POLITECNICO
MILANO 1863
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Music Instrument Classification

Research Project – Selected Topics in Music and Acoustic Engineering

Andrea Crisafulli, Giacomo De Toni, Marco Porcella, Gianluigi Vecchini

Person IDs: 11096492, 10978662, 11094994, 10743937

Student IDs: 270163, 273703, 273521, 273131

Advisor: Prof. Carabias Orti Julio Jose

Abstract

This research project, developed as part of the "Selected Topics in Music and Acoustic Engineering" course, addresses the task of automatic multilabel musical instrument classification using deep learning techniques. The aim is to build and evaluate a system capable of identifying multiple instruments simultaneously from real-world music recordings, leveraging the MedleyDB dataset, which contains professionally produced multitrack stems and annotations. The project implements a full audio processing pipeline including dataset construction, time–frequency feature extraction (via mel-spectrograms), convolutional neural network training, and quantitative performance analysis. Particular attention is devoted to handling overlapping sources, designing macro-categories for improved generalization, and balancing the dataset through data augmentation. Despite hardware limitations, the system achieves robust classification results and provides insights into the role of timbre in complex musical textures. The work contributes to the broader field of Music Information Retrieval (MIR), demonstrating the practical effectiveness of deep learning methods for instrument recognition in polyphonic audio.

Summary

Figures List

Tabs List

Cells List

1

Project motivations

The main goal of this project is to develop a system for automatic music instrument classification in polyphonic music recordings. This task is a challenging but important problem in the field of music information retrieval that can also be helpful in other classification problems, such as musical genre classification and music transcription. Automatic recognition of musical instruments is a central topic in the field of Music Information Retrieval (MIR), with applications ranging from automatic tagging and music search based on timbre, to audio analysis and transcription. However, identifying instruments in real-world polyphonic contexts remains a challenging task due to the spectral overlap between sources and the timbral variability caused by different styles, performances, and production conditions. In this project, carried out within the course *Selected Topics in Music and Audio Engineering*, we aim to design and evaluate a system for automatic musical instrument classification from real audio signals, using techniques from machine learning and deep learning. The system was trained and evaluated using the MedleyDB dataset, which contains professionally recorded multitrack music, including both isolated instrument stems and complete stereo mixes. This structure allowed us to explore classification under two different audio conditions: monophonic sources (single instrument stems) and polyphonic textures (full mixes with multiple simultaneous sources). The project pipeline included several key stages: data parsing and selection, audio preprocessing, feature extraction, model training, performance evaluation and error analysis. Overall, the project aimed not only to implement an effective instrument classification system, but also to gain practical insights into the challenges of working with complex audio data. To facilitate reproducibility and further development, all code and experiments have been made publicly available through a GitHub repository, accessible at:

`https://github.com/GianVecchini/SelectedTopic`

2

Dataset Structure and Preparation

2.1 Overview

The dataset used in this project is MedleyDB (version 1.0), a multitrack audio collection designed for research in Music Information Retrieval (MIR). Each of the 196 tracks contains professionally produced audio recordings, including isolated instrument stems, stereo mixes and a range of melodic and pitch annotations. The goal of the dataset preparation pipeline was to construct a reliable and flexible input for multilabel instrument classification, starting from raw metadata and audio files. This involved filtering and parsing the data, remapping detailed instrument tags to broader macrocategories, selecting audio types and encoding labels in a machine-readable format.

2.2 Dataset Filtering and Configuration

Data extraction was handled through a configurable script that supports Windows and macOS systems. The main control flags allow the user to select which audio file types to include:

- `considerMixFiles` – include stereo mixes (polyphonic, multilabel);
- `considerStemFiles` – include isolated instrument stems (monophonic, single label);
- `considerRawFiles` – include raw multitrack files (monophonic, single label).

The dataset path is resolved based on the OS and audio files are accessed from the directory structure of MedleyDB. An excerpt of the setup logic is shown in code 2.1.

Dataset configuration and OS handling

[2.1]:
```
...
considerMixFiles = True
considerStemFiles = True
considerRawFiles = False

OSys = 0  # 0 = windows, 1 = macOS

if OSys == 0:
    basePath = Path("E:/MedleyDB")
elif OSys == 1:
    basePath = Path("/Volumes/Extreme SSD/MedleyDB")
else:
    errorPath = True
```

```
audioPath = basePath / "Audio"
pitchAnnotationsDir = basePath / "Annotations" /
 ↪  "Pitch_Annotations"
...
```

2.3 Label Simplification via Macrocategories

MedleyDB contains over 80 fine-grained instrument labels, including many stylistic or culturally specific variants. To reduce data sparsity and enhance model interpretability, these labels were manually mapped into 17 aggregated macrocategories, some example of them are reported in code 2.2.

Part of Macrocategories Definition

[2.2]:
```
...
# 1. Voice
'male singer': 'voice',
'female singer': 'voice',
...
# 2. Strings
'violin': 'strings',
'viola': 'strings',
'cello': 'strings',
'erhu': 'strings',
...
# 3. Acoustic Guitars
'acoustic guitar': 'acoustic guitars',
'lap steel guitar': 'acoustic guitars',
...

# 8. Woodwinds
'flute': 'woodwinds',
'flute section': 'woodwinds',
'piccolo': 'woodwinds',
'saxophone': 'woodwinds',
...

# 10. Piano
'piano': 'piano',
'tack piano': 'piano',
...

# 13. Drum Kit
'drum set': 'drums',
'snare drum': 'drums',
'kick drum': 'drums',
...
```

```
# 16. FX / Noise / Electronic Layers
'scratches': 'electronic fx',
'fx/processed sound': 'electronic fx',
...
```

2.4 Metadata Parsing and Label Mapping

Each song directory in MedleyDB contains a YAML file that provides detailed metadata including the list of stems, associated instrument labels, and optional pitch annotations. The parsing procedure begins by iterating through all song folders in the `Audio` directory. For each folder, the corresponding metadata file is loaded using the `PyYAML` library. From the parsed data, stems are extracted along with their original instrument labels. To reduce label fragmentation and improve the model's ability to generalize, each instrument is mapped to a broader macro-category using a predefined dictionary (`labelGroupsDict`). This mapping allows similar instruments (e.g., different types of saxophones or guitars) to be grouped under the same high-level label. Based on configuration flags, the script constructs data entries for stem files, mix files, or both. Each entry includes the song name, the label or multi-label (for mixes), and the file path. For mix files, the script aggregates all instruments present in the song into a single multi-label string. When pitch annotations are available, the script also extracts the time interval during which each stem is active. By averaging the activity centers of all stems in a song, a representative starting point is computed for the mix file. This starting point is used later to extract a 15-second audio segment that maximizes the presence of simultaneous instruments. Finally, all collected entries are stored in a list and converted to a `pandas` DataFrame. Labels are processed using `MultiLabelBinarizer` to obtain a binary representation suitable for multilabel classification. An excerpt from the metadata parsing logic is shown in code 2.3.

Parsing metadata and assigning labels

[2.3]:
```
...
for songDir in audioPath.iterdir():
    yamlFilePath = songDir / f"{songDir.name}_METADATA.yaml"
    with open(yamlFilePath, "r") as f:
        metadata = yaml.safe_load(f)

    stemsData = metadata.get("stems", {})
    for stemId, stem in stemsData.items():
        label = labelGroupsDict[stem.get("instrument")]

        if considerStemFiles:
            filePath = songDir / f"{songDir.name}_STEMS" /
             ↪  stem.get("filename")
            data.append({"song": songDir.name, "label": label,
             ↪  "filePath": filePath})

        if considerMixFiles:
            labelArray.append(label)
...
```

2.5 MultiLabel Binarization and DataFrame Construction

All parsed entries are stored in a Pandas `DataFrame`. For mix files, multiple instruments may be active simultaneously and their macrolabels are stored as pipe-separated strings and parsed into lists. To convert these lists into a format usable by neural networks, we applied a `MultiLabelBinarizer` from `scikit-learn`. This process transforms each sample into a binary vector of length equal to the number of classes, where each position indicates the presence or absence of a specific label.

