Classifying Audio Recordings into Guitar and Bass with Logistic Regression and a Decision Tree

Introduction

In modern hobbyist music production, the signal processing that used to be done with analog circuits is now largely digitized. Many companies produce plugins that can be applied, for example, to clean guitar or bass signal recorded on a laptop. Currently, however, one must manually specify whether they want to select from guitar or bass effect rigs, which differ significantly from each other. In my opinion, it is natural to automate this step requiring no creative input from the artist. To achieve this, the software should detect the instrument class of a recording.

In this report, an attempt at solving this classification problem with machine learning is made. First the problem is formulated, two ML methods are applied, and finally the results are analyzed. The code used is attached as an appendix to this report.

Problem Formulation

Audio recordings of unprocessed guitar and bass signals are used as datapoints. The recordings have a sample rate of 44100 Hz and are 24-bit .wav files. They are 2.5 seconds long and feature one steel stringed acoustic guitar, one nylon stringed acoustic guitar, two models of electric guitars, and two different electric basses. The dataset is made by the author and includes 1200 datapoints. Currently, it is a private dataset. There are 150 recordings of each of the four different guitars and 300 of both basses, so the data is balanced. The acoustic guitars have been recorded with the same flat dynamic microphone to minimize room reflections and noise. The electric instruments have been recorded directly using their pickups and the same preamplifier as the acoustic ones. Thus, the environment is as fair as possible. The recordings have been sliced from longer ones of varied authentic playing, as this is more realistic than isolated single notes. This introduces some noise in the form of partially silent recordings, however. The average volume of each instrument has been adjusted to be similar.

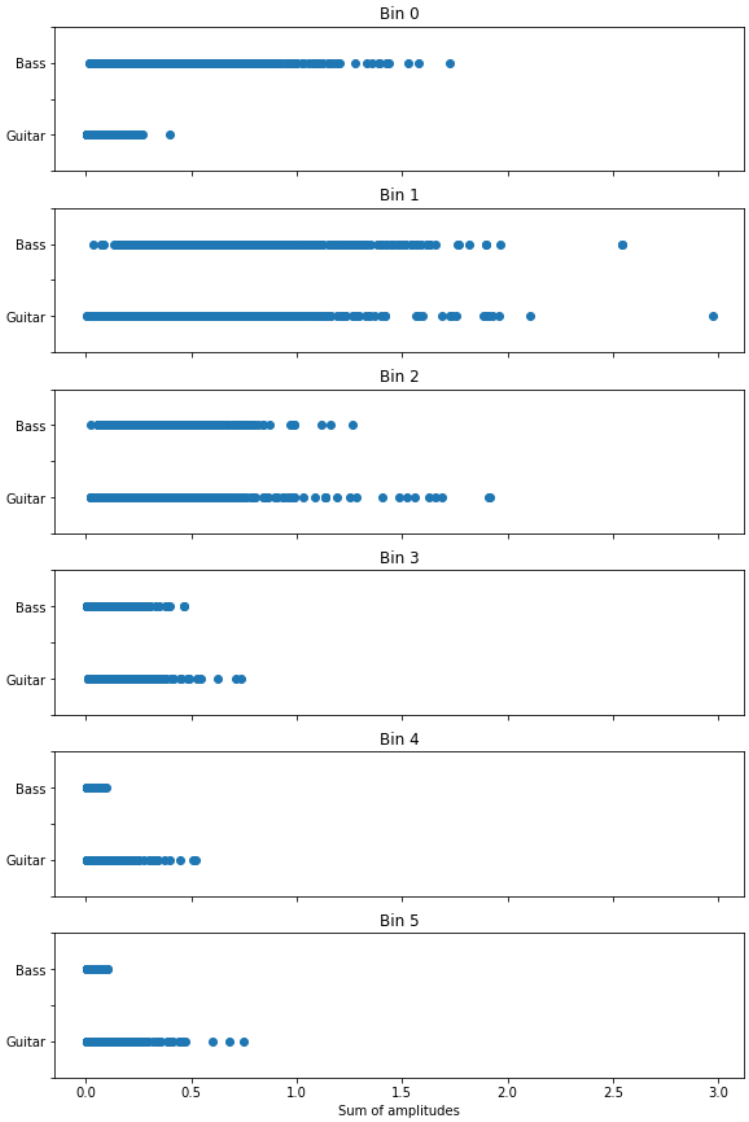
The features are sums of the amplitudes in the following six frequency ranges: [0 Hz, 100 Hz[, [100 Hz, 350 Hz[, [350 Hz, 1000 Hz[, [2 kHz, 5 kHz[, [5 kHz, 10 kHz[, and [10 kHz, 20 kHz]. The amplitude sums are floating point numbers. These bins were chosen based on experience from music mixing and visualizing signals with graphical equalizers. The temporal information of the signals has been intentionally discarded, as the playing and the slicing of the recordings is arbitrary. Based on a search of previous similar applications, others have successfully classified instruments according to more complex features, such as Mel frequency cepstral coefficients (Varpe, 2019). In this project, the aim is to see if binned frequency information is sufficient in the case of guitar and bass.

