Music Recommendation

Table of Contents

Overview

This project aims to develop a machine learning model that classifies songs into different emotional categories based on audio features. The dataset consists of songs with features such as valence, energy, danceability, loudness, and tempo, which influence the perceived emotional tone. By utilizing clustering techniques and classification models (Random Forest and MLP), the goal is to predict emotions like "Energetic," "Joyful," "Calm," "Sad," and "Angry" for each song. This classification can later be used for personalized song recommendations based on mood or emotional preferences.

It is important to note that the model does not analyze lyrics or vocals; it only considers the audio features mentioned above.

Importing Data

```
import pandas as pd

#loading the data set
df = pd.read_csv("SpotifyDataset/spotify_songs.csv")
df.head()
```

	track_id	track_name	track_artist	track_popularity	track
0	6f807x0ima9a1j3VPbc7VN	I Don't Care (with Justin Bieber) - Loud Luxur...	Ed Sheeran	66	2oCs0DGTsRO98
1	0r7CVbZTWZgbTCYdfa2P31	Memories - Dillon Francis Remix	Maroon 5	67	63rPSO264uRjW1
2	1z1Hg7Vb0AhHDiEmnDE79l	All the Time - Don Diablo Remix	Zara Larsson	70	1HoSmj2eLcsrR
3	75FpbthrwQmzHlBJLuGdC7	Call You Mine - Keanu Silva Remix	The Chainsmokers	60	1nqYsOef1yKKu(
4	1e8PAfcKUYoKkxPhrHqw4x	Someone You Loved - Future Humans Remix	Lewis Capaldi	69	7m7vv9wlQ4i0L

5 rows × 23 columns

Explaning variables

Keywords	Values	
danceability	0 - 1.0	
energy	0 - 1.0	
key	Double	
loudness	-60-0	
mode	0-1	
speechiness	Double	Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the rec
acousticness	0 - 1.0	
instrumentalness	Double	
liveness	Double	
valence	0 - 1.0	
tempo	Double	
duration_ms	Double	

Table and dataset taking from https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset/data

Data Cleaning

```
#looking for missing data

print(df.isnull().sum())
```

```
track_id                    0
track_name                  5
track_artist                5
track_popularity            0
track_album_id              0
track_album_name            5
track_album_release_date    0
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
dtype: int64
```

Removing Rows with Missing Values

```
# Drop rows with any missing values
df = df.dropna()

# Verify that there are no more missing values
print(df.isnull().sum())

# Check the updated size of the dataset
print(f"Number of rows after removing missing values: {df.shape[0]}")
```

```
track_id                    0
track_name                  0
track_artist                0
track_popularity            0
track_album_id              0
track_album_name            0
track_album_release_date    0
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
dtype: int64
Number of rows after removing missing values: 32828
```

Checking the data type for each column

```
df.dtypes
```

```
track_id                     object
track_name                   object
track_artist                 object
track_popularity              int64
track_album_id               object
track_album_name             object
track_album_release_date     object
playlist_name                object
playlist_id                  object
playlist_genre               object
playlist_subgenre            object
danceability                float64
energy                      float64
key                           int64
loudness                    float64
mode                          int64
speechiness                 float64
acousticness                float64
instrumentalness            float64
liveness                    float64
valence                     float64
tempo                       float64
```

```
duration_ms                    int64
dtype: object
```

```
#check for dupllicated rows and if any remove them
duplicated_rows = df.duplicated().sum()
print(duplicated_rows)
if duplicated_rows!=0:
    df.drop_duplicates()
```

```
0
```

Preparation

We will classify the data before the training process. For this, we need to select different categories that will be useful for recommending songs accurately based on the user's emotions. For simplicity, we use default emotions that the user can choose from. Below is an example of variables that are likely to contribute to classifying a song based on its emotion. I predict the following associations between variables and emotions:

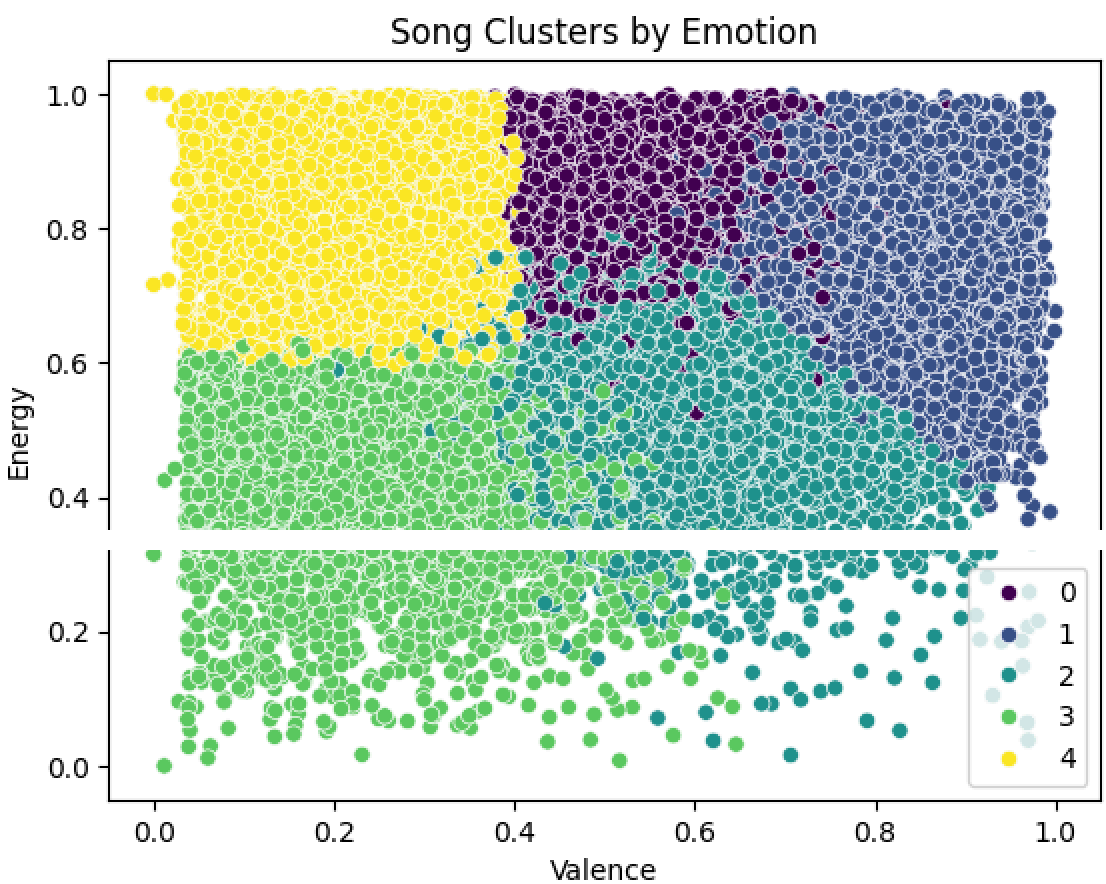
1. Joyful - High valence, energy, and danceability.
2. Sad - Low valence and energy.
3. Calm - Moderate valence, low energy, and low loudness.
4. Energetic - High energy and tempo, moderate valence.
5. Angry - Low valence, high energy

