2EC402

Digital Signal Processing

"Musical Instruments Detection"

Performed By :

Het Shah (22BEC049)

Jaydeep Solanki (22BEC059)

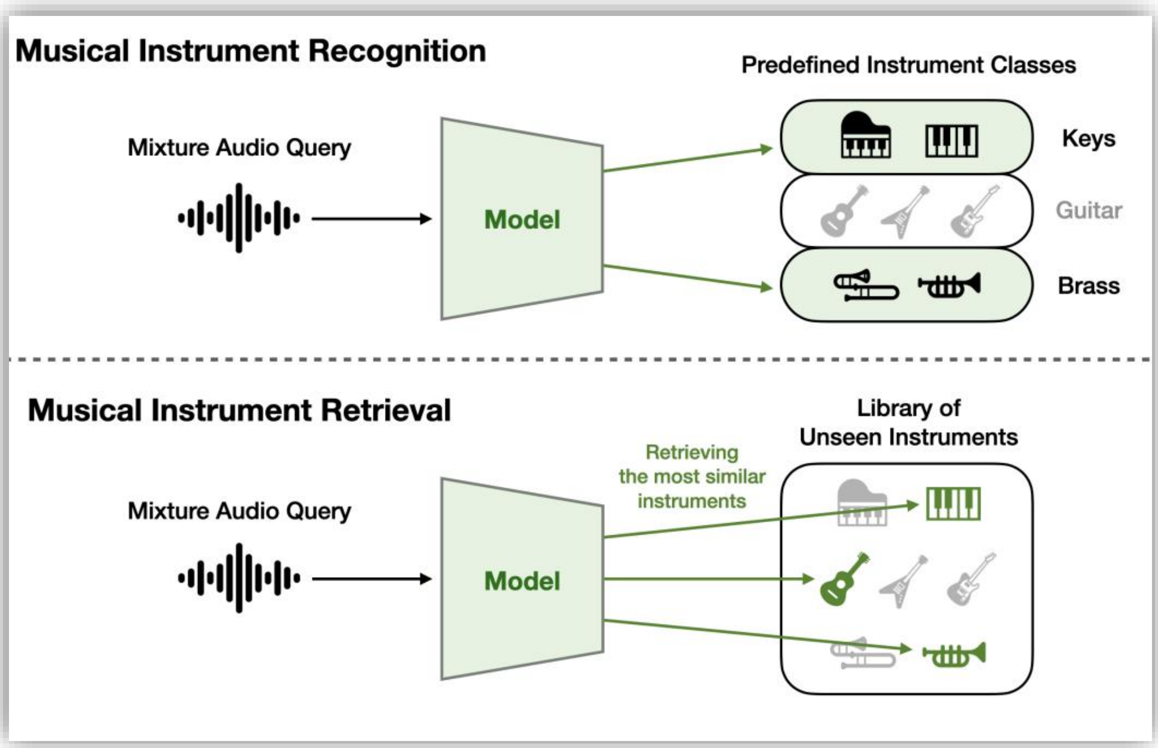


Table of Contents

Introduction

Ours is a project that looks at the landscape of processing audio signals to the dynamics of identifying and classifying musical instruments. Using the machine learning pattern recognition paradigm, we hope to get a working and efficient system that can autonomously detect and classify a wide range of musical instruments from audio recordings. The efforts will be focused on Python programming and advanced machine learning techniques toward building a model for exact instrument identification and classification.

Our work is driven by the desire to ease automated procedures for music analysis, improve the performance of music recommendation systems, and offer user experiences in multimedia applications on a new level. We find the potential for new dimensions of perception in the detailed complexity of musical expression through the synergy of knowledge in signal processing with the magic of machine learning.