The label is the class of the datapoint: 0 = guitar and 1 = bass. Thus, the problem is a binary classification problem.

Methods

In the section "Preprocessing" of the Jupyter notebook (see Appendix), the 1200 audio recordings are transformed into numerical form, before machine learning is applied. The files are loaded as vectors consisting of the amplitudes of each sample. This is done with the Python library librosa. In order to have access to the amplitude information for a certain frequency present in the recordings, Fourier transformation is applied to the vectors of time. The efficient Fast Fourier Transform (FFT) algorithm implemented in the scipy library is used for this. The amplitude information is obtained from the resulting complex values by taking the absolute value. Only positive frequencies are considered in this scenario. The amplitude values corresponding to frequencies in each of the six bins are then summed together, forming the features of the datapoint. These make up the feature matrix with 1200 rows and 6 columns.

The bins described above were chosen based on domain knowledge of the instruments and musical equalization. Bass intuitively has more energy in the lower frequencies while guitars tend to have more brittle high frequencies. Plot 1, made in the section "Plotting for Intuition" of the notebook, justifies this choice by showing higher amplitude values for bass recordings in the lower bins and guitar recordings for the higher bins.



Plot 1. Scatter plot of the relation of the labels and features of datapoints.

The full dataset is split into three parts in the notebook section “Applying Machine Learning”. 33%, 396 datapoints, are reserved for testing and the remaining 67%, 804 datapoints, are used for training and validation. This is done with the sklearn.model_selection function train_test_split, with shuffling on. An equal split is chosen to minimize the risk of getting biased sets for training and validation or testing. For example, getting a significant proportion of recordings that happened right as the playing halted due to a chord change in the same set could be detrimental. To further improve the fairness of the training and validation phases, k-fold cross validation is utilized. The 67% not saved for testing are split into 20 folds, each of which is used as validation for the map trained on the remaining 19 once. 20 folds is a suitable amount, as the effective dimension of the linear model with six features is still considerably smaller than each new training set. The average training and validation accuracies from the cross validation, as well as the test accuracy, are computed to evaluate the predictions obtained for both machine learning methods.

For the first machine learning method, I chose linear regression. Its linear hypothesis space seems like a reasonable starting point since it is the simplest model available. I find it intuitive to believe it is possible to separate between a bass and a guitar recording with a linear map by applying higher weights to the bins at both ends of the spectrum, as plot 1 shows considerable correlation in these. The sklearn.linear_model.LogisticRegression implementation of logistic regression uses the logistic loss as its loss function, which is the primary reason for choosing it as the loss function here. Logistic regression can be interpreted as an approximation of a random variable with a probability density parametrized by the weights (Jung, 2022), which also seems to support the combination of the linear space and logistic loss, as the data is random playing. Logistic loss is not particularly robust towards outliers, as it increases monotonically for a wrong classification. I assume it still suits the task at hand, as it is physically impossible to create severe outliers with these instruments. This is the case at least if the edge bins are assigned the highest weights. A clean guitar cannot produce much energy in the 0-100 Hz region due to its tuning, for example.

The second ML method I used is the decision tree classifier. It has a piecewise constant hypothesis space, thus making it a potential fit for the binned features. I tested several values for the depth of the decision tree, and the empirical results suggest that 7 works best for this scenario. With higher values, the training accuracy quickly rises to 1.0 indicating severe overfitting, and with values differing from 7 in either direction, the testing accuracy falls. Another empirically chosen parameter is the loss function or criterion used. I chose to maximize entropy, or information gain, over minimizing Gini impurity, as it delivered a higher accuracy more consistently. With this parameter set, the algorithm implemented in sklearn.tree.DecisionTreeClassifier places the splits which decrease the uncertainty of the prediction the most in the earliest nodes.

Results

For logistic regression the average training accuracy is 88.3% and the average validation accuracy is 88.2% for the pseudorandom shuffling used. For the decision tree trained with the same 20-fold splits and shuffles the average training accuracy is 97.7% and the average validation accuracy is 92.7%. It is worth noting that these values change slightly over different runs, as they are averages. According to the validation errors, the decision tree provides better results in this application. The test accuracy for the decision tree, computed on a new subset of data, is 91.9%. As the testing and validation accuracies are perceivably lower than the training accuracy, I suspect some overfitting to take place despite the limited tree depth. However, this training accuracy is higher than that of the

logistic regressor, 86.9%, confirming the superiority of the decision tree classifier. For this reason, the decision tree is the final chosen method.