We will use K-Clustering to help us classify the songs quickly.

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Select features
features = ['valence', 'energy', 'danceability', 'loudness', 'tempo']
df_features = df[features]

# Normalize features
scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(df_features)
```

```
# Apply K-Means
#use 5 for clusters since we are observing 5 variables
kmeans = KMeans(n_clusters=5, random_state=42)
clusters = kmeans.fit_predict(normalized_data)

# Add cluster labels to dataset
df['emotion_cluster'] = clusters


# Example Visualize clusters
import seaborn as sns
sns.scatterplot(x=normalized_data[:, 0], y=normalized_data[:, 1], hue=clusters, palette='vir
plt.xlabel('Valence')
plt.ylabel('Energy')
plt.title('Song Clusters by Emotion')
plt.show()
```



Saving Proccessed data.

```
import joblib

# Save preprocessed data and clustering model
```

```
df.to_csv("preprocessed_data.csv", index=False)
joblib.dump(scaler, 'minmax_scaler.pkl')  # Save the scaler
joblib.dump(kmeans, 'kmeans_model.pkl')  # Save the clustering model

#loading the data into variables
scaler = "minmax_scaler.pkl"
kmeans_path = "kmeans_model.pkl"
df_path = "preprocessed_dataframe.csv"
```

Examing the K-clusters

We applied a k-means clustering technique to the graph to help classify the dataset more effectively and visualize an example of valence vs. energy. The graph looks good, and there is some overlapping, which is expected considering that some songs can exhibit a mix of different emotions, resulting in imperfect and indistinct boundaries.

```
# Group data by clusters and calculate feature means
cluster_summary = pd.DataFrame(data=normalized_data, columns=features)
cluster_summary['Cluster'] = kmeans.labels_
cluster_means = cluster_summary.groupby('Cluster').mean()

print(cluster_means)
```

```
          valence    energy  danceability  loudness     tempo
Cluster
0        0.550727  0.830376      0.581754  0.861508  0.541825
1        0.810146  0.768622      0.746998  0.844189  0.499412
2        0.564478  0.567334      0.764183  0.807609  0.469615
3        0.269011  0.435314      0.594407  0.767869  0.485511
4        0.250470  0.806265      0.601295  0.860896  0.525102
```

Examing the quality of our cluster and categorization

```
from sklearn.metrics import silhouette_score

score = silhouette_score(normalized_data, kmeans.labels_)
print(f"Silhouette Score: {score:.2f}")
```