Problem Description

This kind of thing is easy for a human but hard to automate: the classification of musical instruments according to the music being played. Real-world music is very complex. What is more, it may be polyphonic. The extraction of useful information from audio signals is also demanding, and subtleties like tone, quality, and playing style make the identification a yet more difficult task. Since the sound of musical instruments is multimodal complex, it requires a lot of computational resources to recognize them. Therefore, the recognition of instruments is computationally expensive.

There is a need for a high-performance computing system to go through the voluminous involved data, and hence, optimized results are obtained. Such systems will have the power to change the music industry, as they will finally be able to identify an instrument by its characteristics correctly.

Algorithm

1. Start

2. Import necessary libraries for data processing and machine learning.

3. Set a random seed for reproducibility.

4. Define the path to the dataset containing audio files.

5. Retrieve audio files from the specified path.

6. Define classes of musical instruments and their corresponding colors.

7. Assign labels and colors to each audio file based on their class.

8. Convert categorical class labels into numerical format using label encoding.

9. Define constants such as sampling frequency, number of Mel frequency bands, and number of MFCC coefficients.

10. Extract Mel-frequency cepstral coefficients (MFCCs) from each audio file.

11. Standardize feature vectors to have zero mean and unit variance.

12. Split the dataset into training and testing sets.

13. Instantiate a K-Nearest Neighbors (KNN) classifier.

14. Train the classifier in a separate thread to improve efficiency.

15. Predict labels for the test set using the trained classifier.

16. Calculate evaluation metrics such as recall, precision, F1-score, and accuracy.

17. Print the evaluation metrics to the console.

18. Generate a confusion matrix to visualize the classifier's performance.

19. Plot the confusion matrix for visualization.

20. Save the trained model, scaler, and label encoder to disk.

21. Load the saved model, scaler, and label encoder from disk.

22. Extract features from a new audio file.

