## Cell 1 - Imports

In [1]:
```
import numpy as np
import pandas as pd
import xgboost as xgb
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import confusion_matrix
from keras.utils import to_categorical
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from keras.models import Sequential
from keras import layers
from keras.optimizers import Adam
```

## Cell 2 - Processing the file

In [2]:
```
genres = ['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz', 'metal', 'pop', 'reggae', 'rock']

# Load the CSV file
data = pd.read_csv("Data/features_3_sec.csv")
X = data.drop(columns=["label", "filename"])
y = data["label"]

# Extract song IDs from filenames
data['song_id'] = data['filename'].apply(lambda x: x.rsplit('.', 2)[0])  # Extract "blues.00000" from "blues.000

# Group clips by song ID
song_to_clips = {}
for song_id, group in data.groupby('song_id'):
    # Store all features and labels for each clip
    song_to_clips[song_id] = {
        'features': group.drop(columns=['filename', 'song_id', 'label']).values,  # Extract features (all colum
        'labels': group['label'].values  # Extract labels
    }

# Split song IDs into training and test sets
song_ids = list(song_to_clips.keys())
train_ids, test_ids = train_test_split(song_ids, test_size=0.2, random_state=42)

# Prepare training and test data
X_train, y_train, X_test, y_test = [], [], [], []

# Assign clips based on the train-test split
for song_id in song_ids:
    clips = song_to_clips[song_id]
    if song_id in train_ids:
        X_train.extend(clips['features'])  # Add all features for this song to the training set
        y_train.extend(clips['labels'])    # Add all labels for this song to the training set
    else:
        X_test.extend(clips['features'])  # Add all features for this song to the test set
        y_test.extend(clips['labels'])    # Add all labels for this song to the test set

# Convert to numpy arrays
X_train = np.array(X_train)
y_train = np.array(y_train)
X_test = np.array(X_test)
y_test = np.array(y_test)

# Print shapes to verify
print(f"X_train shape: {X_train.shape}, y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}, y_test shape: {y_test.shape}")
```

```
X_train shape: (7993, 58), y_train shape: (7993,)
X_test shape: (1997, 58), y_test shape: (1997,)
```

## Cell 3 - Feature extraction

In [3]:
```
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)


# new data frame with the new scaled data.
cols = X.columns
```

```
min_max_scaler = preprocessing.MinMaxScaler()
np_scaled = min_max_scaler.fit_transform(X)

X = pd.DataFrame(np_scaled, columns = cols)
pca = PCA(n_components=3)
principalComponents = pca.fit_transform(X)
X = np.concatenate((X, pca.fit_transform(X)), axis=1)


X.shape
```

Out[3]: (9990, 61)

## Cell 5 - Train test split

In [4]:
```
# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(x_train)
X_test_scaled = scaler.transform(x_test)

# One-hot encode the labels
y_train_encoded = to_categorical(y_train, num_classes=len(genres))
y_test_encoded = to_categorical(y_test, num_classes=len(genres))

# Check the shapes of the train and test sets - commented out for readability
# Uncomment to see the shape and number of files read in

# print(f"Shape of x_train: {x_train.shape}")
# print(f"Shape of x_test: {x_test.shape}")
# print(f"Shape of y_train: {y_train.shape}")
# print(f"Shape of y_test: {y_test.shape}")
# print(f"Shape of y_train_encoded: {y_train_encoded.shape}")
# print(f"Shape of y_test_encoded: {y_test_encoded.shape}")
```

In [5]:
```
print(X_test_scaled.shape)
print(X_train_scaled.shape)
```

```
(1998, 61)
(7992, 61)
```

## Cell 7 - Training SVM model

In [ ]:
```
# Train and evaluate SVM model
svm_model = SVC(probability=True)
svm_model.fit(X_train_scaled, y_train)
y_pred_svm = svm_model.predict(X_test_scaled)
svc_accuracy = accuracy_score(y_test, y_pred_svm)
print(f"SVM accuracy: {svc_accuracy:.3f}")
```

```
SVM accuracy: 0.8593593593593594
```

## Cell 8 - Defining CNN

In [7]:
```
# Define the CNN model architecture
cnn_model = Sequential([
    # First convolutional layer with 'same' padding
    layers.Input(shape = (61,1)),
    layers.Conv1D(64, 3, activation='relu', padding='same',),
    layers.MaxPooling1D(2, padding='same'),  # MaxPooling2D with same padding

    # Second convolutional layer with 'same' padding
    layers.Conv1D(128, 3, activation='relu', padding='same'),
    layers.MaxPooling1D(2, padding='same'),

    # Third convolutional layer with 'same' padding
    layers.Conv1D(256, 3, activation='relu', padding='same'),
    layers.MaxPooling1D(2, padding='same'),

    # Flatten the output of the convolutional layers
    layers.Flatten(),

    # Fully connected layers
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.5),

    # Output layer with softmax activation
    layers.Dense(len(genres), activation='softmax')
])
```

```python
# Compile the model
cnn_model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

# Summarize the model architecture
#cnn_model.summary()
print("Model compiled")
```

```
Model compiled
```

