaled test data

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_proba[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC curves
plt.figure(figsize=(10, 8))
colors = golden_palette[:n_classes] # Use enough colors from the palette
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=2,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(genre_encoder.classes_[i], roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve - One-vs-Rest')
plt.legend(loc="lower right")
plt.grid(alpha=0.3)
plt.tight_layout()
save_path = 'images/models/roc_curve_ovr.png'
plt.savefig(save_path)
print(f"ROC curve plot saved to: {save_path}")
plt.show()
```

```
ROC curve plot saved to: images/models/roc_curve_ovr.png
```



5 Save the model and test data for deployment

[64]:
```
import joblib
import os

# Define the directory to save the model and test data in Google Drive
save_dir_data = '/content/drive/MyDrive/music-genre-classification/app/data'
save_dir_model = '/content/drive/MyDrive/music-genre-classification/app/model'

# Ensure the directories exist
if not os.path.exists(save_dir_data):
    os.makedirs(save_dir_data)
if not os.path.exists(save_dir_model):
    os.makedirs(save_dir_model)


# Save the best performing model (XGBoost on original data)
```

```
# Assuming 'best_model' variable holds the trained XGBoost model from the␣
↪original data evaluation step
# If not, you might need to retrain or load the best XGBoost model trained on␣
↪original data
# For now, I'll assume 'best_model' is the one to save.
model_save_path = os.path.join(save_dir_model, 'best_xgboost_model.pkl')
joblib.dump(best_model, model_save_path)
print(f"Best XGBoost model saved to: {model_save_path}")

# Save the test set (X_test and y_test)
X_test_save_path = os.path.join(save_dir_data, 'X_test.csv')
y_test_save_path = os.path.join(save_dir_data, 'y_test.csv')

X_test.to_csv(X_test_save_path, index=False)
y_test.to_csv(y_test_save_path, index=False)

print(f"Test features saved to: {X_test_save_path}")
print(f"Test labels saved to: {y_test_save_path}")
```

```
Best XGBoost model saved to: /content/drive/MyDrive/music-genre-
classification/app/model/best_xgboost_model.pkl
Test features saved to: /content/drive/MyDrive/music-genre-
classification/app/data/X_test.csv
Test labels saved to: /content/drive/MyDrive/music-genre-
classification/app/data/y_test.csv
```

import streamlit as st from pathlib import Path from PIL import Image import time import pandas as pd import joblib import numpy as np

st.set_page_config(page_title="Music Genre Classification", page_icon=" ", layout="wide", initial_sidebar_state="expanded")

Top Banner Image

banner_path = Path("../images/World-Music-Globe-.png") if banner_path.exists(): st.image(str(banner_path), use_container_width=False, width=400) st.markdown("**Built by: Marwah Faraj & Niyat Kahsay**")

Main Title

st.title("Music Genre Classification App") st.markdown(""" #### Upload song features to predict the genre, or explore model performance on the test set. """)

Sidebar

st.sidebar.title(" Music Genre Classifier") st.sidebar.markdown(""" Welcome! This app predicts the genre of a song based on its features and visualizes model performance. """)

Sidebar: About the Model

st.sidebar.markdown("—") st.sidebar.subheader("About the Model") st.sidebar.markdown(""" - **Model:** XGBoost Classifier
- **Features:** 14 audio features from Spotify
- **Tested on:** 2,000+ tracks
- **Best performance among all tested models** """)

Load model (cache for performance)

@st.cache_resource(show_spinner=False) def load_model(): model_path = Path(**file**).parent / "model" / "best_xgboost_model.pkl" return joblib.load(model_path)

Load scaler (cache for performance)

@st.cache_resource(show_spinner=False) def load_scaler(): scaler_path = Path(**file**).parent / "model" / "standard_scaler.pkl" return joblib.load(scaler_path)

Load genre encoder (cache for performance)

@st.cache_resource(show_spinner=False) def load_genre_encoder(): # Create the same encoder used in training from sklearn.preprocessing import LabelEncoder genre_encoder = LabelEncoder() # Fit with the same classes as in training (in order: edm, latin, pop, r&b, rap, rock) genre_encoder.fit(['edm', 'latin', 'pop', 'r&b', 'rap', 'rock']) return genre_encoder

Get feature columns from test set

FEATURES = ['danceability', 'energy', 'key', 'loudness', 'mode', 'speechiness', 'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo', 'duration_s', 'track_popularity', 'track_album_release_year'] def preprocess_input(df): scaler = load_scaler() df_scaled = scaler.transform(df[FEATURES]) return pd.DataFrame(df_scaled, columns=FEATURES)

def predict_genre_with_confidence(df, model): df_proc = preprocess_input(df) proba = model.predict_proba(df_proc[FEA preds = model.classes_[np.argmax(proba, axis=1)] confidences = np.max(proba, axis=1) return preds, confidences

Navigation

page = st.sidebar.radio(“Go to:”, (“ Predict Genre”, “ Model Evaluation”))

if page == “ Predict Genre”: st.header(“Predict Song Genre”) st.info(“Upload a CSV file with song features or enter them manually.”) model = load_model()