Label binarization using MultiLabelBinarizer

[2.4]:
```
...
mlbAllDataset = MultiLabelBinarizer()
audioLabelsBinary = mlbAllDataset.fit_transform(df["labelList"])
audioLabelsBinary = np.asarray(audioLabelsBinary)
...
```

This encoded label matrix was used throughout the model training pipeline. A final consistency check ensured that the number of binary label rows matched the number of audio files parsed. The distribution of instrument labels in the final dataset is shown in figure 2.1. The plot confirms the natural imbalance in real-world music recordings.

2.6 Summary

The dataset preparation pipeline builds a robust foundation for the multilabel classification task. It combines file parsing, label remapping and multilabel encoding into a reproducible framework. The use of macrocategories reduces complexity, while metadata and annotations provide additional insight into audio segmentation. This preprocessed dataset served as the input for all feature extraction and model training procedures explained subsequently.

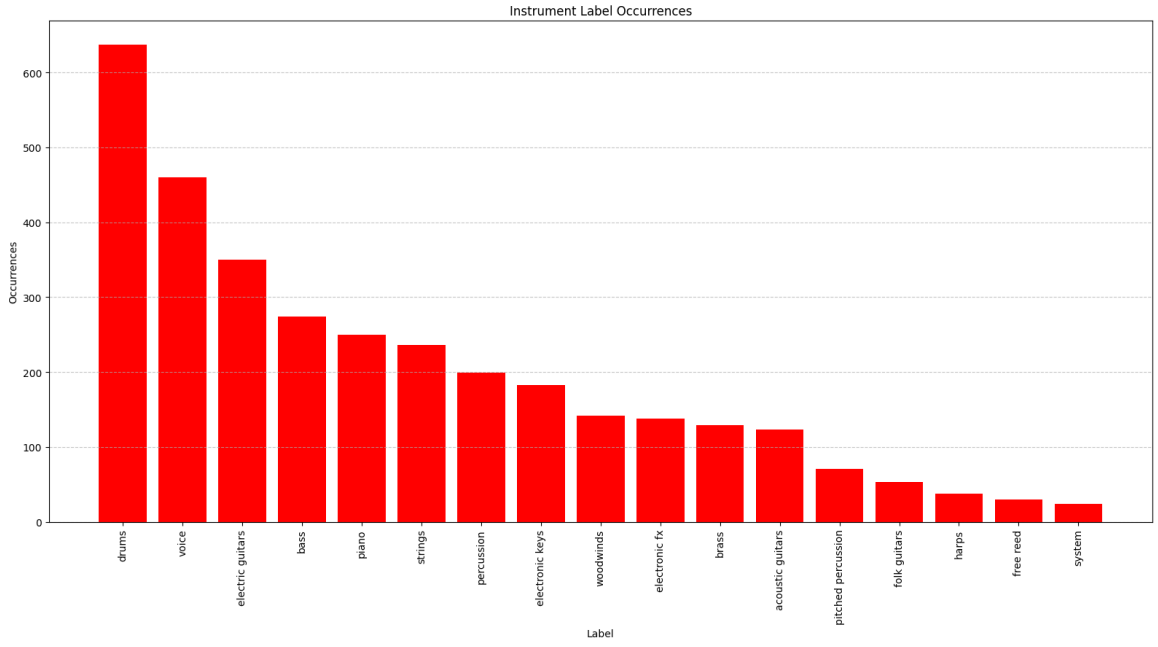

Figure 2.1: Instrument label distribution in the dataset

3

Audio Feature Extraction & Data Preparation

3.1 Audio Extraction and Segmentation

All audio signals were loaded using `librosa` at a fixed sampling rate of 22050 Hz and normalized to have unit peak amplitude. For each file, a segment of 15 seconds was extracted starting either from the average activation time estimated from pitch annotations (for mix files), or from the first index exceeding a minimum energy threshold (for stems). If the segment was shorter than 15 seconds, it was zero-padded. The audio loading and segmentation logic is illustrated in code 3.1.

Signal extraction and normalization

[3.1]:
```
...
signals = []
for x in tqdm(audioFilesToExtract, desc="Loading audio
 ↪  files..."):
    y, _ = librosa.load(x, sr=22050)
    if np.max(np.abs(y)) > 0:
        y = y / np.max(np.abs(y))

    if startingTimeDict.get(idx) is not None:
        startIndex = int(startingTimeDict[idx])
    elif np.any(y > minAmplitude):
        startIndex = np.argmax(y > minAmplitude)
    else:
        startIndex = 0

    endIndex = startIndex + numSamples
    if endIndex <= len(y):
        y = y[startIndex:endIndex]
    elif len(y) >= numSamples:
        y = y[-numSamples:]
    else:
        paddingNeeded = numSamples - len(y)
        y = np.pad(y[startIndex:], (0, paddingNeeded),
         ↪  'constant')
```

```
    signals.append(y.astype(np.float32))
    idx += 1
...
```

3.2 Mel-Spectrogram Computation

Each signal was converted into a mel-spectrogram using the `librosa` implementation. The transformation included a Short-Time Fourier Transform (STFT), followed by mapping to the mel scale and logarithmic compression to decibel values. The conversion process is shown in code 3.2. Some representative mel-spectrograms of stem and mix files are shown in figures 3.1.

Mel-spectrogram computation

[3.2]:
```
...
melSpegrams = []
for signal in tqdm(signals, desc="Processing audio signals..."):
    S = librosa.feature.melspectrogram(y=signal, sr=22050)
    SdB = librosa.power_to_db(S, ref=np.max)
    melSpegrams.append(SdB)
...
```

3.3 Dataset Splitting

To ensure class balance across splits in the multilabel setting, we applied the iterative stratification technique provided by `skmultilearn`. The original dataset was divided into 60% training, 24% validation and 16% test samples. The initial class distribution across splits is reported in figure 3.2.

Iterative stratified splitting

[3.3]:
```
...
indices = np.arange(len(melSpegrams)).reshape(-1, 1)
X_idx_train, y_train, X_idx_temp, y_temp =
 ↪  iterative_train_test_split(indices, labelsToLoad,
 ↪  test_size=0.40)
X_idx_val, y_val, X_idx_test, y_test =
 ↪  iterative_train_test_split(X_idx_temp, y_temp,
 ↪  test_size=0.40)
X_train = melSpegrams[X_idx_train.ravel()]
X_val = melSpegrams[X_idx_val.ravel()]
X_test = melSpegrams[X_idx_test.ravel()]

signals_train = [signals[i] for i in X_idx_train.ravel()]
signals_val = [signals[i] for i in X_idx_val.ravel()]
signals_test = [signals[i] for i in X_idx_test.ravel()]
...
```