## Cell 9 - Training CNN model

In [8]:
```python
# Train the CNN model
cnn_model.fit(x_train, y_train_encoded, validation_data=(x_test, y_test_encoded), epochs=30, batch_size=32)
```

```
Epoch 1/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 14ms/step - accuracy: 0.2959 - loss: 1.8735 - val_accuracy: 0.5586 - val_loss: 1
.2105
Epoch 2/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.5760 - loss: 1.1829 - val_accuracy: 0.6702 - val_loss: 0
.9430
Epoch 3/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.6685 - loss: 0.9604 - val_accuracy: 0.7087 - val_loss: 0
.8352
Epoch 4/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.7120 - loss: 0.7923 - val_accuracy: 0.7548 - val_loss: 0
.7146
Epoch 5/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.7543 - loss: 0.6819 - val_accuracy: 0.7878 - val_loss: 0
.6150
Epoch 6/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 14ms/step - accuracy: 0.7898 - loss: 0.5931 - val_accuracy: 0.7908 - val_loss: 0
.6079
Epoch 7/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.8032 - loss: 0.5537 - val_accuracy: 0.8468 - val_loss: 0
.4772
Epoch 8/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.8494 - loss: 0.4306 - val_accuracy: 0.8463 - val_loss: 0
.4738
Epoch 9/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8699 - loss: 0.3775 - val_accuracy: 0.8614 - val_loss: 0
.4066
Epoch 10/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8827 - loss: 0.3266 - val_accuracy: 0.8669 - val_loss: 0
.4081
Epoch 11/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.8966 - loss: 0.3024 - val_accuracy: 0.8814 - val_loss: 0
.3637
Epoch 12/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.9118 - loss: 0.2687 - val_accuracy: 0.8884 - val_loss: 0
.3317
Epoch 13/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.9196 - loss: 0.2261 - val_accuracy: 0.8904 - val_loss: 0
.3500
Epoch 14/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.9340 - loss: 0.1968 - val_accuracy: 0.8804 - val_loss: 0
.3717
Epoch 15/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.9405 - loss: 0.1685 - val_accuracy: 0.8879 - val_loss: 0
.3654
Epoch 16/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.9357 - loss: 0.1770 - val_accuracy: 0.8994 - val_loss: 0
.3300
Epoch 17/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - accuracy: 0.9559 - loss: 0.1422 - val_accuracy: 0.8909 - val_loss: 0
.3462
Epoch 18/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.9537 - loss: 0.1349 - val_accuracy: 0.9059 - val_loss: 0
.3180
Epoch 19/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 14ms/step - accuracy: 0.9622 - loss: 0.1141 - val_accuracy: 0.9084 - val_loss: 0
.3268
Epoch 20/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.9629 - loss: 0.1058 - val_accuracy: 0.9164 - val_loss: 0
.3114
Epoch 21/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - accuracy: 0.9647 - loss: 0.1060 - val_accuracy: 0.9109 - val_loss: 0
.3202
Epoch 22/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.9641 - loss: 0.1010 - val_accuracy: 0.9179 - val_loss: 0
.3072
Epoch 23/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.9653 - loss: 0.0989 - val_accuracy: 0.9054 - val_loss: 0
.3584
```

```
Epoch 24/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.9742 - loss: 0.0723 - val_accuracy: 0.9119 - val_loss: 0
.3532
Epoch 25/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - accuracy: 0.9722 - loss: 0.0791 - val_accuracy: 0.8974 - val_loss: 0
.3730
Epoch 26/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - accuracy: 0.9669 - loss: 0.0869 - val_accuracy: 0.9074 - val_loss: 0
.3558
Epoch 27/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - accuracy: 0.9736 - loss: 0.0755 - val_accuracy: 0.9174 - val_loss: 0
.3452
Epoch 28/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - accuracy: 0.9744 - loss: 0.0699 - val_accuracy: 0.9124 - val_loss: 0
.3592
Epoch 29/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - accuracy: 0.9790 - loss: 0.0613 - val_accuracy: 0.9179 - val_loss: 0
.3382
Epoch 30/30
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - accuracy: 0.9761 - loss: 0.0680 - val_accuracy: 0.9214 - val_loss: 0
.3085
```

Out[8]:
```
<keras.src.callbacks.history.History at 0x1a96e91dfd0>
```

## Training a Dense Neural Network

In [9]:
```
import tensorflow as tf
# Define an enhanced MLP architecture with better regularization and optimization
def create_mlp(input_shape, num_classes):
    model = Sequential([
        layers.Input(shape=input_shape),

        # Hidden layers with batch norm and dropout
        layers.Dense(1024, activation='relu', kernel_initializer='he_normal'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),

        layers.Dense(512, activation='relu', kernel_initializer='he_normal'),
        layers.BatchNormalization(),
        layers.Dropout(0.4),

        layers.Dense(256, activation='relu', kernel_initializer='he_normal'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        # Additional hidden layer
        layers.Dense(128, activation='relu', kernel_initializer='he_normal'),
        layers.BatchNormalization(),

        # Output layer
        layers.Dense(num_classes, activation='softmax')
    ])

    # Custom optimizer configuration
    optimizer = Adam(
        learning_rate=0.001,
        beta_1=0.9,
        beta_2=0.999,
        clipnorm=1.0  # Gradient clipping
    )

    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=['accuracy',
                tf.keras.metrics.Precision(name='precision'),
                tf.keras.metrics.Recall(name='recall')]
    )
    return model

mlp_model = create_mlp((x_train.shape[1],), len(genres))
mlp_model.summary()
```

**Model: "sequential_1"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_2 (Dense) | (None, 1024) | 63,488 |
| batch_normalization (BatchNormalization) | (None, 1024) | 4,096 |
| dropout_1 (Dropout) | (None, 1024) | 0 |
| dense_3 (Dense) | (None, 512) | 524,800 |
| batch_normalization_1 (BatchNormalization) | (None, 512) | 2,048 |
| dropout_2 (Dropout) | (None, 512) | 0 |
| dense_4 (Dense) | (None, 256) | 131,328 |
| batch_normalization_2 (BatchNormalization) | (None, 256) | 1,024 |
| dropout_3 (Dropout) | (None, 256) | 0 |
| dense_5 (Dense) | (None, 128) | 32,896 |
| batch_normalization_3 (BatchNormalization) | (None, 128) | 512 |
| dense_6 (Dense) | (None, 10) | 1,290 |