23. Scale the extracted features using the loaded scaler.

24. Predict the class of the new audio file using the loaded classifier.

25. Decode the numerical class prediction back to its original class name using the label encoder.

26. Print the predicted class for the new audio file.

27. End

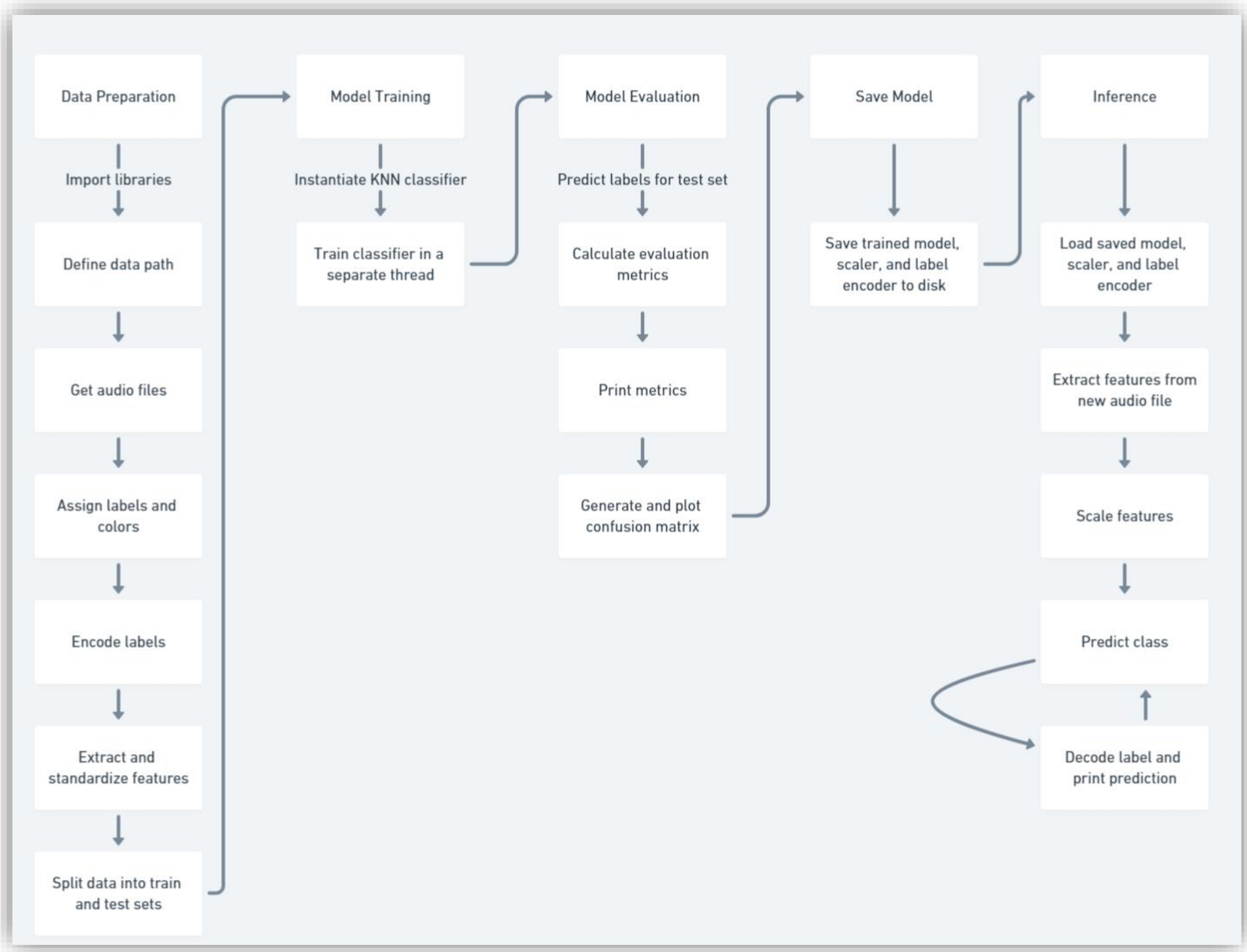
Working

- Data Preparation:
    - Importing of Libraries: Import libraries including fnmatch, numpy, itertools, librosa, matplotlib, seaborn, sklearn, threading, and joblib.
    - Set Random Seed: Set a random seed value for reproducibility of results.
    - Define Data Path: Define the path to the dataset.
    - get_audio_files(): Retrieve audio files from a directory based on a provided extension using fnmatch.
    - Define Class and Color Dictionary: Define classes along with their corresponding colors for labeling and visualization.
    - Get Labels and Colors: Obtain class labels and corresponding colors for each audio file.
    - Encoding Labels: Encode class labels using LabelEncoder to convert them into numerical values.
    - Constants: Define constants such as sampling frequency (fs), number of bands in Mel frequency bands (n_mels), and number of MFCC coefficients (n_mfcc).
    - Feature Extraction: Extract MFCCs of each audio file using Librosa.
    - Standardize Features: Standardize feature vectors to have zero mean and unit variance using StandardScaler.
    - Split Dataset: Split the dataset into training and test datasets using train_test_split.
- Model Training:
    - Instantiate Classifier: Instantiate a K-Nearest Neighbors classifier.
    - Train Classifier on Thread: Train the classifier in another thread.
- Model Evaluation:
    - Predicting Labels: Use the trained classifier to predict labels for the test set.
    - Calculate Metrics: Calculate evaluation metrics such as recall, precision, f1-score, and accuracy using functions from sklearn.metrics.
    - Print Metrics: Print evaluation metrics to the console.
    - Create Confusion Matrix: Create a confusion matrix using confusion_matrix function.
    - Plot Function Confusion Matrix: Define a function called plot_confusion_matrix() to output the confusion matrix.
- Save model:

- Save Model to Disk: The trained model with attached scaler and label encoder is saved using joblib.