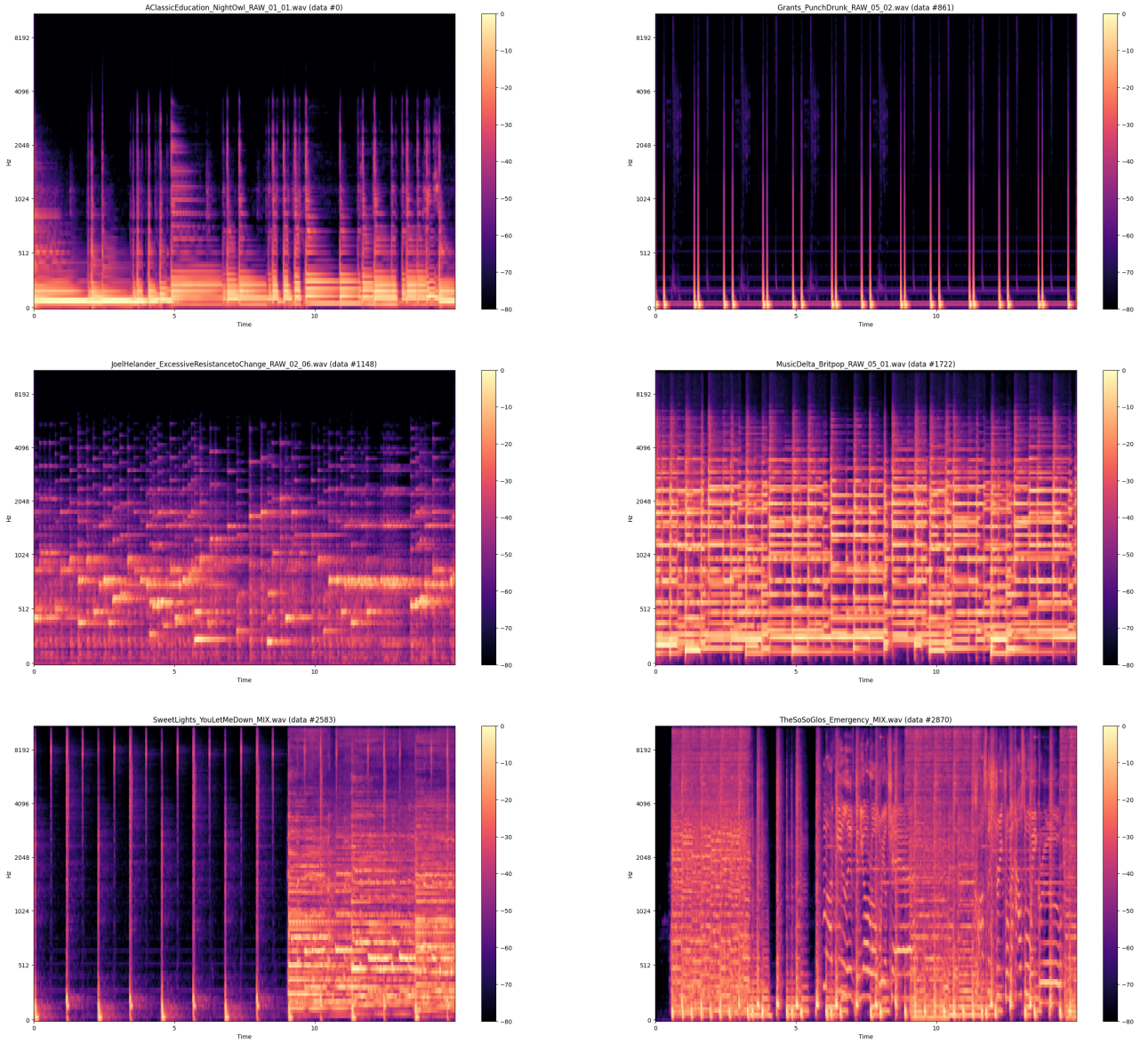

Figure 3.1: Examples of mel spectrograms

3.4 Data Augmentation and Balancing

Due to the natural imbalance in the dataset, a data augmentation strategy was employed to oversample underrepresented classes. For each class, samples containing only one active label were selected and synthetically augmented through:

- Time-stretching – simulates tempo variation by warping the spectrogram along the time axis;
- Pitch-shifting – simulates transposition by warping the spectrogram along the frequency axis.

The augmentation functions are shown in code 3.4 and the re-extraction strategy in code 3.5. The class distribution after augmentation is reported in figure 3.3.

Random augmentation of mel-spectrograms

[3.4]:
```
...
def augmenter(spegram):
```

```
...
    originalShape = spegram.shape
    modifications = []

    if random.random() < 0.75:
        rate = random.uniform(0.5, 1.5)
        stretched = scipy.ndimage.zoom(spegram, (1, rate),
         ↪  order=1)
        spegram = padding(stretched, originalShape)
        modifications.append(f"time_stretch({rate:.2f})")

    if random.random() < 0.75:
        semitones = random.uniform(-12, 12)
        freqFactor = 2 ** (semitones / 12)
        shifted = scipy.ndimage.zoom(spegram, (freqFactor, 1),
         ↪  order=1)
        spegram = padding(shifted, originalShape)
        modifications.append(f"pitch_shift({semitones:.2f})")

    return spegram, modifications
...
```

Re-extraction of audio segments

[3.5]:

```
...
def reExtractWithOffSet(filePath, duration=10, sr=22050,
 ↪  minAmp=minAmplitude):
    y, _ = librosa.load(filePath, sr=sr)
    if np.max(np.abs(y)) > 0:
        y = y / np.max(np.abs(y))

    highAmpIndices = np.where(np.abs(y) > minAmp)[0]
    maxStart = len(y) - duration * sr
    startSample = choice(highAmpIndices[highAmpIndices <=
     ↪  maxStart]) if len(highAmpIndices) else 0

    endSample = startSample + int(duration * sr)
    yAug = y[startSample:endSample]

    if len(yAug) < duration * sr:
        yAug = np.tile(yAug, (duration * sr) // len(yAug) +
         ↪  1)[:duration * sr]

    mel = librosa.feature.melspectrogram(y=yAug, sr=sr)
    return librosa.power_to_db(mel, ref=np.max)
...
```

3.5 Summary

This chapter described the audio processing pipeline from raw waveform loading to mel-spectrogram generation and dataset preparation. The use of pitch annotations and amplitude thresholds helped identify relevant segments, while iterative splitting ensured stratified coverage. Finally, data augmentation compensated for class imbalance and improved generalization for the deep learning model trained in the next stage.

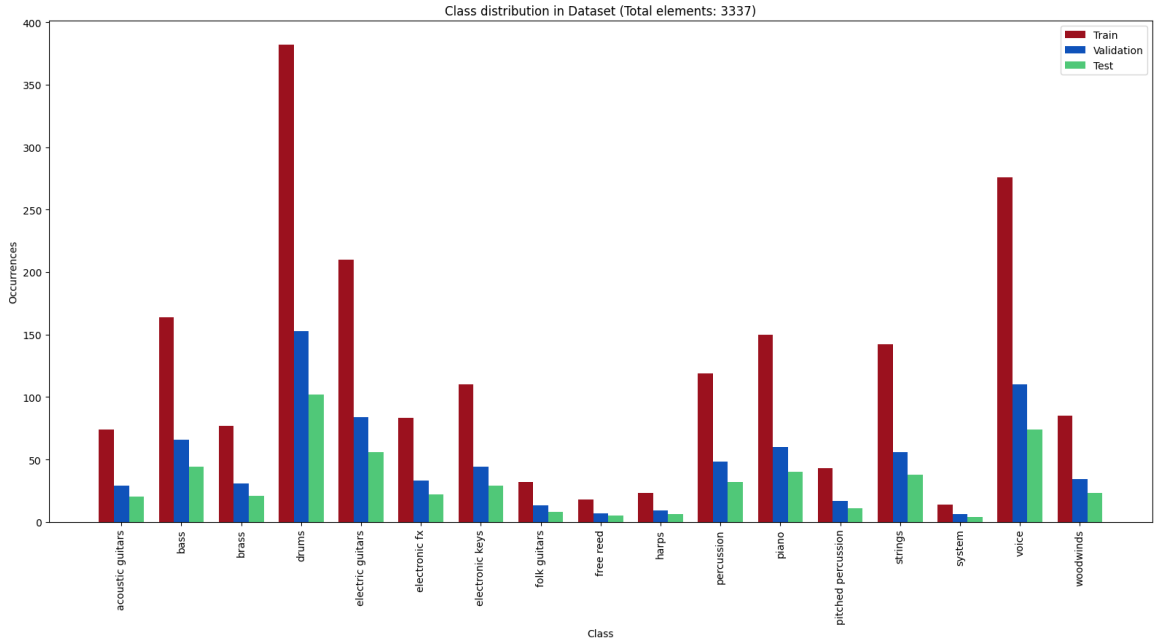

Figure 3.2: Class distribution across dataset splits (unbalanced)