**Total params:** 761,482 (2.90 MB)

**Trainable params:** 757,642 (2.89 MB)

**Non-trainable params:** 3,840 (15.00 KB)

In [10]:

```
from keras.callbacks import EarlyStopping, ReduceLROnPlateau
# Add callbacks for better training
mlp_callbacks = [
    EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True),
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6),
]

mlp_model.fit(
    x_train, y_train_encoded,
    validation_data=(x_test, y_test_encoded),
    epochs=30,
    batch_size=128,  # Larger batch size
    callbacks=mlp_callbacks,
)
```

```
Epoch 1/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - accuracy: 0.3585 - loss: 1.9251 - precision: 0.5199 - recall: 0.2228 - val_accuracy: 0.2718 - val_loss: 1.9234 - val_precision: 0.8222 - val_recall: 0.0556 - learning_rate: 0.0010
Epoch 2/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.6030 - loss: 1.1138 - precision: 0.7326 - recall: 0.4648 - val_accuracy: 0.3654 - val_loss: 1.7166 - val_precision: 0.5981 - val_recall: 0.1572 - learning_rate: 0.0010
Epoch 3/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.6766 - loss: 0.9236 - precision: 0.7796 - recall: 0.5644 - val_accuracy: 0.4269 - val_loss: 1.5386 - val_precision: 0.5514 - val_recall: 0.2603 - learning_rate: 0.0010
Epoch 4/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.6853 - loss: 0.8802 - precision: 0.7793 - recall: 0.5919 - val_accuracy: 0.5185 - val_loss: 1.2524 - val_precision: 0.6302 - val_recall: 0.3744 - learning_rate: 0.0010
Epoch 5/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.7240 - loss: 0.7768 - precision: 0.8005 - recall: 0.6414 - val_accuracy: 0.6406 - val_loss: 0.9697 - val_precision: 0.7473 - val_recall: 0.5165 - learning_rate: 0.0010
Epoch 6/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.7395 - loss: 0.7317 - precision: 0.8059 - recall: 0.6563 - val_accuracy: 0.6862 - val_loss: 0.8704 - val_precision: 0.7885 - val_recall: 0.5746 - learning_rate: 0.0010
Epoch 7/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.7603 - loss: 0.6864 - precision: 0.8260 - recall: 0.6855 - val_accuracy: 0.7663 - val_loss: 0.6451 - val_precision: 0.8336 - val_recall: 0.6922 - learning_rate: 0.0010
Epoch 8/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.7634 - loss: 0.6600 - precision: 0.8235 - recall: 0.6953 - val_accuracy: 0.8053 - val_loss: 0.5356 - val_precision: 0.8750 - val_recall: 0.7533 - learning_rate: 0.0010
Epoch 9/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.7867 - loss: 0.6049 - precision: 0.8451 - recall: 0.7250 - val_accuracy: 0.8198 - val_loss: 0.5195 - val_precision: 0.8685 - val_recall: 0.7638 - learning_rate: 0.0010
Epoch 10/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.7865 - loss: 0.5912 - precision: 0.8408 - recall: 0.7354 - val_accuracy: 0.8373 - val_loss: 0.4737 - val_precision: 0.8736 - val_recall: 0.7918 - learning_rate: 0.0010
Epoch 11/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8097 - loss: 0.5378 - precision: 0.8557 - recall: 0.7592 -
```

```
val_accuracy: 0.8313 - val_loss: 0.4707 - val_precision: 0.8766 - val_recall: 0.7928 - learning_rate: 0.0010
Epoch 12/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - accuracy: 0.8215 - loss: 0.5059 - precision: 0.8642 - recall: 0.7716 -
val_accuracy: 0.8463 - val_loss: 0.4473 - val_precision: 0.8772 - val_recall: 0.8148 - learning_rate: 0.0010
Epoch 13/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8150 - loss: 0.5009 - precision: 0.8585 - recall: 0.7783 -
val_accuracy: 0.8564 - val_loss: 0.4395 - val_precision: 0.8846 - val_recall: 0.8248 - learning_rate: 0.0010
Epoch 14/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8278 - loss: 0.4944 - precision: 0.8733 - recall: 0.7889 -
val_accuracy: 0.8674 - val_loss: 0.3935 - val_precision: 0.8966 - val_recall: 0.8418 - learning_rate: 0.0010
Epoch 15/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8386 - loss: 0.4540 - precision: 0.8773 - recall: 0.7993 -
val_accuracy: 0.8784 - val_loss: 0.3649 - val_precision: 0.9031 - val_recall: 0.8584 - learning_rate: 0.0010
Epoch 16/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8367 - loss: 0.4622 - precision: 0.8679 - recall: 0.7990 -
val_accuracy: 0.8789 - val_loss: 0.3608 - val_precision: 0.9035 - val_recall: 0.8579 - learning_rate: 0.0010
Epoch 17/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8481 - loss: 0.4236 - precision: 0.8817 - recall: 0.8170 -
val_accuracy: 0.8574 - val_loss: 0.4077 - val_precision: 0.8869 - val_recall: 0.8363 - learning_rate: 0.0010
Epoch 18/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8427 - loss: 0.4440 - precision: 0.8768 - recall: 0.8090 -
val_accuracy: 0.8869 - val_loss: 0.3413 - val_precision: 0.9090 - val_recall: 0.8599 - learning_rate: 0.0010
Epoch 19/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.8571 - loss: 0.3956 - precision: 0.8865 - recall: 0.8247 -
val_accuracy: 0.8769 - val_loss: 0.3759 - val_precision: 0.8998 - val_recall: 0.8539 - learning_rate: 0.0010
Epoch 20/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8544 - loss: 0.4033 - precision: 0.8853 - recall: 0.8229 -
val_accuracy: 0.8729 - val_loss: 0.3577 - val_precision: 0.9015 - val_recall: 0.8519 - learning_rate: 0.0010
Epoch 21/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8629 - loss: 0.3859 - precision: 0.8901 - recall: 0.8308 -
val_accuracy: 0.8904 - val_loss: 0.3401 - val_precision: 0.9068 - val_recall: 0.8719 - learning_rate: 0.0010
Epoch 22/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8716 - loss: 0.3636 - precision: 0.8963 - recall: 0.8422 -
val_accuracy: 0.8894 - val_loss: 0.3232 - val_precision: 0.9081 - val_recall: 0.8749 - learning_rate: 0.0010
Epoch 23/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8706 - loss: 0.3622 - precision: 0.8958 - recall: 0.8431 -
val_accuracy: 0.9029 - val_loss: 0.2976 - val_precision: 0.9195 - val_recall: 0.8804 - learning_rate: 0.0010
Epoch 24/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8701 - loss: 0.3555 - precision: 0.8950 - recall: 0.8474 -
val_accuracy: 0.9009 - val_loss: 0.2954 - val_precision: 0.9192 - val_recall: 0.8884 - learning_rate: 0.0010
Epoch 25/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8825 - loss: 0.3453 - precision: 0.9029 - recall: 0.8571 -
val_accuracy: 0.9094 - val_loss: 0.2856 - val_precision: 0.9205 - val_recall: 0.8984 - learning_rate: 0.0010
Epoch 26/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8813 - loss: 0.3344 - precision: 0.9021 - recall: 0.8567 -
val_accuracy: 0.9099 - val_loss: 0.2776 - val_precision: 0.9228 - val_recall: 0.8979 - learning_rate: 0.0010
Epoch 27/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8902 - loss: 0.3075 - precision: 0.9115 - recall: 0.8703 -
val_accuracy: 0.8959 - val_loss: 0.3153 - val_precision: 0.9084 - val_recall: 0.8839 - learning_rate: 0.0010
Epoch 28/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.8910 - loss: 0.3175 - precision: 0.9106 - recall: 0.8661 -
val_accuracy: 0.9064 - val_loss: 0.2757 - val_precision: 0.9208 - val_recall: 0.8959 - learning_rate: 0.0010
Epoch 29/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8979 - loss: 0.2912 - precision: 0.9187 - recall: 0.8756 -
val_accuracy: 0.9154 - val_loss: 0.2598 - val_precision: 0.9290 - val_recall: 0.9039 - learning_rate: 0.0010
Epoch 30/30
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - accuracy: 0.8973 - loss: 0.2896 - precision: 0.9152 - recall: 0.8785 -
val_accuracy: 0.9134 - val_loss: 0.2607 - val_precision: 0.9255 - val_recall: 0.9014 - learning_rate: 0.0010
```