```
Silhouette Score: 0.21
```

Improving our Silhouette Score

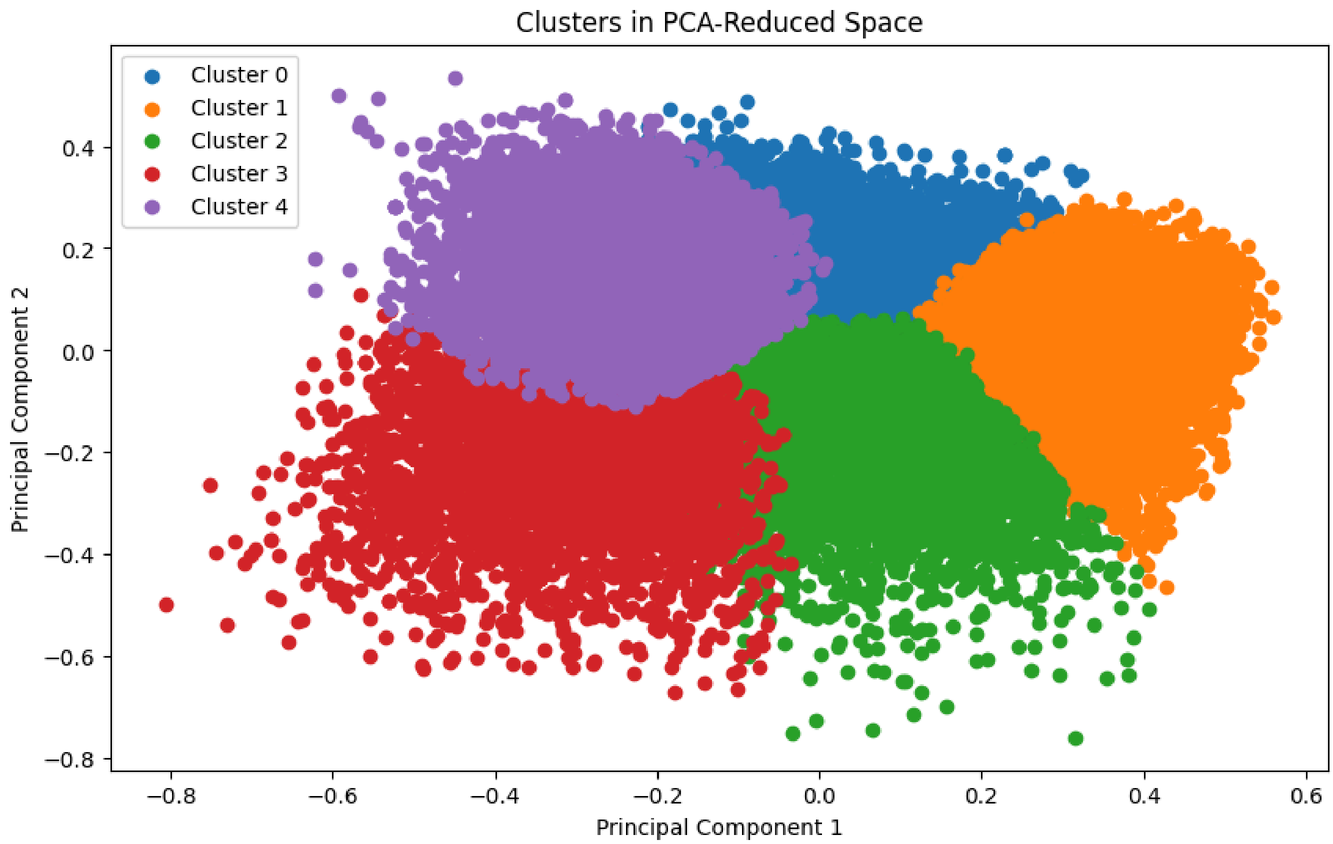
The goal is to improve our silhouette score as much as possible before we finally consider whether the score makes sense conceptually.

```
from sklearn.decomposition import PCA

# Apply PCA to reduce to 2 components
pca = PCA(n_components=2)
pca_result = pca.fit_transform(normalized_data)

# Add the principal components to the dataframe
df['PC1'] = pca_result[:, 0]
df['PC2'] = pca_result[:, 1]

# Display explained variance ratio for PCA components
print("Explained Variance Ratio:", pca.explained_variance_ratio_)
```

```
Explained Variance Ratio: [0.47558111 0.27755501]
```

The explained variance indicates how spread out our data is in the reduced-dimensional space. We have a variance of 0.47 from PC1 and 0.277 from PC2, with a total of approximately 75%. This means most of the data is represented in the reduced 2D space, with PC1 being the most informative dimension.

```
# Add the cluster labels to the dataframe
df['Cluster'] = kmeans.labels_

# Plot the clusters with their labels
plt.figure(figsize=(10, 6))
for cluster in range(5):  #5 clusters
    cluster_data = df[df['Cluster'] == cluster]
    plt.scatter(cluster_data['PC1'], cluster_data['PC2'], label=f'Cluster {cluster}')

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('Clusters in PCA-Reduced Space')
plt.legend()
plt.show()
```



Analyze Results

This was expected; the graph retains most of the important features, but we need to analyze which variables are contributing the most to the PCA direction. We can analyze the loadings to inspect this.

```
# Get PCA loadings
loadings = pca.components_

# Create a dataframe of the loadings
features = ['valence', 'energy', 'danceability', 'loudness', 'tempo']
pca_loadings = pd.DataFrame(loadings, columns=features, index=['PC1', 'PC2'])

print(pca_loadings)
```

```
      valence    energy  danceability  loudness     tempo
PC1  0.938985  0.207813      0.270254  0.042428 -0.016834
```

```
PC2 -0.121554  0.904502     -0.297068  0.220886  0.173382
```

Making sense

The value ranges from [-1, 1] in the PCA values, and we have two principal components that tell us which variable is affecting it the most.

In PC1, valence has the highest negative influence, indicating that PC1 is heavily tied to the emotional positivity of a song. Songs with lower values represent sad, depressing songs. The other variables don't have as much influence as valence, with loudness and tempo minimally affecting PC1, but they are not the dominant driving features.

In PC2, energy has the highest negative influence, showing that PC2 is tied to the intensity and activity of the song. The other variables have less influence.

Songs with:

Low PC1 & Low PC2: Likely sad, calm tracks. High PC1 & Low PC2: Likely happy, calm tracks. Low PC1 & High PC2: Likely sad, intense tracks. High PC1 & High PC2: Likely happy, intense tracks.

```
# Map emotion labels to clusters
cluster_labels = {
    0: "Energetic",
    1: "Joyful",
    2: "Calm",
    3: "Sad",
    4: "Angry"
}