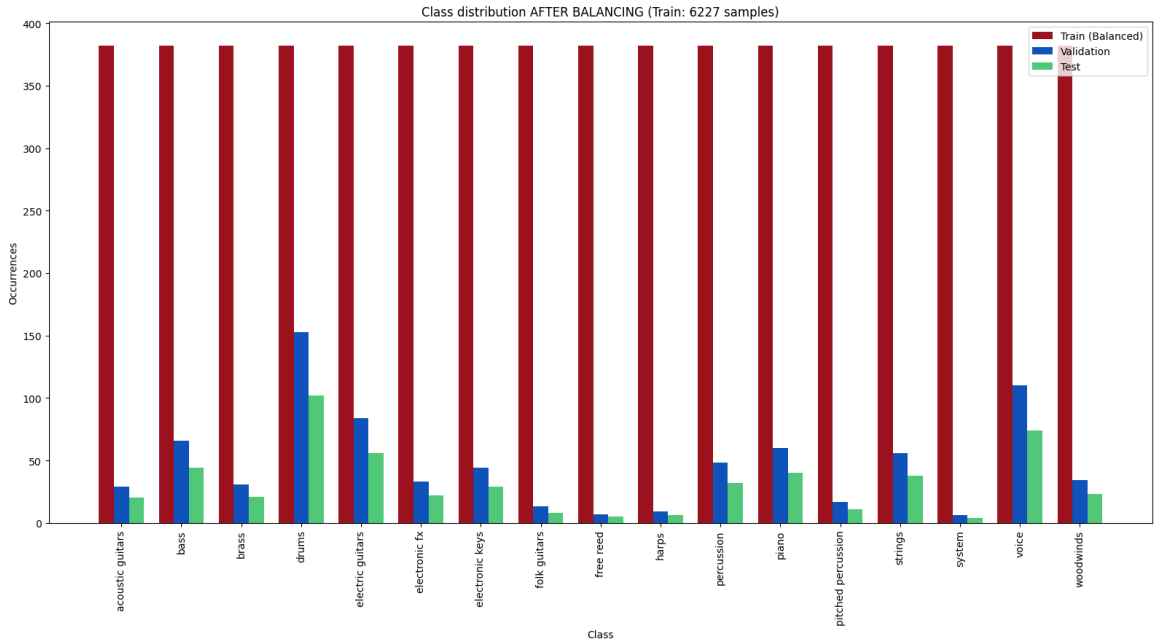

Figure 3.3: Class distribution after balancing via data augmentation

4

Model Architecture and Training Strategy

4.1 Convolutional Neural Network Design

To address the multilabel classification of musical instruments from mel-spectrogram inputs, we designed a deep Convolutional Neural Network (CNN) inspired by the work of Han et al. (2016). The input to the network is a tensor of shape $(128, T, 1)$, where 128 represents the number of mel-frequency bins and T the number of time frames, which depends on the length of the audio segment. The architecture comprises four convolutional blocks, each consisting of a Conv2D layer, a MaxPooling2D layer and dropout regularization. These are followed by a global average pooling layer and two dense layers. A final sigmoid activation enables multilabel classification. The architecture is detailed in table 4.1.

Layer (type)	Output Shape	Param #
Conv2D	(None, 128, 862, 32)	320
MaxPooling2D	(None, 43, 288, 32)	0
Dropout	(None, 43, 288, 32)	0
Conv2D	(None, 43, 288, 64)	18,496
MaxPooling2D	(None, 15, 96, 64)	0
Dropout	(None, 15, 96, 64)	0
Conv2D	(None, 15, 96, 128)	73,856
MaxPooling2D	(None, 5, 32, 128)	0
Dropout	(None, 5, 32, 128)	0
Conv2D	(None, 5, 32, 256)	295,168
Dropout	(None, 5, 32, 256)	0
GlobalAveragePooling2D	(None, 256)	0
Dense	(None, 512)	131,584
Dropout	(None, 512)	0
Dense	(None, 17)	8,721

Table 4.1: Summary of the CNN model architecture

4.2 Compilation and Optimization

The network was compiled using the `Adam` optimizer, with a learning rate manually tuned to 3×10^{-4} for improved convergence. The loss function adopted is `binary_crossentropy`,

which is appropriate for multilabel classification tasks. The model's performance during training was monitored using the following metrics:

- Binary Accuracy, which evaluates classification performance on thresholded predictions.
- AUC (Area Under the Curve), computed in a multilabel context, offering a ranking-based measure of prediction quality.

4.3 Callbacks and Regularization

To enhance generalization and prevent overfitting, the following callbacks (shown in code 4.1) were integrated into the training pipeline:

- `EarlyStopping` with a patience of 50 epochs based on validation loss.
- `ModelCheckpoint` saving the best model weights in `.keras` formats.
- `CSVLogger` to record epoch-wise training and validation metrics.

Callbacks definition

[4.1]:
```
...
csvLogger = CSVLogger(csvLogPath, append = True)

earlyStop = EarlyStopping(
    monitor = 'val_loss',
    patience = 50,
    restore_best_weights = True,
    verbose = 1
)

checkpointKeras = ModelCheckpoint(
    filepath = checkpointPathKeras,
    monitor = 'val_loss',
    save_best_only = True,
    verbose = 1
)

callbacks = [csvLogger, earlyStop, checkpointKeras]
...
```

4.4 Training Procedure

The model was trained for 300 (at max) epochs using a batch size of 32. The dataset had previously undergone balancing through oversampling and augmentation techniques, as discussed in chapter 3. Validation was performed at each epoch using a separate stratified validation set. The full training configuration is reported in code 4.2.

CNN training procedure

[4.2]:
```
...
opt = keras.optimizers.Adam(learning_rate = 0.0003)

modelCNN.compile(
```

```
    optimizer = opt,
    loss = 'binary_crossentropy',
    metrics = [
        BinaryAccuracy(name = 'binary_accuracy'),
        AUC(multi_label = True, name = 'auc')
    ]
)
batchSize = 32
epochs = 300
history = modelCNN.fit(
    X_train_balanced, y_train_balanced,
    validation_data = (X_val, y_val),
    batch_size = batchSize,
    epochs = epochs,
    verbose = 0,
    callbacks = callbacks
)
...
```

4.5 Results

Figure 4.1 shows the training history for both the binary accuracy and loss metrics on the training and validation sets. The model showed smooth convergence and robust generalization capabilities. The EarlyStopping mechanism prevented overfitting and the use of dropout and batch-level regularization was effective in maintaining performance stability. The CNN model demonstrates high stability and generalization across multilabel instrument classification. In the subsequent chapter, its performance on the test set will be analyzed more thoroughly, including per-class precision, recall and confusion matrix insights.