Out[10]:
```
<keras.src.callbacks.history.History at 0x1a9754d0ca0>
```

In [11]:
```
y_pred_cnn = mlp_model.predict(x_test)
y_pred_cnn = np.argmax(y_pred_cnn, axis=1)
accuracy_cnn = accuracy_score(y_test, y_pred_cnn)
print(f"Accuracy (CNN): {accuracy_cnn:.3f}")
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
Accuracy (CNN): 0.915
```

## Cell 10 - XGBoost model

In [12]:
```
xgb_model = xgb.XGBClassifier(n_estimators=1000, learning_rate=0.05)
xgb_model.fit(x_train, y_train)
y_pred_xgb = xgb_model.predict(x_test)
```

## Random forest model

In [13]:
```
# Initialize the Random Forest Classifier
rf_model = RandomForestClassifier(
    n_estimators=100,
    max_depth=None,
    min_samples_split=2,
```

```
    min_samples_leaf=1,
    random_state=42
)

# Train the model
rf_model.fit(x_train, y_train)

# Make predictions on the test set
y_pred_rf = rf_model.predict(x_test)

# Evaluate the model
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Accuracy: {accuracy_rf:.3f}")
```

```
Accuracy: 0.874
```

## Cell 11 - Evaluate models

In [14]:

```
# Predict using the CNN model
y_pred_cnn = cnn_model.predict(x_test)
y_pred_cnn = np.argmax(y_pred_cnn, axis=1)
y_pred_mlp = mlp_model.predict(x_test)
y_pred_mlp = np.argmax(y_pred_mlp, axis=1)

# Calculate accuracy
accuracy_cnn = accuracy_score(y_test, y_pred_cnn)
print(f"Accuracy (CNN): {accuracy_cnn:.3f}")

accuracy_mlp = accuracy_score(y_test, y_pred_mlp)
print(f"Accuracy (DNN): {accuracy_mlp:.3f}")

accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f"Accuracy (SVM): {accuracy_svm:.3f}")

accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Accuracy (RFC): {accuracy_rf:.3f}")

xgb_accuracy = accuracy_score(y_test, y_pred_xgb)
print(f"Accuracy (XGB): {xgb_accuracy:.3f}")
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Accuracy (CNN): 0.921
Accuracy (DNN): 0.915
Accuracy (SVM): 0.859
Accuracy (RFC): 0.874
Accuracy (XGB): 0.919
```

## Cell 12 - Creating an ensemble using soft voting

In [15]:

```
# Get predictions from each model
xgb_preds_proba = xgb_model.predict_proba(x_test)
rf_preds_proba = rf_model.predict_proba(x_test)
svm_preds_proba = svm_model.predict_proba(X_test_scaled)
cnn_preds_proba = cnn_model.predict(x_test)
mlp_preds_proba = mlp_model.predict(x_test)

# Average the predictions (soft voting)
avg_preds_proba = (xgb_preds_proba + svm_preds_proba + cnn_preds_proba +  rf_preds_proba + mlp_preds_proba) / 5

# Convert probabilities to class predictions
ensemble_preds = np.argmax(avg_preds_proba, axis=1)

# Evaluate the ensemble performance
soft_ensemble_accuracy = accuracy_score(y_test, ensemble_preds)
print(f"Ensemble Accuracy: {soft_ensemble_accuracy:.3f}")
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Ensemble Accuracy: 0.938
```