# Assign labels
cluster_summary['Emotion'] = cluster_summary['Cluster'].map(cluster_labels)

#count of each song in the cluster
cluster_counts =  cluster_summary['Emotion'].value_counts()
print(cluster_counts)
```

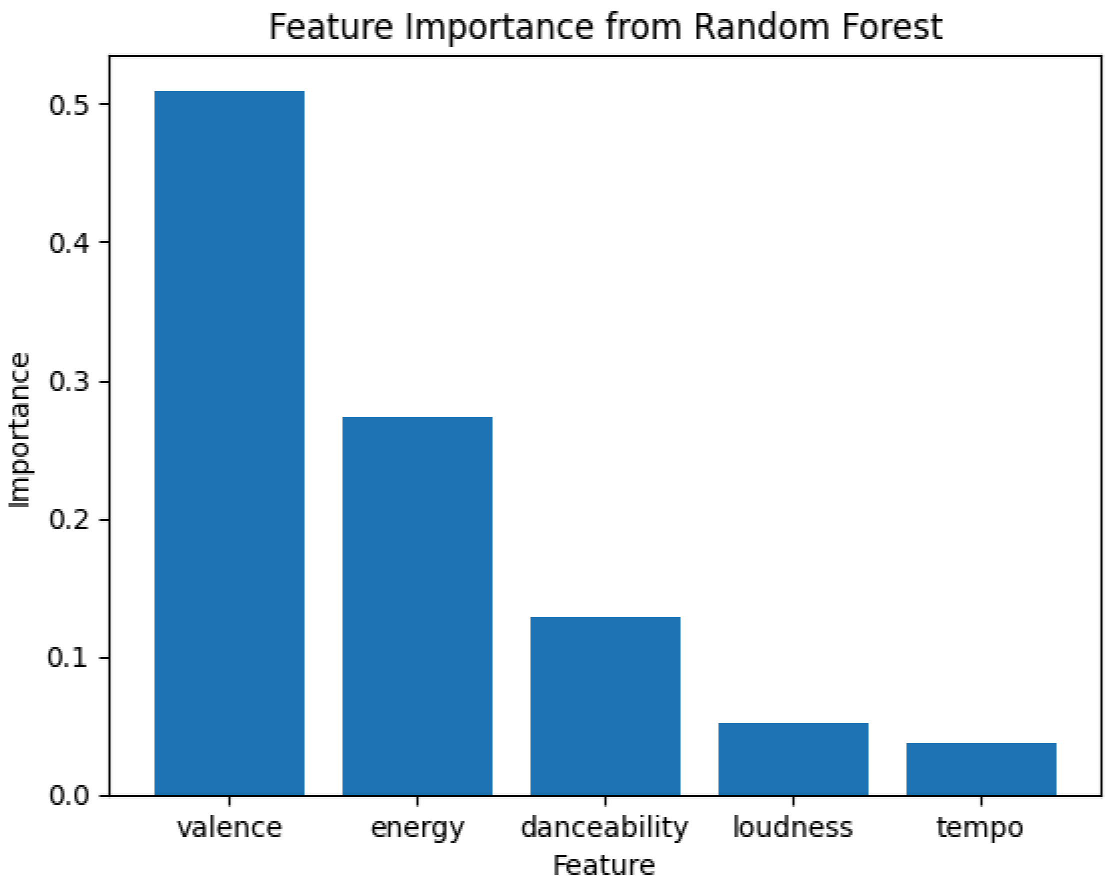
```
Emotion
Joyful       7804
Calm         7093
Energetic    6694
Angry        6630
Sad          4607
Name: count, dtype: int64
```

Analyzing further

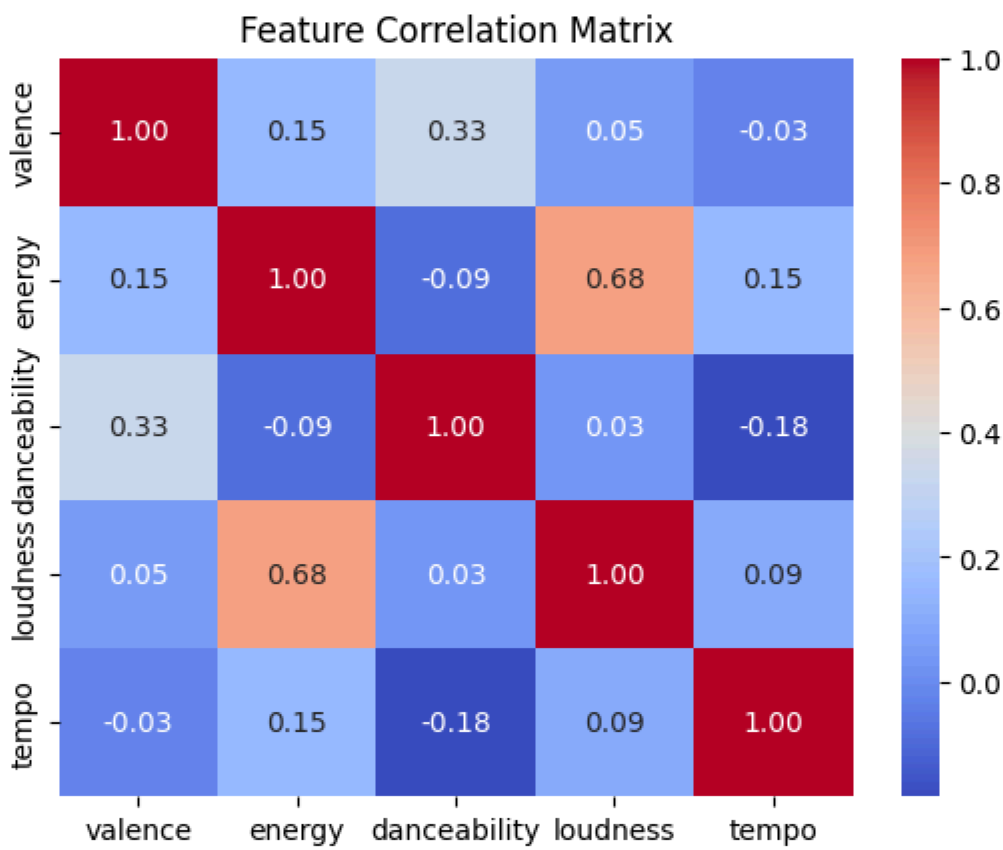
Obtaining an 'importance' level that tells us which variables contribute the most to distinct clusters is important. From the findings, valence and energy play the most significant roles. Loudness is the least important, with a very low value of 0.05. Given that loudness is so low, this variable can be dropped in the further process.