- Inference
  - Load Model from Disk: Load a saved model from disk along with scaler and label encoder.
  - Feature Extraction from New Audio File: Extract features from a new audio file using the extract_features() function.
  - Scale Features: Scale features according to the loaded scaler.
  - Predict with KNN: Classify the new audio file using K-Nearest Neighbors.
  - Decoding Label: Decode the numerical label back into its original class name with the label encoder.
  - print_prediction: Print the predicted class for the new audio file to the console.
- Summary: The code performs data preparation, model training, evaluation, model saving, and inference. It applies the KNN classifier with MFCC features extracted from audio files to classify musical instruments. The trained model is saved for further predictions on new audio files.

Flowchart



Appendix 1 (Code)

```
import fnmatch
import numpy as np
import itertools
import librosa
import librosa.display
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import recall_score, precision_score, accuracy_score,
f1_score, confusion_matrix, classification_report
import threading
import joblib

np.random.seed(1)

data_path = './dataset'

def get_audio_files(path, extension='*.mp3'):
    files = []
    for root, _, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, extension):
            files.append(os.path.join(root, filename))
    return files

audio_files = get_audio_files(data_path)

classes = ['flute', 'sax', 'oboe', 'cello', 'trumpet', 'viola']
color_dict = {'cello': 'blue', 'flute': 'red', 'oboe': 'green', 'trumpet':
'black', 'sax': 'magenta', 'viola': 'yellow'}

def get_labels_and_colors(files, classes, color_dict):
    labels = []
```

```
    colors = []
    for file in files:
        for cls in classes:
            if cls in file:
                labels.append(cls)
                colors.append(color_dict[cls])
                break
        else:
            labels.append('other')
            colors.append('gray')
    return labels, colors

labels, colors = get_labels_and_colors(audio_files, classes, color_dict)

label_encoder = LabelEncoder()
classes_num = label_encoder.fit_transform(labels)

fs = 44100
n_mels = 128
n_mfcc = 13

def extract_features(file, fs=44100, n_mels=128, n_mfcc=13):
    y, sr = librosa.load(file, sr=fs)
    y /= np.max(np.abs(y))
    S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=n_mels)
    mfcc = librosa.feature.mfcc(S=librosa.power_to_db(S), n_mfcc=n_mfcc)
    return np.mean(mfcc, axis=1)

feature_vectors = [extract_features(file) for file in audio_files]

scaler = StandardScaler()
scaled_feature_vectors = scaler.fit_transform(feature_vectors)

X_train, X_test, y_train, y_test = train_test_split(scaled_feature_vectors,
classes_num, test_size=0.25, random_state=0, stratify=classes_num)
```