In [16]:

```
print(y_pred_svm)
print(y_pred_cnn)
```

```
[8 5 0 ... 4 3 8]
[4 5 0 ... 4 3 8]
```

In [17]:

```
print("The predictions before averaging", avg_preds_proba.shape)
print("The predictions after averaging", ensemble_preds.shape)
```

```
The predictions before averaging (1998, 10)
The predictions after averaging (1998,)
```

## Cell 13 - Creating an ensemble using hard voting

In [18]:
```
from scipy.stats import mode
from sklearn.metrics import accuracy_score
import numpy as np

# Step 1: Get class predictions from each model
xgb_preds = xgb_model.predict(x_test)  # Class labels
rf_preds = rf_model.predict(x_test)
svm_preds = svm_model.predict(X_test_scaled)
cnn_preds = np.argmax(cnn_model.predict(x_test), axis=1)  # Convert CNN probs to class labels
mlp_preds = np.argmax(mlp_model.predict(x_test), axis=1)  # Convert DNN probs to class labels


# Step 2: Stack predictions
all_preds = np.array([xgb_preds, svm_preds, cnn_preds, rf_preds + mlp_preds])  # Shape: (3, num_samples)

# Step 3: Perform majority voting (hard voting)
ensemble_preds, _ = mode(all_preds, axis=0, keepdims=True)  # Get most common class per sample
ensemble_preds = ensemble_preds.flatten()  # Convert to 1D array

# Step 4: Evaluate accuracy
hard_ensemble_accuracy = accuracy_score(y_test, ensemble_preds)
print(f"Hard Voting Ensemble Accuracy: {hard_ensemble_accuracy:.3f}")
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Hard Voting Ensemble Accuracy: 0.920
```

In [19]:
```
print("The predictions before averaging", all_preds.shape)
print("The predictions after averaging", ensemble_preds.shape)
```

```
The predictions before averaging (4, 1998)
The predictions after averaging (1998,)
```

## Weighted voting

In [20]:
```
import numpy as np
from sklearn.metrics import accuracy_score

# Define the weights based on model accuracies
weights = {
    #'mlp': accuracy_mlp,
    'cnn': accuracy_cnn,
    'svm': accuracy_svm,
    'rf': accuracy_rf,
    'xgb': xgb_accuracy
}

# Normalize the weights so they sum to 1
total_weight = sum(weights.values())
weights = {k: v / total_weight for k, v in weights.items()}

# Get predictions from each model
xgb_preds_proba = xgb_model.predict_proba(x_test)
rf_preds_proba = rf_model.predict_proba(x_test)
svm_preds_proba = svm_model.predict_proba(X_test_scaled)
cnn_preds_proba = cnn_model.predict(x_test)
mlp_preds_proba = cnn_model.predict(x_test)


# Apply weights to the predicted probabilities
weighted_xgb_proba = xgb_preds_proba * weights['xgb']
weighted_rf_proba = rf_preds_proba * weights['rf']
weighted_svm_proba = svm_preds_proba * weights['svm']
weighted_cnn_proba = cnn_preds_proba * weights['cnn']
weighted_mlp_proba = mlp_preds_proba * weights['cnn']


# Combine the weighted probabilities
weighted_avg_proba = weighted_xgb_proba +  weighted_cnn_proba + weighted_svm_proba + weighted_rf_proba

# Convert weighted probabilities to class predictions
weighted_ensemble_preds = np.argmax(weighted_avg_proba, axis=1)

# Evaluate the weighted ensemble performance
weighted_ensemble_accuracy = accuracy_score(y_test, weighted_ensemble_preds)
print(f"Weighted Ensemble Accuracy: {weighted_ensemble_accuracy:.3f}")
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
Weighted Ensemble Accuracy: 0.936
```

In [21]:
```
y_test[1]
```
Out[21]: np.int64(5)

In [22]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import numpy as np

# Step 1: Get predictions from base models
xgb_train_preds = xgb_model.predict_proba(x_train)
rf_train_preds = rf_model.predict_proba(x_train)
svm_train_preds = svm_model.predict_proba(X_train_scaled)
cnn_train_preds = cnn_model.predict(x_train)
mlp_train_preds = mlp_model.predict(x_train)

xgb_test_preds = xgb_model.predict_proba(x_test)
rf_test_preds = rf_model.predict_proba(x_test)
svm_test_preds = svm_model.predict_proba(X_test_scaled)
cnn_test_preds = cnn_model.predict(x_test)
mlp_test_preds = mlp_model.predict(x_test)

# Step 2: Create new dataset for meta-model training
stacked_train = np.hstack((xgb_train_preds, svm_train_preds, cnn_train_preds,  rf_train_preds + mlp_train_preds
stacked_test = np.hstack((xgb_test_preds, svm_test_preds, cnn_test_preds, rf_test_preds + mlp_test_preds))

# Step 3: Train meta-model
meta_model = LogisticRegression()
meta_model.fit(stacked_train, y_train)

# Step 4: Predict with the meta-model
ensemble_preds = meta_model.predict(stacked_test)

# Step 5: Evaluate performance
stack_ensemble_accuracy = accuracy_score(y_test, ensemble_preds)
print(f"Stacking Ensemble Accuracy: {stack_ensemble_accuracy:.3f}")
```
```
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step
250/250 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Stacking Ensemble Accuracy: 0.943
```

In [23]:
```
first = stacked_test[0]