```
from sklearn.ensemble import RandomForestClassifier

X = normalized_data
y = kmeans.labels_  # Cluster labels from k-means

# Train Random Forest
rf = RandomForestClassifier(random_state=42)
rf.fit(X, y)

feature_importances = rf.feature_importances_
feature_names = ['valence', 'energy', 'danceability', 'loudness', 'tempo']

# Display results
for name, importance in zip(feature_names, feature_importances):
    print(f"Feature: {name}, Importance: {importance:.2f}")


plt.bar(feature_names, feature_importances)
plt.title("Feature Importance from Random Forest")
plt.ylabel("Importance")
plt.xlabel("Feature")
plt.show()
```

```
Feature: valence, Importance: 0.51
Feature: energy, Importance: 0.27
Feature: danceability, Importance: 0.13
Feature: loudness, Importance: 0.05
Feature: tempo, Importance: 0.04
```




```
import pandas as pd
import seaborn as sns

# Convert the numpy array back to a DataFrame
columns = ['valence', 'energy', 'danceability', 'loudness', 'tempo']  #feature names
normalized_df = pd.DataFrame(normalized_data, columns=columns)

# Compute pairwise correlation
corr_matrix = normalized_df.corr()

# Visualize the correlation matrix
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Feature Correlation Matrix")
plt.show()
```



Training

Time to train the data now that I have explored the dataset through different technique for emotional classification for the song.

Forest Model

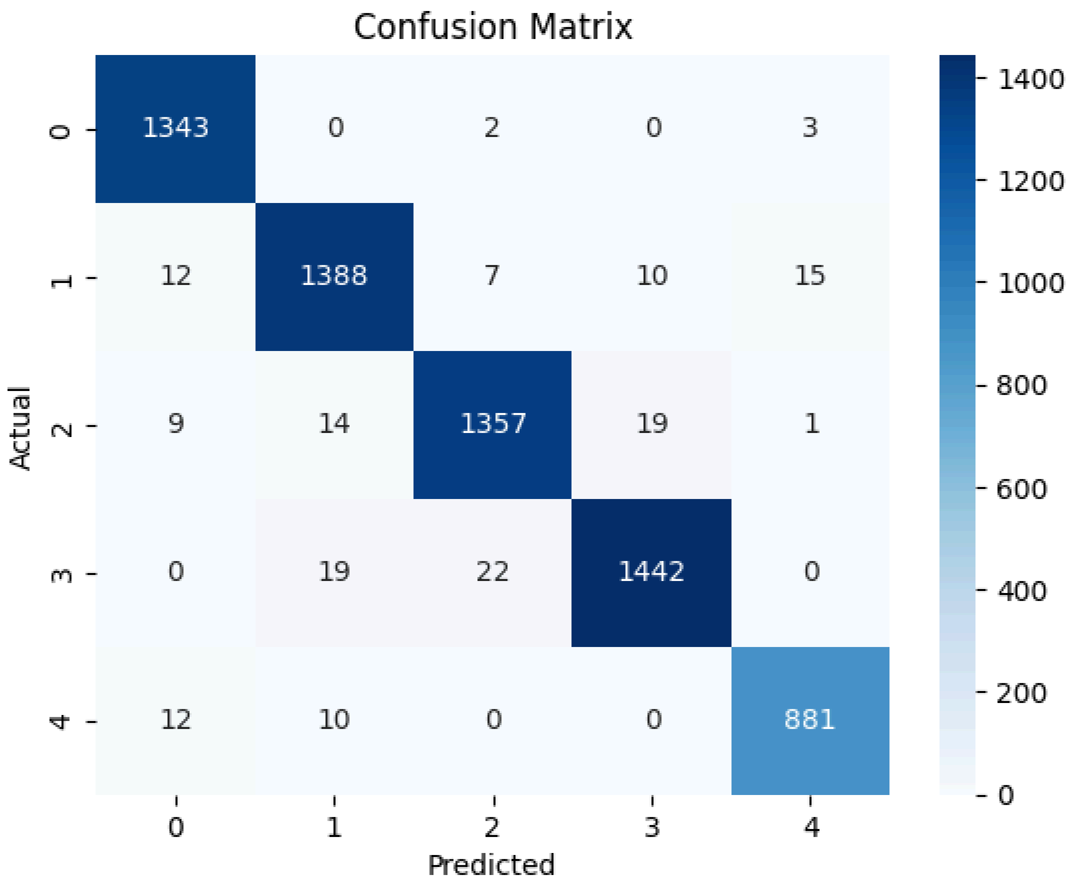
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Input features (X) and target labels (y)
X = normalized_data  # preprocessed feature set
y = cluster_summary['Emotion'] # assigned emotional categories

# Train-test-validation split (60% training, 20% validation, 20% test)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=


# Encode the target labels
encoder = LabelEncoder()
y_train_encoded = encoder.fit_transform(y_train)
y_test_encoded = encoder.transform(y_test)
```

```
from sklearn.ensemble import RandomForestClassifier

# Initialize the model
clf = RandomForestClassifier(random_state=42, n_estimators=200,max_depth=None,min_samples_sp

# Train the model
clf.fit(X_train, y_train)
```

```
RandomForestClassifier
RandomForestClassifier(n_estimators=200, random_state=42)
```

```
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Make predictions
y_pred = clf.predict(X_test)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Detailed classification report
print(classification_report(y_test, y_pred))

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues")
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.9763935424916235
              precision    recall  f1-score   support

       Angry       0.98      1.00      0.99      1348
        Calm       0.97      0.97      0.97      1432
   Energetic       0.98      0.97      0.97      1400
      Joyful       0.98      0.97      0.98      1483
         Sad       0.98      0.98      0.98       903

    accuracy                           0.98      6566
   macro avg       0.98      0.98      0.98      6566
weighted avg       0.98      0.98      0.98      6566
```