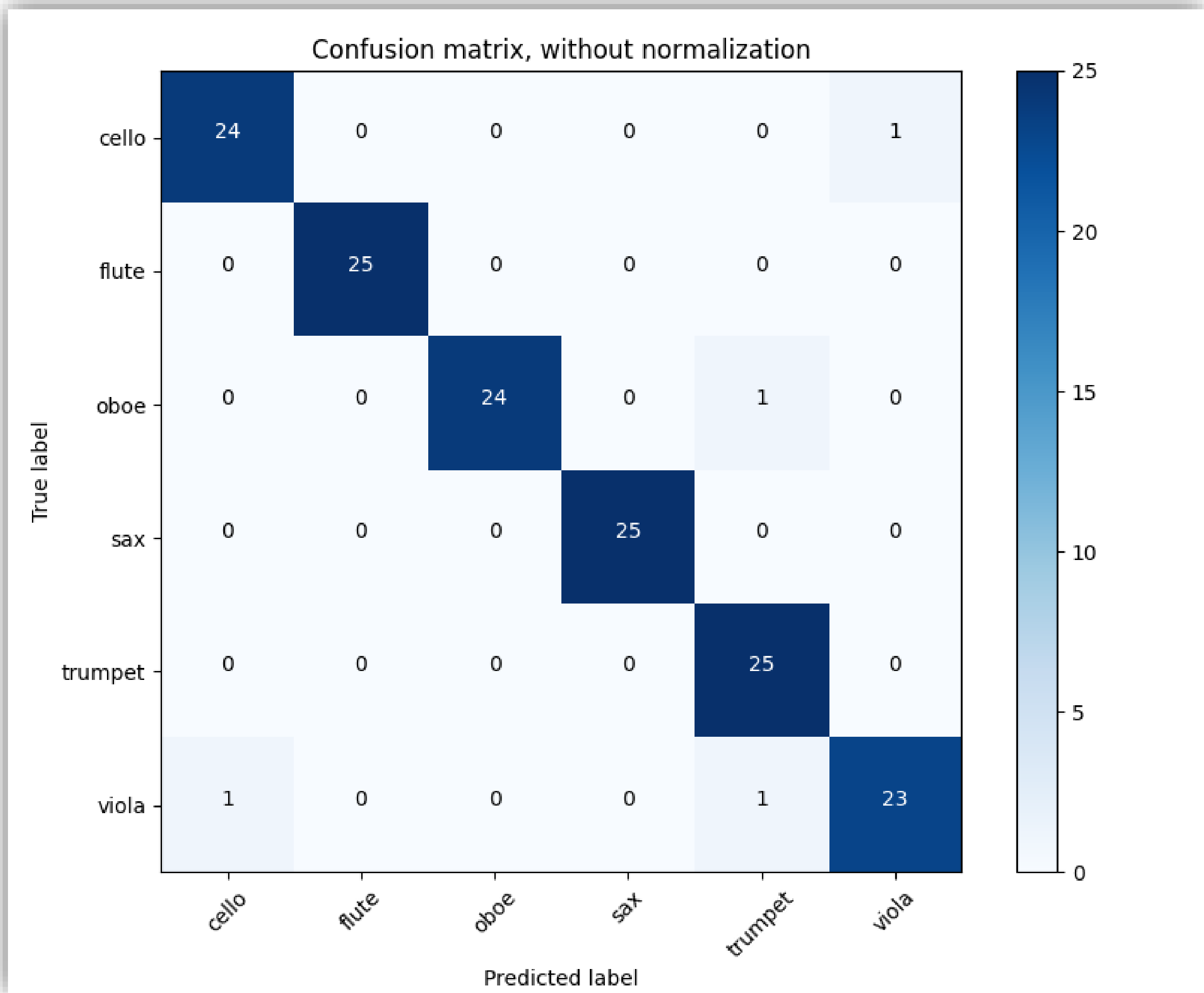
```
k_neighbors = 1
knn_classifier = KNeighborsClassifier(n_neighbors=k_neighbors)


def train_classifier(classifier, X_train, y_train):
    classifier.fit(X_train, y_train)


thread = threading.Thread(target=train_classifier, args=(knn_classifier,
X_train, y_train))
thread.start()
thread.join()


model_dict = {'model': knn_classifier, 'scaler': scaler, 'label_encoder':
label_encoder}
joblib.dump(model_dict, 'music.joblib')


predicted_labels = knn_classifier.predict(X_test)
print(predicted_labels)


recall = recall_score(y_test, predicted_labels, average=None)
precision = precision_score(y_test, predicted_labels, average=None)
f1 = f1_score(y_test, predicted_labels, average=None)
accuracy = accuracy_score(y_test, predicted_labels)


print("Recall:", recall)
print("Precision:", precision)
print("F1-Score:", f1)
print("Accuracy:", accuracy)


class_names = label_encoder.classes_
print(classification_report(y_test, predicted_labels, target_names=class_names))


def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion
matrix', cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
```

```
        title = 'Normalized confusion matrix'
    else:
        title = 'Confusion matrix, without normalization'

    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center",
color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

cnf_matrix = confusion_matrix(y_test, predicted_labels)
np.set_printoptions(precision=2)

plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix,
without normalization')

plt.show()
```