print("Model 1:", first[:10])
print("Model 2:", first[10:20])
print("Model 3:", first[20:30])
print("Model 4:", first[30:40])
```
```
Model 1: [1.01780698e-01 1.04402355e-03 5.79456007e-03 2.22072396e-02
 5.31825006e-01 8.21004040e-04 7.74849718e-03 7.10365275e-05
 2.65636802e-01 6.30711615e-02]
Model 2: [2.55820816e-02 5.60426753e-05 1.16278271e-02 9.22202272e-04
 3.66323840e-02 1.35783528e-04 2.94415968e-04 4.82770885e-05
 9.12513156e-01 1.21878300e-02]
Model 3: [1.90308569e-09 1.41559058e-18 1.81511802e-11 1.75559544e-05
 9.55622613e-01 7.70977859e-14 5.56924437e-11 8.08517858e-13
 4.43598963e-02 1.24103154e-08]
Model 4: [1.01473531e-01 1.01106830e-02 1.40563264e-01 1.12757944e-01
 1.20096859e+00 1.00511976e-02 3.01493700e-02 5.94143581e-04
 3.00521762e-01 9.28095793e-02]
```

In [24]:
```
stacked_test.shape
```
Out[24]: (1998, 40)

## Print all accuracies

In [25]:
```
print(f"Accuracy (CNN): {accuracy_cnn:.3f}")
print(f"Accuracy (DNN): {accuracy_mlp:.3f}")
print(f"Accuracy (SVM): {accuracy_svm:.3f}")
print(f"Accuracy (RFC): {accuracy_rf:.3f}")
print(f"Accuracy (XGB): {xgb_accuracy:.3f}")

print()

print(f"Weighted Ensemble Accuracy: {weighted_ensemble_accuracy:.3f}")
print(f"Hard Voting Ensemble Accuracy: {hard_ensemble_accuracy:.3f}")
print(f"Soft Voting Accuracy: {soft_ensemble_accuracy:.3f}")
print(f"Stacking Ensemble Accuracy: {stack_ensemble_accuracy:.3f}")
```

```
Accuracy (CNN): 0.921
Accuracy (DNN): 0.915
Accuracy (SVM): 0.859
Accuracy (RFC): 0.874
Accuracy (XGB): 0.919

Weighted Ensemble Accuracy: 0.936
Hard Voting Ensemble Accuracy: 0.920
Soft Voting Accuracy: 0.938
Stacking Ensemble Accuracy: 0.943
```

## Creating a confusion matrix for the ensemble

In [26]:

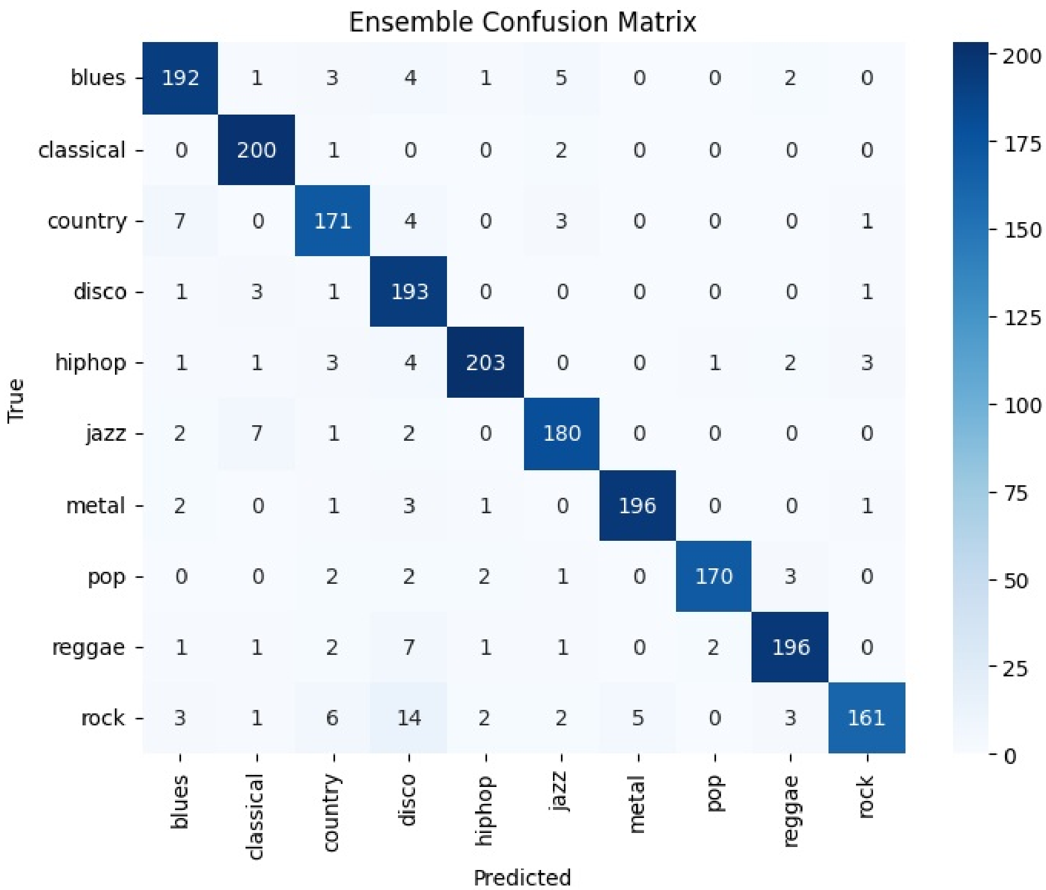
```python
# Get predictions from each model (XGBoost, SVM, CNN)
xgb_preds = xgb_model.predict(x_test)
svm_preds = svm_model.predict(x_test)
rf_preds = rf_model.predict(x_test)
cnn_preds = np.argmax(cnn_model.predict(x_test), axis=1)

# Soft Voting: average the predicted probabilities
xgb_probs = xgb_model.predict_proba(x_test)
svm_probs = svm_model.predict_proba(x_test)
rf_probs = rf_model.predict_proba(x_test)
cnn_probs = cnn_model.predict(x_test)