Hypertuning Paramaters Forest Model

```
# from sklearn.model_selection import GridSearchCV

# # Define parameter grid, simplify paramter removed other test
# param_grid = {
#     'n_estimators': [100, 200],
#     'max_depth': [None, 10, 20],
#     'min_samples_split': [2, 5]
# }
```

```
# # Initialize GridSearch
# grid_search = GridSearchCV(RandomForestClassifier(random_state=42), param_grid, cv=5, scor

# # Fit GridSearch
# grid_search.fit(X_train, y_train)

# # Best parameters and accuracy
# print("Best Parameters:", grid_search.best_params_)
# print("Best Score:", grid_search.best_score_)
```

```
Best Parameters: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 200}
Best Score: 0.9749693614421965
```

```
#Save the trained model
joblib.dump(clf, 'song_emotion_classifier_forest_model.pkl')

#Load the model later if we need it
clf = joblib.load('song_emotion_classifier_forest_model.pkl')
```

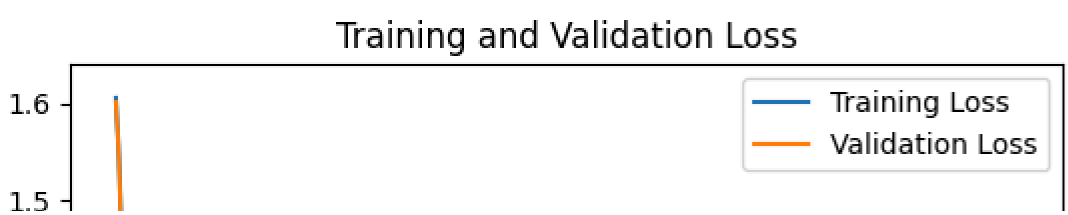
Neural Network MLP

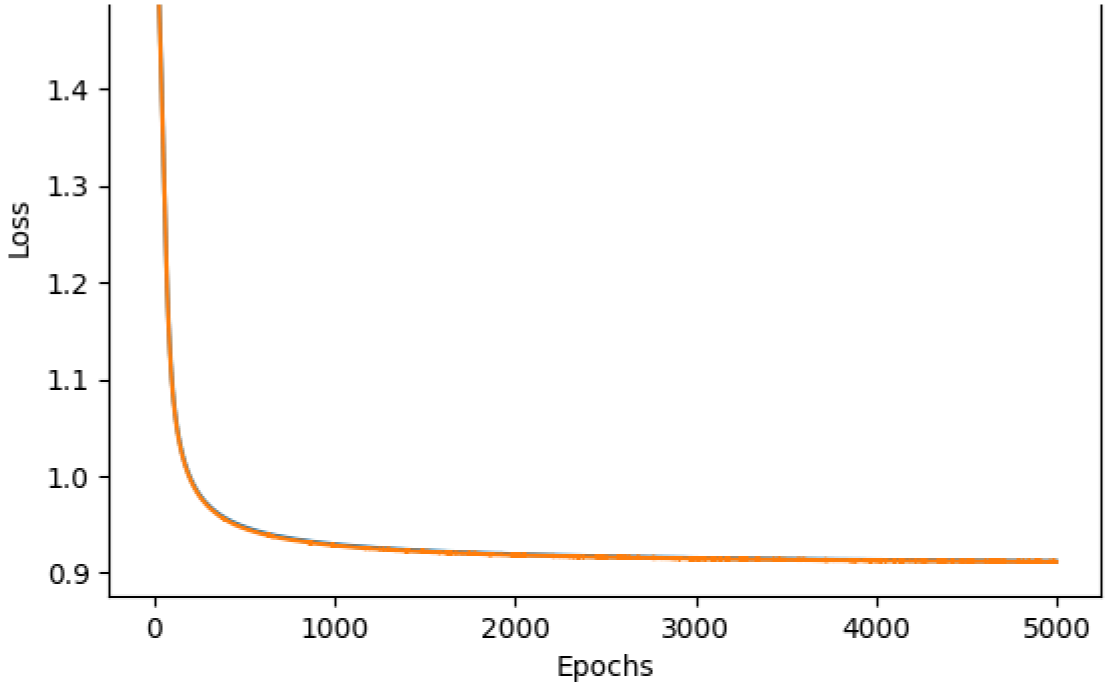
```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Neural network model
class EmotionNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(EmotionNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.softmax = nn.Softmax(dim=1)

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

# Hyperparameters
```

```
input_size = 5  # Number of features
hidden_size = 32  #tuned
output_size = 5  # Number of emotions
learning_rate = 0.01 #tuned
epochs = 5000

# Splitting data into train, validation, and test sets
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=

# Encode the target labels
encoder = LabelEncoder()
y_train_encoded = encoder.fit_transform(y_train)
y_val_encoded = encoder.transform(y_val)
y_test_encoded = encoder.transform(y_test)

# Initialize the model, loss function, and optimizer
model = EmotionNet(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Converting  data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_encoded, dtype=torch.long)

X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val_encoded, dtype=torch.long)

X_test_tensor = torch.tensor(X_test, dtype=torch.float32)

# Track the loss for plotting
train_losses = []
val_losses = []

# Training loop
for epoch in range(epochs):
    model.train()

    # Forward pass
    outputs = model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Store the training loss
    train_losses.append(loss.item())

    # Validation loss
```

```
        model.eval()
        with torch.no_grad():
            val_outputs = model(X_val_tensor)
            val_loss = criterion(val_outputs, y_val_tensor)

        val_losses.append(val_loss.item())

        if (epoch+1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}, Val Loss: {val_loss.ite

    # Plotting the training and validation loss
    plt.plot(range(epochs), train_losses, label='Training Loss')
    plt.plot(range(epochs), val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')
    plt.show()

    # After training, make predictions on the test set
    model.eval()
    y_pred = model(X_test_tensor).argmax(dim=1).numpy()

    # Compute accuracy
    accuracy = accuracy_score(y_test_encoded, y_pred)
    print(f"Accuracy: {accuracy}")
```

```
Epoch [100/5000], Loss: 1.0946, Val Loss: 1.0923
Epoch [200/5000], Loss: 0.9980, Val Loss: 0.9965
Epoch [300/5000], Loss: 0.9697, Val Loss: 0.9682
Epoch [400/5000], Loss: 0.9557, Val Loss: 0.9543
Epoch [500/5000], Loss: 0.9472, Val Loss: 0.9458
Epoch [600/5000], Loss: 0.9414, Val Loss: 0.9401
Epoch [700/5000], Loss: 0.9372, Val Loss: 0.9359
Epoch [800/5000], Loss: 0.9340, Val Loss: 0.9328
Epoch [900/5000], Loss: 0.9314, Val Loss: 0.9302
Epoch [1000/5000], Loss: 0.9293, Val Loss: 0.9282
Epoch [1100/5000], Loss: 0.9275, Val Loss: 0.9265
Epoch [1200/5000], Loss: 0.9260, Val Loss: 0.9250
Epoch [1300/5000], Loss: 0.9247, Val Loss: 0.9236
Epoch [1400/5000], Loss: 0.9236, Val Loss: 0.9229
Epoch [1500/5000], Loss: 0.9226, Val Loss: 0.9218
Epoch [1600/5000], Loss: 0.9218, Val Loss: 0.9209
Epoch [1700/5000], Loss: 0.9210, Val Loss: 0.9202
Epoch [1800/5000], Loss: 0.9203, Val Loss: 0.9196
Epoch [1900/5000], Loss: 0.9196, Val Loss: 0.9188
Epoch [2000/5000], Loss: 0.9190, Val Loss: 0.9182
Epoch [2100/5000], Loss: 0.9185, Val Loss: 0.9177
Epoch [2200/5000], Loss: 0.9180, Val Loss: 0.9174
Epoch [2300/5000], Loss: 0.9176, Val Loss: 0.9168
Epoch [2400/5000], Loss: 0.9171, Val Loss: 0.9164
Epoch [2500/5000], Loss: 0.9168, Val Loss: 0.9161
Epoch [2600/5000], Loss: 0.9164, Val Loss: 0.9159
Epoch [2700/5000], Loss: 0.9161, Val Loss: 0.9154
Epoch [2800/5000], Loss: 0.9157, Val Loss: 0.9151
Epoch [2900/5000], Loss: 0.9154, Val Loss: 0.9148