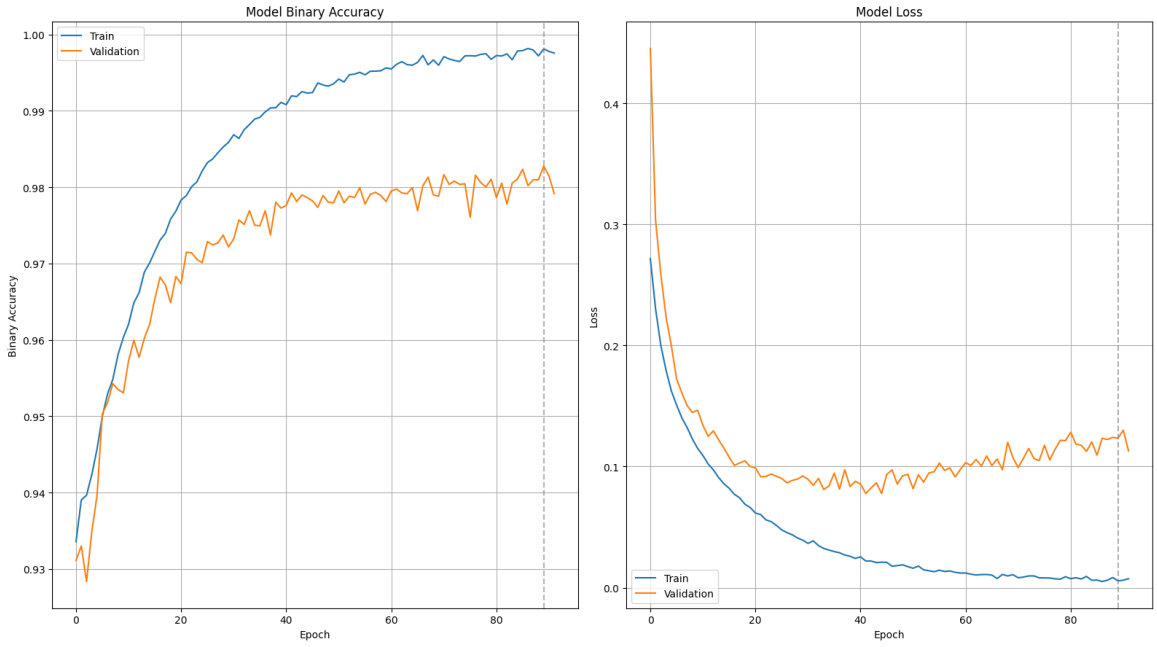

Figure 4.1: Training and validation metrics

5

Model Evaluation and Analysis

5.1 Quantitative Evaluation on Test Data

Finally, once the model has been trained on the train set and validated on the validation set, the model can be evaluated on the actual test set to see how it performs on unseen data. Before taking a look at the performance, in chapter 4 it's reported that the validation binary accuracy is $\simeq 0.02$ lower than the test's one, while the validation loss is $\simeq 0.1$ higher than the test loss. Indeed, the model doesn't perform an extreme overfitting; in this way, it has learned the general patterns of each mel spectrogram in a way that can be easily extended to unseen data. Before presenting the evaluation results, it is important to recall that the model is trained on multi-label targets using a `MultiLabelBinarizer`. This implies that each sample can belong to multiple classes simultaneously. Consequently, the model outputs, for each sample, a vector containing the probabilities of each label to be active.

5.2 General performance

The model has been evaluated on the test set using three key metrics: the binary cross-entropy loss, binary accuracy, and the Area Under the ROC Curve (AUC). The obtained results are:

- Loss (Binary Cross-Entropy): measures the average difference between the predicted probabilities and the actual labels, penalizing more when the probabilities are far off the correct predictions. A lower loss indicates that the model is producing probability distributions that closely match the true labels. The value obtained is 0.076, this ensures a good agreement between the predicted probabilities and the true multi-label targets.
- Binary Accuracy: in multi-label classification, it checks whether each label (0 or 1) is predicted correctly. It does not require that all labels for a sample be correct at once, making it a more lenient measure than strict accuracy. The value obtained is 97.97, this means that nearly all predicted labels match the ground truth.
- AUC (Area Under the Receiver Operating Characteristic Curve): evaluates the model's ability to discriminate between the presence and absence of each class, based on the predicted probabilities. A score of 94.38% indicates excellent separability between the positive and negative cases for each label.

Overall, the model demonstrates strong performance on the test set, with high binary accuracy and AUC, and low loss.

5.3 Confusion Matrix Analysis for unique instruments

It can now be taken a look at the performance of the model more in detail, considering only the predictions on audios with unique instruments. To do that, the probability of a label being active is imposed to be higher than 0.5; then such label is considered active, negative otherwise. For a more interpretable evaluation, a confusion matrix is built by focusing only on test samples with a single active label (excluding multi-instrument "MIX" cases). The code below summarizes this post-processing:

Confusion matrix computation

[5.1]:
```
...
y_pred_probs = modelCNN.predict(X_test)
y_pred_binary = (y_pred_probs > 0.5).astype(int)

# Extract labels for samples with only one active instrument
y_test_simplified, y_pred_simplified = [], []
for i, row in enumerate(y_test):
    gt = np.where(row == 1)[0]
    pred = np.where(y_pred_binary[i] == 1)[0]
    if len(gt) == 1:
        y_test_simplified.append(mlb.classes_[gt[0]])
        if len(pred) > 0:
            y_pred_simplified.append(mlb.classes_[pred[0]])

cm = confusion_matrix(y_test_simplified, y_pred_simplified,
 ↪  labels=mlb.classes_)
...
```

As it can be seen from figure 5.1, in the confusion matrix each row corresponds to the true labels, and each column corresponds to the predicted labels. The diagonal elements indicate the number of times the model correctly identified a given instrument class, while the off-diagonal elements show misclassifications, i.e., how often one class was confused for another. The model performs very well on several classes. For instance, `drums` (86), `voice` (58), and `electric guitars` (43) have high true positive counts, indicating strong recognition capability. However, the model is working worst on the labels with fewer audio samples, as `harps`, `woodwinds` and `free reeds`. As said before, the diagonal dominance indicates that most predictions are correct, although confusion still arises on instruments with fewer audio samples.

5.4 Performance on mix samples

On the mix samples it can be said that the performance struggles a bit more, considering that in some cases the labels assigned to some mix are not present in the interval sampled from the original audio. To take a look at the overall performance over the mix samples, it is suggested to take a look at the python notebook, in which each estimation is associated to the original labels, the audio sample, and the mel spectrogram. Furthermore, the notebook includes also a table containing all the True/False Positive/Negative estimations. As expected, the model struggles more on this type of task, but is still able to find an instrument when it is well hearable in the audio track and not hidden by the others. An excerpt of the function used