# Average the probabilities (soft voting)
ensemble_probs = (xgb_probs + svm_probs + cnn_probs + rf_probs) / 4
ensemble_preds = np.argmax(ensemble_probs, axis=1)
ensemble_cm = confusion_matrix(y_test, ensemble_preds)

# Plot the confusion matrix for the ensemble model
plt.figure(figsize=(8, 6))
sns.heatmap(ensemble_cm, annot=True, fmt="d", cmap="Blues", xticklabels=genres, yticklabels=genres)
plt.title("Ensemble Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

```
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
```




## Creating a confusion matrix for SVC

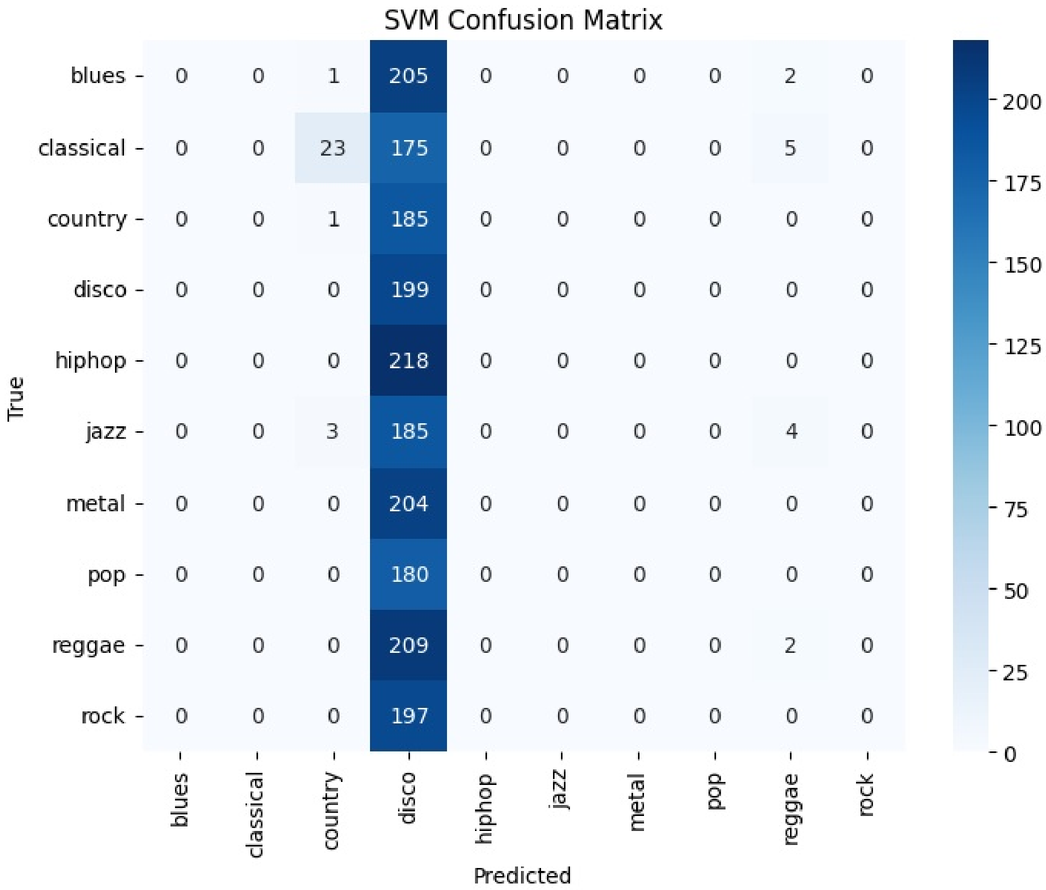
In [27]:

```python
svm_preds = svm_model.predict(x_test)
svm_cm = confusion_matrix(y_test, svm_preds)

# Plot the confusion matrix
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(svm_cm, annot=True, fmt="d", cmap="Blues", xticklabels=genres, yticklabels=genres)
plt.title("SVM Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
#plt.show()
```

Out[27]: Text(70.72222222222221, 0.5, 'True')