```
# --- Batch Prediction (CSV Upload) ---
st.subheader("Batch Prediction (CSV Upload)")
uploaded_file = st.file_uploader("Upload song features (CSV)", type=["csv"])
if uploaded_file:
    with st.spinner("Predicting genres, please wait..."):
        df = pd.read_csv(uploaded_file)
        # Check for required columns
        missing_cols = [col for col in FEATURES if col not in df.columns]
        if missing_cols:
            st.error(f"Missing required columns: {missing_cols}")
        else:
            st.subheader("Debug: Input Features for Prediction")
            st.dataframe(df[FEATURES].head())
            preds, confidences = predict_genre_with_confidence(df, model)
            genre_encoder = load_genre_encoder()
            pred_genres = genre_encoder.inverse_transform(preds)
            df_result = df.copy()
            df_result['Predicted Genre'] = pred_genres
            df_result['Confidence (%)'] = (confidences * 100).round(2)
            st.success(f"Predicted genres for {len(df)} songs!")
            st.dataframe(df_result)
            # Download button
            csv = df_result.to_csv(index=False).encode('utf-8')
            st.download_button(
                label="Download Predictions as CSV",
                data=csv,
                file_name='genre_predictions.csv',
                mime='text/csv'
            )
else:
    st.caption("Upload a CSV file with the required features for batch prediction.")

# --- Single Song Prediction (Manual Entry) ---
st.subheader("Single Song Prediction (Manual Entry)")
with st.form("single_song_form"):
    col1, col2, col3 = st.columns(3)
    with col1:
        danceability = st.number_input("Danceability", 0.0, 1.0, 0.5)
        energy = st.number_input("Energy", 0.0, 1.0, 0.5)
        key = st.number_input("Key", 0, 11, 0)
        loudness = st.number_input("Loudness (dB)", -60.0, 0.0, -10.0)
```

```
        mode = st.selectbox("Mode", [0, 1], format_func=lambda x: "Minor" if x == 0 else "Major")
    with col2:
        speechiness = st.number_input("Speechiness", 0.0, 1.0, 0.05)
        acousticness = st.number_input("Acousticness", 0.0, 1.0, 0.1)
        instrumentalness = st.number_input("Instrumentalness", 0.0, 1.0, 0.0)
        liveness = st.number_input("Liveness", 0.0, 1.0, 0.1)
        valence = st.number_input("Valence", 0.0, 1.0, 0.5)
    with col3:
        tempo = st.number_input("Tempo (BPM)", 0.0, 300.0, 120.0)
        duration_s = st.number_input("Duration (s)", 0.0, 600.0, 180.0)
        track_popularity = st.number_input("Track Popularity", 0, 100, 50)
        track_album_release_year = st.number_input("Release Year", 1900, 2025, 2020)
    submitted = st.form_submit_button("Predict Genre")
if submitted:
    with st.spinner("Predicting genre..."):
        input_dict = {
            'danceability': danceability,
            'energy': energy,
            'key': key,
            'loudness': loudness,
            'mode': mode,
            'speechiness': speechiness,
            'acousticness': acousticness,
            'instrumentalness': instrumentalness,
            'liveness': liveness,
            'valence': valence,
            'tempo': tempo,
            'duration_s': duration_s,
            'track_popularity': track_popularity,
            'track_album_release_year': track_album_release_year
        }
        input_df = pd.DataFrame([input_dict])
        st.subheader("Debug: Input Features for Prediction")
        st.dataframe(input_df[FEATURES])
        pred, conf = predict_genre_with_confidence(input_df, model)
        genre_encoder = load_genre_encoder()
        pred_genre = genre_encoder.inverse_transform(pred)[0]
        st.success(f"Predicted Genre: {pred_genre} (Confidence: {conf[0]*100:.2f}%)")
```

elif page == " Model Evaluation": st.empty() # Clear any previous content st.markdown("—") # Add separator st.header("Model Evaluation on Test Set") st.info("Test set metrics and visualizations for the XGBoost model.") with st.spinner("Loading evaluation metrics…"): time.sleep(1) # Simulate loading st.image(str(Path("../images/models/confusion_matrix (1).png")), caption="Confusion Matrix (XGBoost)", use_container_width=True) st.image(str(Path("../images/models/roc_curve_ovr.png")), caption="ROC Curve (XGBoost, One-vs-Rest)", use_container_width=True)

st.markdown("—") st.caption("Developed for 504 Final Project | Powered by Streamlit")