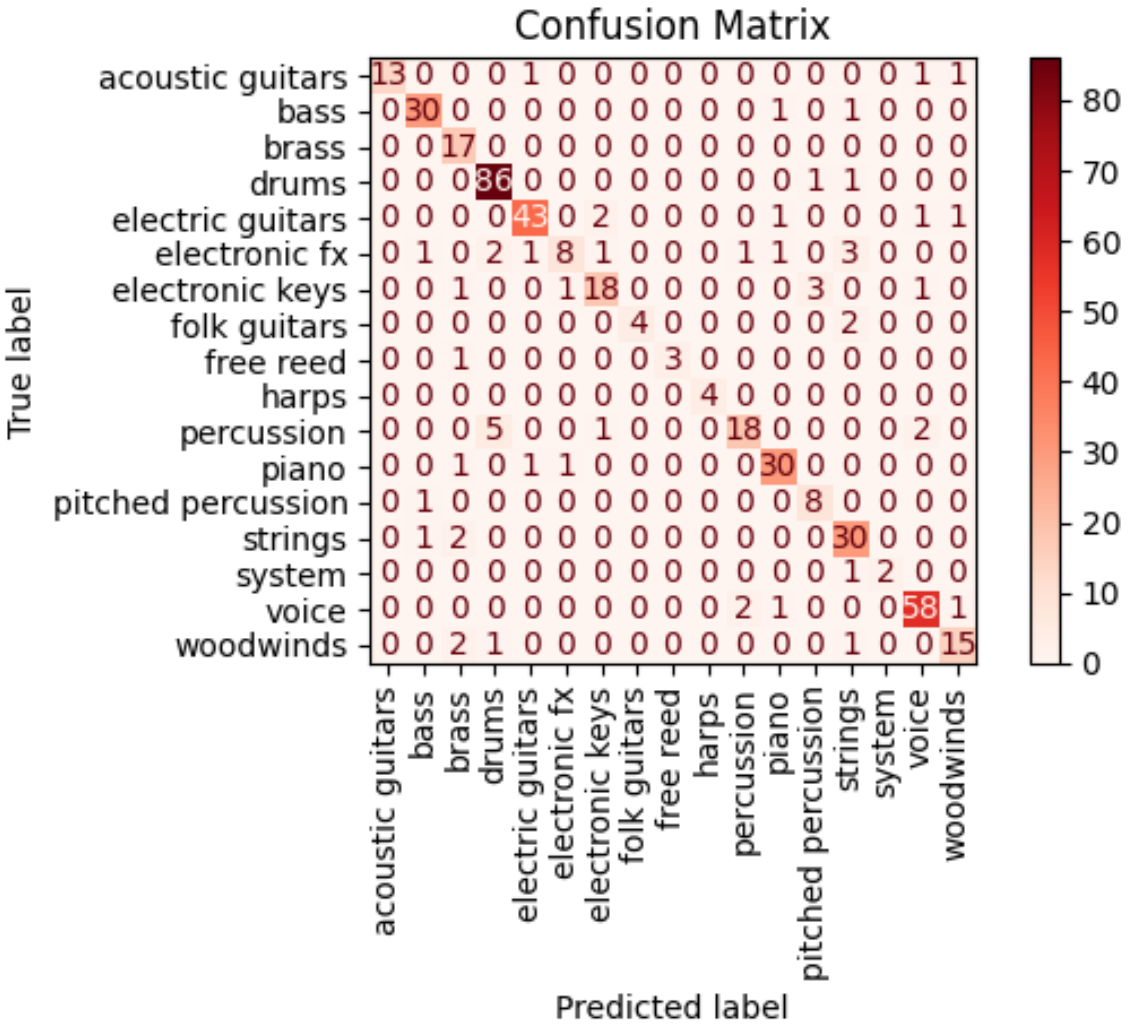

Figure 5.1: Confusion matrix

for this analysis is shown in code 5.2.

Analysis of multilabel predictions on mix samples

[5.2]:
```
...
mixIndices = [i for i, row in enumerate(y_true) if np.sum(row) >
 ↪  1]

for idx in mixIndices[:maxSamples]:
    trueLabels =
     ↪  set(mlb.inverse_transform(np.array([y_true[idx]]))[0])
    predLabels =
     ↪  set(mlb.inverse_transform(np.array([y_pred[idx]]))[0])


    tp = sorted(trueLabels & predLabels)     # True positives
    fn = sorted(trueLabels - predLabels)     # False negatives
    fp = sorted(predLabels - trueLabels)     # False positives

    rows.append({
```

```
        'Sample': idx,
        'True Labels': sorted(trueLabels),
        'Pred Labels': sorted(predLabels),
        'TP': tp,
        'FN': fn,
        'FP': fp,
        'Correct': len(tp),
        'Missed': len(fn),
        'Wrong': len(fp)
    })

dfMix = pd.DataFrame(rows)
...
```

5.5 Summary - Classification Report

To further characterize the model's performance across all instrument classes, a multi-label classification report is generated. The Python code is as follows:

Classification report generation

[5.3]:
```
...
report = classification_report(
    y_test, y_pred_binary,
    target_names=mlb.classes_,
    zero_division=0
)

print(report)
...
```

The classification report is summarized in table 5.1, which reveals significant variance in performance among different classes. The micro-averaged F1-score of 0.84 indicates that the model performs well across all samples. While the macro-averaged F1-score is 0.80, which reflects how the model performs across each class regardless of support, suggesting that performance is relatively balanced but still has trouble with the classes that have fewer samples. Let's take some conclusions based on the classification report:

- High-performing classes such as `drums`, `voice`, and `piano` achieved F1-scores above 0.85, due to higher samples available and more distinct acoustic features.
- `Harps` achieved a high precision, though the support for this class is very low (6), this suggests potential overfitting..
- Classes like `electronic fx`, `folk guitars`, and `percussion` had relatively low recall values, indicating the model struggles to detect them reliably.
- The samples-averaged F1-score of 0.84 further confirms that the model identifies the correct set of instruments with reasonable accuracy.

As last thing is noticeable that while precision is generally high across most classes (macro average = 0.90), recall is lower (macro average = 0.73), suggesting the model is conservative in its predictions and may miss some true positives. This underlines the problem of class imbalance that has been partially resolved by augmenting it as explained in chapter 3.

Class	**Precision**	**Recall**	**F1-score**	**Support**
Acoustic Guitars	0.88	0.70	0.78	20
Bass	0.93	0.84	0.88	44
Brass	0.77	0.81	0.79	21
Drums	0.93	0.90	0.92	102
Electric Guitars	0.92	0.84	0.88	56
Electronic FX	0.80	0.36	0.50	22
Electronic Keys	0.86	0.66	0.75	29
Folk Guitars	0.80	0.50	0.62	8
Free Reed	1.00	0.60	0.75	5
Harps	1.00	0.67	0.80	6
Percussion	0.86	0.56	0.68	32
Piano	0.97	0.78	0.86	40
Pitched Percussion	0.75	0.82	0.78	11
Strings	0.91	0.82	0.86	38
System	1.00	1.00	1.00	4
Voice	0.95	0.85	0.90	74
Woodwinds	0.89	0.74	0.81	23
Micro Avg	0.91	0.78	0.84	535
Macro Avg	0.90	0.73	0.80	535
Weighted Avg	0.91	0.78	0.83	535
Samples Avg	0.85	0.84	0.84	535

Table 5.1: Precision, Recall and F1-score per class on the test

6

Genre & Instrument Occurrence Analysis

6.1 Introduction

This chapter aims to analyze the distribution of instruments across musical genres in the MedleyDB dataset. The goal is to understand how instruments and instrument categories (both fine-grained and coarse) relate to different genres, and to investigate patterns of co-occurrence that may affect classification performance. Four types of analyses were conducted: the original instrument–genre association, the same analysis after grouping instruments into 17 macro-categories, and then the co-occurrence statistics using both fine and grouped labels. The parsing procedure first extracts metadata from each song in the dataset using the `.yaml` files provided. For each song, the genre and the full list of active instruments are recovered and stored in a structured dataset. Each instrument list is parsed and transformed into a binary label matrix using `MultiLabelBinarizer`, as shown in code 6.1.

Genre extraction and binarization

[6.1]:
```
...
dfGenres["labelList"] = dfGenres["label"].str.split("|")

mlbGenresDataset = MultiLabelBinarizer()
songInstrumentsBinary =
 ↪  mlbGenresDataset.fit_transform(dfGenres["labelList"])
...
```

6.2 Instrument–Genre Occurrences (Original Labels)

A first analysis computes the frequency of each instrument per genre using the original labels extracted from the metadata. This results in a large and detailed matrix where rows represent instruments and columns represent genres. An excerpt of the matrix was visualized using a red colormap for better interpretability, as shown in figure 6.1.

6.3 Instrument–Genre Occurrences (Macro Labels)

To obtain a more interpretable and higher-level representation, original labels were mapped into 17 macro-categories (see chapter 2). Each set of instruments per song was transformed

into the corresponding macro-label set. The binary matrix was then recalculated and aggregated by genre, as shown in code 6.2.

Mapping to macro-categories

[6.2]:
```
dfGenres["macroLabelList"] = dfGenres["labelList"].apply(
    lambda instruments: list({labelGroupsDict.get(inst, inst) for
     ↪  inst in instruments})
)

mlbMacro = MultiLabelBinarizer()
macroBinary = mlbMacro.fit_transform(dfGenres["macroLabelList"])
```

The resulting matrix, shown in figure 6.2, offers a clearer view of how broad families of instruments (e.g., strings, brass, keys) are distributed across genres.