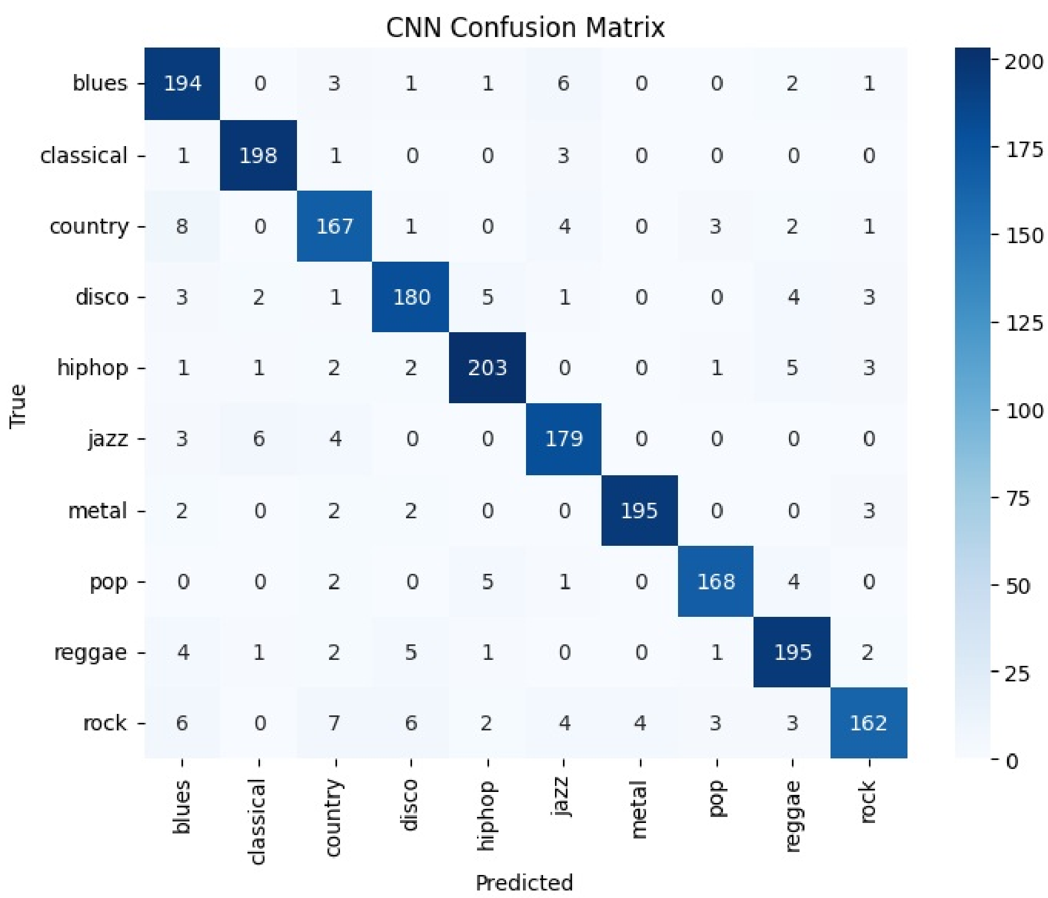



## Creating a confusion matrix for CNN

In [28]:
```
plt.clf
cnn_preds = np.argmax(cnn_model.predict(x_test), axis=1)
cnn_cm = confusion_matrix(y_test, cnn_preds)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cnn_cm, annot=True, fmt="d", cmap="Blues", xticklabels=genres, yticklabels=genres)
plt.title("CNN Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

**63/63** ━━━━━━━━━━━━━━━━━━━━ **0s** 4ms/step

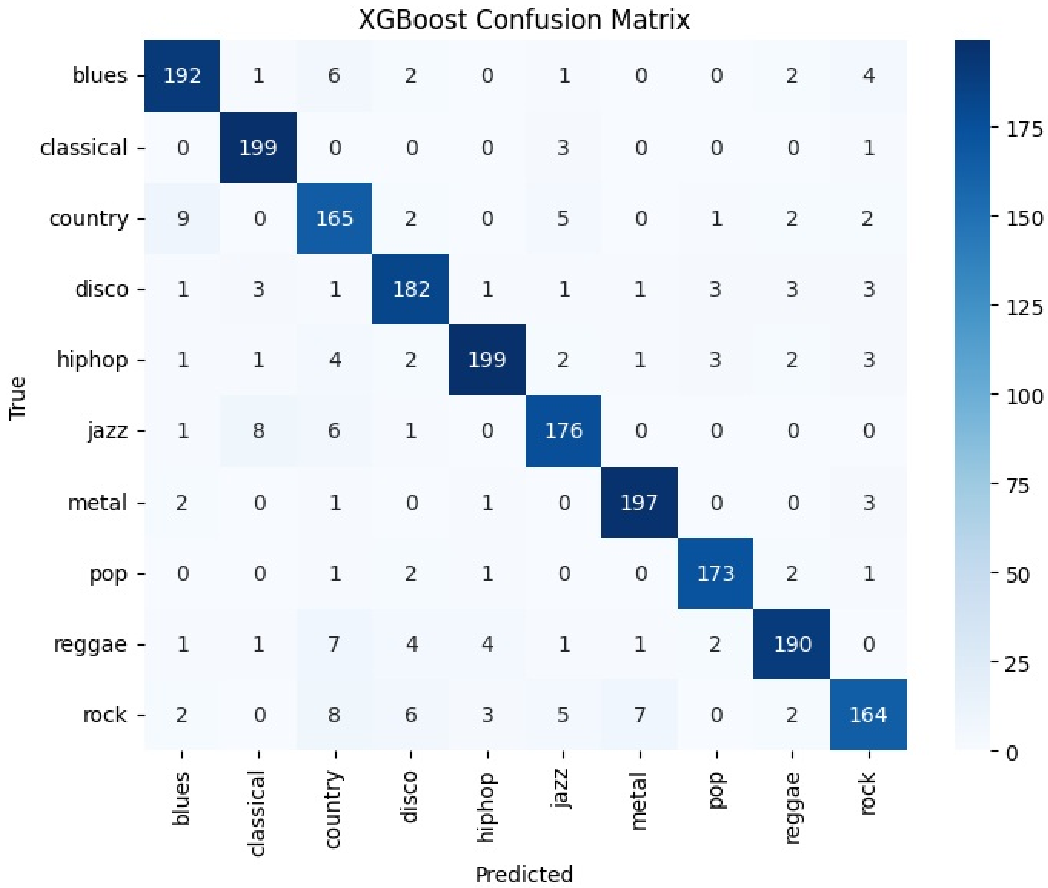



## Creating a confusion matrix for XGB

In [29]:

```
plt.clf
xgb_preds = xgb_model.predict(x_test)
xgb_cm = confusion_matrix(y_test, xgb_preds)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(xgb_cm, annot=True, fmt="d", cmap="Blues", xticklabels=genres, yticklabels=genres)
plt.title("XGBoost Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```


XGBoost Confusion Matrix
True
Predicted
blues
classical
country
disco
hiphop
jazz
metal
pop
reggae
rock
192 1 6 2 0 1 0 0 2 4
0 199 0 0 0 3 0 0 0 1
9 0 165 2 0 5 0 1 2 2
1 3 1 182 1 1 1 3 3 3
1 1 4 2 199 2 1 3 2 3
1 8 6 1 0 176 0 0 0 0
2 0 1 0 1 0 197 0 0 3
0 0 1 2 1 0 0 173 2 1
1 1 7 4 4 1 1 2 190 0
2 0 8 6 3 5 7 0 2 164
175
150
125
100
75
50
25
0