Epoch [3000/5000], Loss: 0.9152, Val Loss: 0.9146
Epoch [3100/5000], Loss: 0.9149, Val Loss: 0.9143
Epoch [3200/5000], Loss: 0.9147, Val Loss: 0.9141
Epoch [3300/5000], Loss: 0.9144, Val Loss: 0.9138
Epoch [3400/5000], Loss: 0.9142, Val Loss: 0.9136
Epoch [3500/5000], Loss: 0.9140, Val Loss: 0.9135
Epoch [3600/5000], Loss: 0.9139, Val Loss: 0.9136
Epoch [3700/5000], Loss: 0.9136, Val Loss: 0.9131
Epoch [3800/5000], Loss: 0.9134, Val Loss: 0.9130
Epoch [3900/5000], Loss: 0.9133, Val Loss: 0.9129
Epoch [4000/5000], Loss: 0.9131, Val Loss: 0.9127
Epoch [4100/5000], Loss: 0.9130, Val Loss: 0.9124
Epoch [4200/5000], Loss: 0.9128, Val Loss: 0.9123
Epoch [4300/5000], Loss: 0.9128, Val Loss: 0.9123
Epoch [4400/5000], Loss: 0.9125, Val Loss: 0.9121
Epoch [4500/5000], Loss: 0.9124, Val Loss: 0.9119
Epoch [4600/5000], Loss: 0.9123, Val Loss: 0.9118
Epoch [4700/5000], Loss: 0.9121, Val Loss: 0.9117
Epoch [4800/5000], Loss: 0.9120, Val Loss: 0.9115
Epoch [4900/5000], Loss: 0.9119, Val Loss: 0.9116
Epoch [5000/5000], Loss: 0.9118, Val Loss: 0.9113
```






```
Accuracy: 0.9984770027413951
```

```
# from sklearn.base import BaseEstimator
# from sklearn.neural_network import MLPClassifier
# from sklearn.model_selection import GridSearchCV
# import torch
# import torch.nn as nn
# import torch.optim as optim
# from sklearn.preprocessing import LabelEncoder
# from sklearn.metrics import accuracy_score
# import numpy as np
# from sklearn.model_selection import train_test_split

# # Neural network model
# class EmotionNet(nn.Module):
#     def __init__(self, input_size, hidden_size, output_size):
#         super(EmotionNet, self).__init__()
#         self.fc1 = nn.Linear(input_size, hidden_size)
#         self.relu = nn.ReLU()
#         self.fc2 = nn.Linear(hidden_size, output_size)
#         self.softmax = nn.Softmax(dim=1)

#     def forward(self, x):
#         x = self.fc1(x)
#         x = self.relu(x)
#         x = self.fc2(x)
#         x = self.softmax(x)
#         return x

# # Wrapper class to use PyTorch model in GridSearchCV
```

```
# class MLPWrapper(BaseEstimator):
#     def __init__(self, input_size, hidden_size, output_size, learning_rate=0.001, epochs=1
#         self.input_size = input_size
#         self.hidden_size = hidden_size
#         self.output_size = output_size
#         self.learning_rate = learning_rate
#         self.epochs = epochs

#     def fit(self, X, y):
#         # Convert X and y to PyTorch tensors
#         X_tensor = torch.tensor(X, dtype=torch.float32)
#         y_tensor = torch.tensor(y, dtype=torch.long)

#         # Initialize the model, loss function, and optimizer
#         self.model = EmotionNet(self.input_size, self.hidden_size, self.output_size)
#         self.criterion = nn.CrossEntropyLoss()
#         self.optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)

#         # Training loop
#         for epoch in range(self.epochs):
#             self.model.train()
#             outputs = self.model(X_tensor)
#             loss = self.criterion(outputs, y_tensor)

#             # Backward pass and optimization
#             self.optimizer.zero_grad()
#             loss.backward()
#             self.optimizer.step()

#         return self

#     def predict(self, X):
#         self.model.eval()
#         X_tensor = torch.tensor(X, dtype=torch.float32)
#         y_pred = self.model(X_tensor).argmax(dim=1).numpy()
#         return y_pred

# # Define hyperparameter grid for GridSearchCV
# param_grid = {
#     'hidden_size': [32, 64],
#     'learning_rate': [0.01],
#     'epochs': [100,500,1000,5000]
# }

# # Prepare data
# X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_state=42)
# X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_stat

# # Encode target labels
# encoder = LabelEncoder()
# y_train_encoded = encoder.fit_transform(y_train)
```

```
# y_val_encoded = encoder.transform(y_val)
# y_test_encoded = encoder.transform(y_test)

# # Initialize the MLPWrapper and GridSearchCV
# mlp_wrapper = MLPWrapper(input_size=X_train.shape[1], hidden_size=64, output_size=len(encc
# grid_search = GridSearchCV(mlp_wrapper, param_grid, cv=3, scoring='accuracy', n_jobs=-1)

# # Fit GridSearchCV
# grid_search.fit(X_train, y_train_encoded)

# # Best parameters and score
# print("Best Parameters:", grid_search.best_params_)
# print("Best Score:", grid_search.best_score_)

# # Make predictions with the best model
# best_model = grid_search.best_estimator_
# y_pred = best_model.predict(X_test)

# # Compute accuracy
# accuracy = accuracy_score(y_test_encoded, y_pred)
# print(f"Test Accuracy: {accuracy}")
```