Appendix 2 (Code)

```
import os
import numpy as np
import librosa
import joblib

np.random.seed(1)

fs = 44100
n_mels = 128
n_mfcc = 13

def extract_features(file, fs=44100, n_mels=128, n_mfcc=13):
    y, sr = librosa.load(file, sr=fs)
    y /= np.max(np.abs(y))
    S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=n_mels)
    mfcc = librosa.feature.mfcc(S=librosa.power_to_db(S), n_mfcc=n_mfcc)
    return np.mean(mfcc, axis=1)

model_dict = joblib.load('music.joblib')
knn_classifier = model_dict['model']
scaler = model_dict['scaler']
label_encoder = model_dict['label_encoder']

new_audio_file = r"C:\Users\JaySs\OneDrive\Desktop\New folder (2)\flute.wav"
new_feature_vector = extract_features(new_audio_file)
scaled_new_feature_vector = scaler.transform([new_feature_vector])
predicted_label_num = knn_classifier.predict(scaled_new_feature_vector)
predicted_label_name = label_encoder.inverse_transform(predicted_label_num)[0]

print(f"Predicted class for {os.path.basename(new_audio_file)}:
{predicted_label_name}")
```

Results




```
Recall: [0.96 1.   0.96 1.   1.   0.92]
Precision: [0.96       1.         1.         1.         0.92592593 0.95833333]
F1-Score: [0.96       1.         0.97959184 1.         0.96153846 0.93877551]
Accuracy: 0.9733333333333334
              precision    recall  f1-score   support

       cello       0.96      0.96      0.96        25
       flute       1.00      1.00      1.00        25
        oboe       1.00      0.96      0.98        25
         sax       1.00      1.00      1.00        25
     trumpet       0.93      1.00      0.96        25
       viola       0.96      0.92      0.94        25

    accuracy                           0.97       150
   macro avg       0.97      0.97      0.97       150
weighted avg       0.97      0.97      0.97       150
```

- Tested an audio file which gave correct output.

```
PS C:\Users\JaySs\OneDrive\Desktop\Sem 4\5. DSP\Special Assignment> & "C:/Program Files/Python311/python.exe" "c:/Users/JaySs/OneDrive/Desktop/Sem 4/5. DSP/Special Assignment/predict.py"
Predicted class for trumpet.wav: trumpet
PS C:\Users\JaySs\OneDrive\Desktop\Sem 4\5. DSP\Special Assignment>
```

Observation

Generally, the model works quite well, but it makes a few errors in identifying some of the instruments in the audio files, given that the frequency of the instruments used is very close. On further observation, the rise of such mispredictions was noticed in cases where the frequencies of various instruments overlapped, and therefore, the model was confused in the identification process. Such an issue cannot be accepted for instrumentality correctness in the audio files, so it should be mitigated after further investigation.

Conclusion

In the developed k-Nearest Neighbors (kNN) music genre classification system, a holistic approach is used in the process of implementing automation to the process of identification of the musical instruments involved in an audio file. However, the system has a drawback: close-related instruments are hardly appropriately distinguished if all their frequencies overlap. It introduces erroneous instrument identification that will decrease the overall correctness rate.

Further exploration can be carried out for the types of particular scenarios in which the mispredictions are happening and ways to mitigate them. Possible approaches include:

- Feature Engineering: Exploring alternative features beyond MFCCs that may be employed to represent single instruments more effectively.
- Model Selection: The use of other machine-learning models to address overlapping frequencies in a better manner should be considered.
- Data augmentation: Increase the diversity and size of the data to be used for training, specifically if an instrument mixture has signals with overlapping frequencies to improve the model's generalization performance.
- Post-Processing Techniques: Use some forms of ensembling or smoothing as post-processing techniques in order to bring more accuracy in predictions and reduce the misclassification error rate.

The challenge is to address these and tweak the architecture, which would make the instrument identification more accurate and more reliable in its precision but would also lead to usefulness in many more applications of audio analysis.