6.4 Instrument Co-occurrence (Original Labels)

The next step was to investigate which instruments tend to appear together within the same mix. A co-occurrence matrix was computed by counting pairwise appearances of all instruments across all songs. The result, shown in figure 6.3, highlights clusters of instruments that often play simultaneously.

6.5 Instrument Co-occurrence (Macro Labels)

Finally, the co-occurrence matrix was recalculated using macro-categories. This yields a more compact representation of relationships between broader timbral groups. The result is illustrated in figure 6.4 and helps to understand structural tendencies in arrangement and instrumentation across the dataset. These analyses not only support interpretability of genre-instrument relationships but also help in identifying potential sources of model confusion due to frequent instrument co-occurrences. Such knowledge can guide the design of future architectures and post-processing strategies tailored to music classification tasks.

6.6 Conclusion

The genre-instrument analysis conducted in this chapter provided useful insights into the structure of the dataset. Specifically:

- It showed that certain instruments are strongly associated with specific genres, which may influence the classification results.
- The co-occurrence matrices revealed which instruments tend to appear together, both using detailed labels and macro-categories.
- This information can support future improvements, such as better data balancing or the use of hierarchical models that take genre and instrument groups into account.

Overall, the analysis of metadata adds an important layer of understanding that can support both model design and interpretation of classification results.