Saving and Loading the Model

```
torch.save(model.state_dict(), 'emotion_model.pth') #save the model to pth file
def load_model(model_class, model_path, input_size, hidden_size, output_size):
    model = model_class(input_size, hidden_size, output_size)
    model.load_state_dict(torch.load(model_path, weights_only=True))
    model.eval()  # Set to evaluation mode
    return model

# Example usage:
input_size = 5
hidden_size = 32
output_size = 5
model = load_model(EmotionNet, 'emotion_model.pth', input_size, hidden_size, output_size)
```

Testing

```
import numpy as np
import torch
from sklearn.preprocessing import LabelEncoder
import joblib
```

```
# Load the scaler
scaler = joblib.load('minmax_scaler.pkl')

# Define emotion to label mapping
emotion_to_label = {'Angry': 0, 'Calm': 1, 'Energetic': 2, 'Joyful': 3, 'Sad': 4}

def predict_emotion(user_mood):
    # Map user input to emotion label (numeric encoding)
    emotion_map = {
        "Energetic": [0.550727, 0.830376, 0.581754, 0.861508, 0.541825],
        "Joyful": [0.810146, 0.768622, 0.746998, 0.844189, 0.499412],
        "Calm": [0.564478, 0.567334, 0.764183, 0.807609, 0.469615],
        "Sad": [0.269011, 0.435314, 0.594407, 0.767869, 0.485511],
        "Angry": [0.250470, 0.806265, 0.601295, 0.860896, 0.525102]
    }

    # Get the corresponding feature vector for the selected emotion
    emotion_label = emotion_map.get(user_mood, None)
    if emotion_label is None:
        print("Invalid emotion selected!")
        return None

    # Convert the emotion features into a numpy array and then into a torch tensor
    emotion_input = np.array([emotion_label])
    emotion_input_tensor = torch.tensor(emotion_input, dtype=torch.float32)

    # Make prediction using the model
    with torch.no_grad():
        outputs = model(emotion_input_tensor)  # Forward pass
        emotion_pred = outputs.argmax(dim=1).numpy()  # Get the predicted class
    predicted_emotion = encoder.inverse_transform(emotion_pred)
    return predicted_emotion[0]

def assign_emotion_to_songs(df, model, encoder):
    for _, row in df.iterrows():
        # Extract features
        features = np.array([row[['valence', 'energy', 'danceability', 'loudness', 'tempo'
        print(f"Input features: {features}")  # Debug statement

        # Normalize features using the saved scaler
        normalized_features = scaler.transform(features)
        features_tensor = torch.tensor(normalized_features, dtype=torch.float32)

        # Predict emotion
        with torch.no_grad():
            outputs = model(features_tensor)  # Forward pass
            pred = outputs.argmax(dim=1).numpy()  # Get the predicted class
        predicted_emotion = encoder.inverse_transform(pred)

        print(f"Prediction (encoded): {pred}")  # Debug statement
```