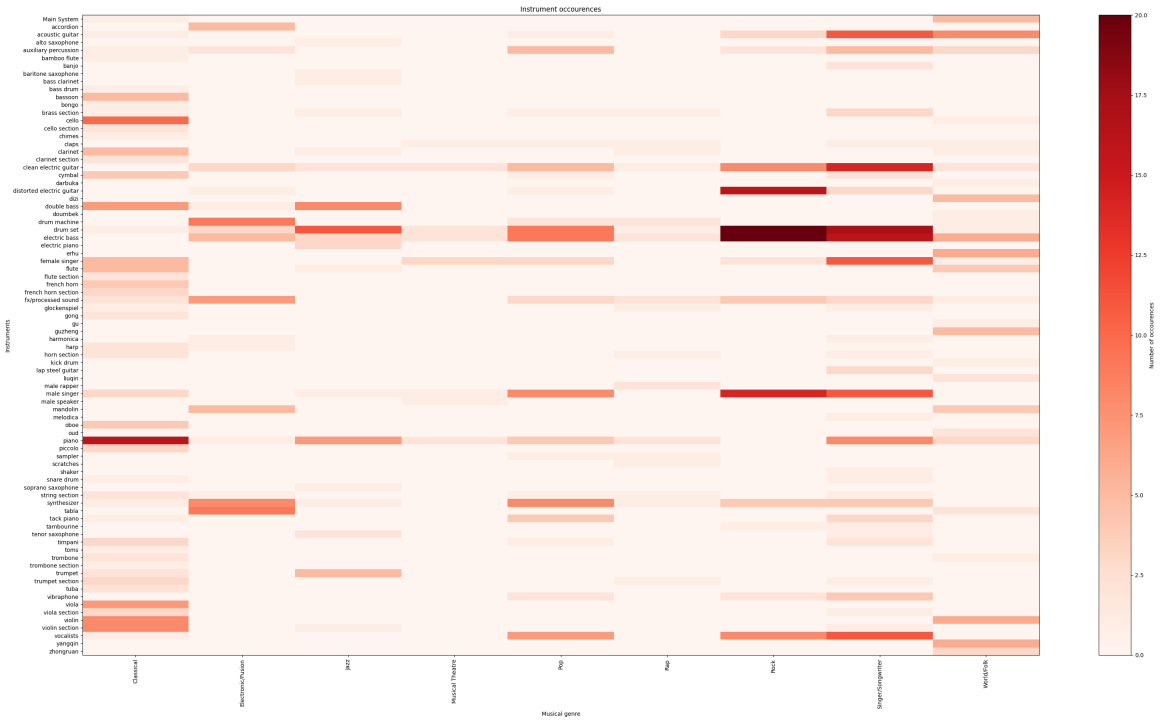

Figure 6.1: Instrument–Genre occurrences using original instrument labels

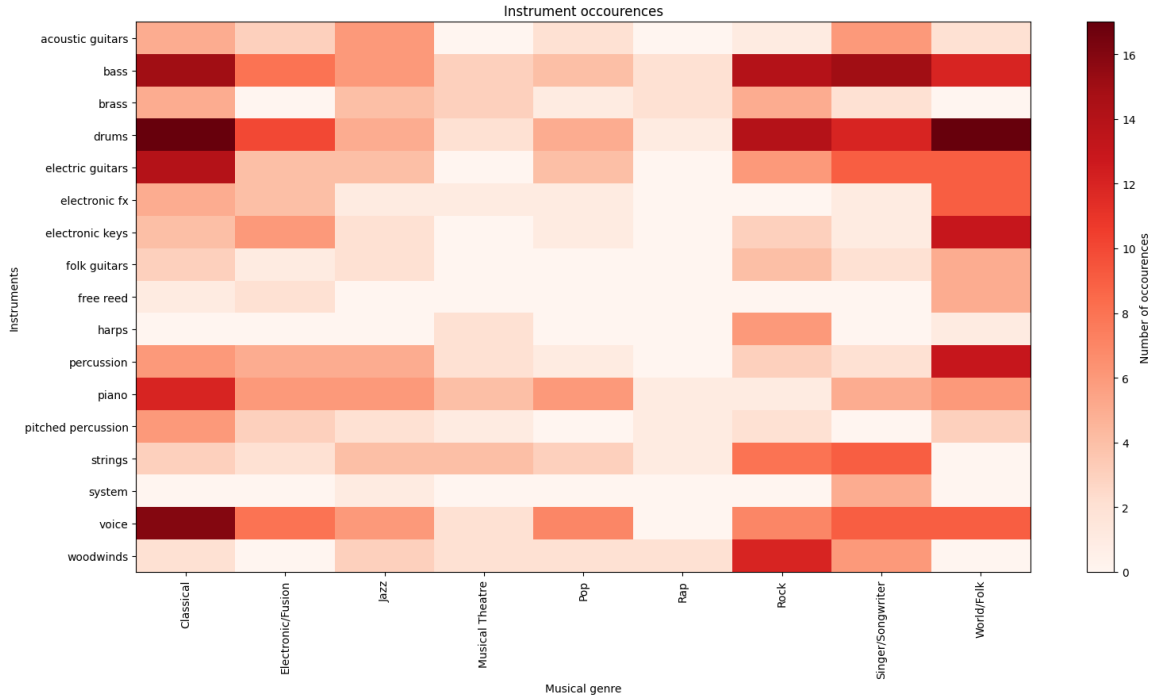

Figure 6.2: Instrument–Genre occurrences using 17 macro-categories

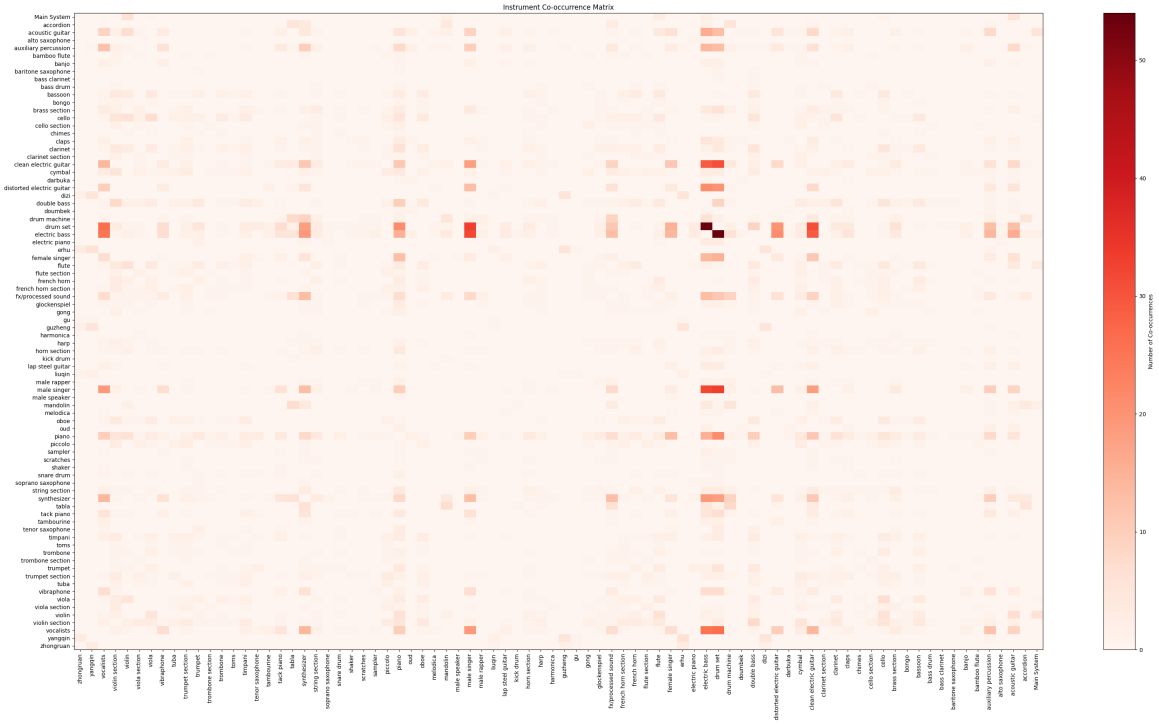

Figure 6.3: Instrument co-occurrence using original instrument labels

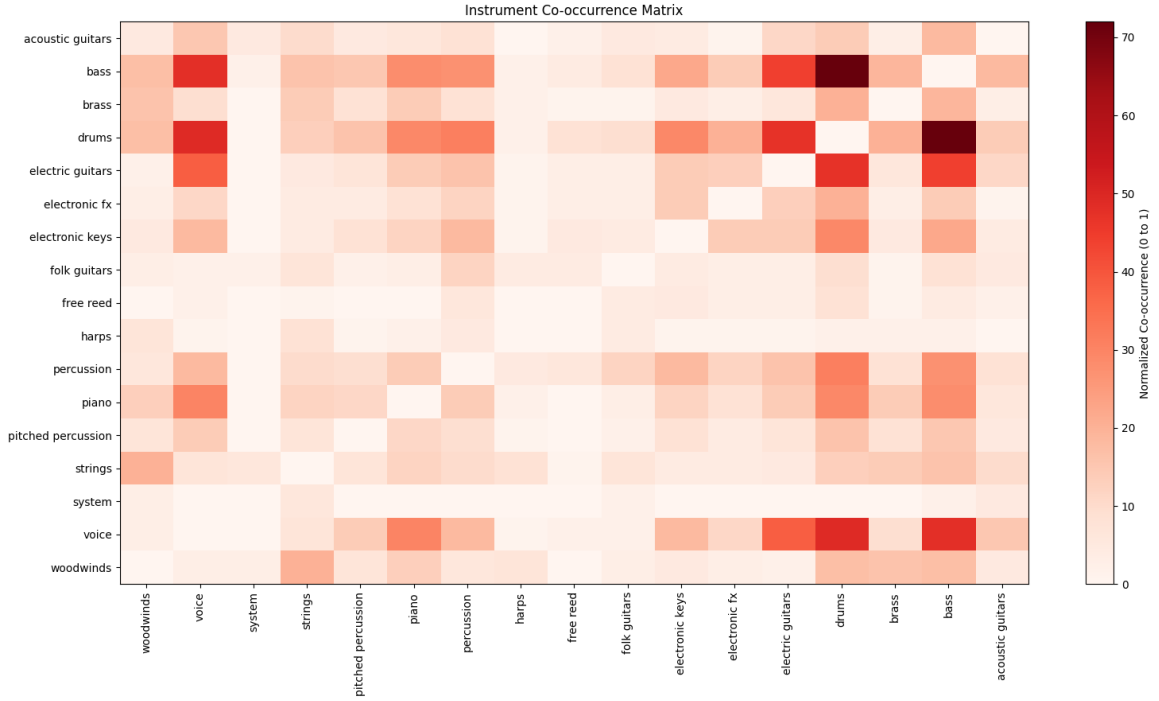

Figure 6.4: Instrument co-occurrence using 17 macro-categories

7 Conclusions

This project tackled the problem of multilabel musical instrument classification using deep learning methods applied to the MedleyDB dataset. A full pipeline was implemented—from dataset assembly and preprocessing to model training and evaluation—with a particular focus on ensuring modularity, interpretability and generalization. The model's input consisted of mel-spectrograms derived from carefully segmented audio excerpts. These segments were selected based on pitch annotations or amplitude heuristics to ensure the presence of active instruments. Several preprocessing strategies were tested to standardize audio duration, enhance class balance and manage overlapping sources. A convolutional neural network was employed for classification, leveraging successive convolutional and pooling layers, dropout regularization and global average pooling before multilabel sigmoid prediction.

7.1 Iterative Development Strategy

Throughout the project, a number of experimental configurations were explored to optimize performance under practical computational constraints. Initially, approximately 15 macro-categories were considered and only `MIX` and `STEM` files were used to form the dataset. This configuration was split using a classic 70%-15%-15% train-validation-test ratio. However, early experiments revealed that the validation set was too limited to capture class variability and so a revised split of 60%-24%-16% was adopted. To further enrich the data, `RAW` files were also introduced and the number of target classes was temporarily reduced to improve learning stability. These intermediate experiments yielded valuable insights into data availability, class balance and model capacity. The final configuration resulted in a dataset encompassing 17 macro-categories, incorporating all `MIX`, `STEM` and `RAW` files available in MedleyDB and using iterative stratification to preserve multilabel distributions.

7.2 Performance Summary

The resulting model demonstrated strong classification capabilities across most classes. Frequent categories such as `drums`, `voice` and `electric guitars` achieved consistently high accuracy and AUC scores, while rare or harmonically overlapping instruments (e.g., `clarinet`, `harp`) were more prone to misclassification. Confusion matrices and classification reports supported these findings, highlighting the effects of class imbalance despite data augmentation. The genre-level analysis added a complementary perspective, exposing stylistic associations between genres and instrument combinations and revealing patterns of co-occurrence both at the fine-grained and macro-category levels.

7.3 Limitations

Several constraints impacted the scope and scalability of the project. First and foremost, limited computational resources restricted the volume of data and model complexity that could be handled. As a result, not all instruments in the original MedleyDB taxonomy were included and the number of training epochs and batch size had to be tuned conservatively. Additionally, GPU memory limitations imposed bounds on input spectrogram resolution and data augmentation depth. The use of raw audio for training, while beneficial for diversity, introduced additional preprocessing and memory demands. Furthermore, while the use of macro-categories helped reduce fragmentation and improve model performance, it also introduced a level of abstraction that may obscure finer instrument-level distinctions. This trade-off was necessary to ensure model learnability but remains a limitation in scenarios requiring detailed timbral resolution.

7.4 Final Remarks

This study highlights the effectiveness of deep convolutional architectures for real-world multilabel music analysis tasks and demonstrates how thoughtful preprocessing and iterative experimentation can overcome resource limitations. The methodologies and insights developed in this work lay the groundwork for future advancements in music information retrieval, particularly in multimodal and metadata-rich environments.