Conclusion

This report describes the application of machine learning to the problem of correctly classifying guitar and bass recordings, a task helpful in music production. Signal amplitudes within six frequency bins are considered features, and the binary category of 0 = guitar or 1 = bass is the label. Data preprocessing, including Fast Fourier Transforming, is done to extract relevant feature information from the recordings in the time domain. The data is then split into three equal subsets. The training and validation subsets are further processed with 20-fold cross validation, to ensure fairness. Logistic regression and decision tree classification are applied to these shuffled folds. The quality of the predictions is assessed using the average accuracy metric over the folds. This validation yields the result that the decision tree is more suitable for this problem.

Although the testing accuracy of 91.9% obtained for the decision tree is rather good, I believe there is room for improvement. For example, if a commercial product with accuracy close to 100% were to be developed, more data with variety in the recording environment, instruments, microphones, microphone placement, and playing style should be collected.

Different machine learning methods should also be considered, especially ones which are more robust against noisy data. Recordings containing silence or noise, such as speech or room noise, are inevitable in a realistic setting. Especially the ability of the logistic regressor, making use of logistic loss, to handle such noise is limited. This is reflected in the results. The performance of the decision tree is mostly limited by its tendency to overfit to the training data when a large number of nodes is used.

References

Jung, A., 2022. *Machine Learning: The Basics.* Singapore: Springer.

Varpe, Jørgen., 2019. Classification of string instruments [online]. *The MCT Blog.* 14 September. [viewed 08 February 2022]. Available from: https://mct-master.github.io/machine-learning/2019/09/14/ML-Classify-instr.html

Appendix

March 28, 2022

[1]:
```
import librosa # Used for processing audio data
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq, fftshift
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import accuracy_score
```

1 Preprocessing

Fetching the raw audio data and preprocessing it to create a numerical feature matrix and label vector.

[2]:
```
# Defining constants
fs = 44100 # sample rate
t = 1/fs # sample spacing
d = 2.5 # duration in seconds
n = int(d * fs) # number of samples in duration

X = np.empty((1200,6)) # Creating the feature matrix of shape (number of␣
↪samples = 1200, number of features = 6)
y = np.empty((1200,)) # Creating the label vector of shape (1200,)

# Fetching the raw audio data and preprocessing it

i = 0 # Index of current feature and label

# Steel stringed acoustic guitar
for j in range(1, 150+1):
    s, _ = librosa.load(f"data/steel_steel_REC_{j}.wav") # Loading the audio␣
↪file as amplitude values for each sample
    s = s[0:n] # Cutting the recording to have exact duration d
    S = 2.0/n * np.abs(fft(s)[:n//2]) # The scaled absolute values of the␣
↪discrete Fourier transformation of the
                                      # recording on the positive frequency␣
↪axis, i.e. the amplitudes.
```

```
    freq = fftfreq(n=n, d=t)[:n//2] # The positive frequencies present in the␣
↪recording

    # Finding indices of the edge frequencies of the bins
    ind_0Hz, = np.where(np.isclose(freq, 0, atol=1/(t*n))) # Indices close to 0␣
↪Hz
    ind_100Hz, = np.where(np.isclose(freq, 100, atol=1/(t*n))) # Indices close␣
↪to 100 Hz
    ind_350Hz, = np.where(np.isclose(freq, 350, atol=1/(t*n))) # Indices close␣
↪to 350 Hz
    ind_1kHz, = np.where(np.isclose(freq, 1000, atol=1/(t*n))) # Indices close␣
↪to 1 kHz
    ind_2kHz, = np.where(np.isclose(freq, 2000, atol=1/(t*n))) # Indices close␣
↪to 2 kHz
    ind_5kHz, = np.where(np.isclose(freq, 5000, atol=1/(t*n))) # Indices close␣
↪to 5 kHz
    ind_10kHz, = np.where(np.isclose(freq, 10000, atol=1/(t*n))) # Indices␣
↪close to 10 kHz
    ind_20kHz, = np.where(np.isclose(freq, 20000, atol=1/(t*n))) # Indices␣
↪close to 20 kHz

    # Bins
    bin_0 = np.arange(ind_0Hz[0], ind_100Hz[0]) # [0 Hz, 100 Hz[
    bin_1 = np.arange(ind_100Hz[0], ind_350Hz[0]) # [100 Hz, 350 Hz[
    bin_2 = np.arange(ind_350Hz[0], ind_1kHz[0]) # [350 Hz, 1000 Hz[
    bin_3 = np.arange(ind_2kHz[0], ind_5kHz[0]) # [2 kHz, 5 kHz[
    bin_4 = np.arange(ind_5kHz[0], ind_10kHz[0]) # [5 kHz, 10 kHz[
    bin_5 = np.arange(ind_10kHz[0], ind_20kHz[0] + 1) # [10 kHz, 20 kHz]

    # Summing over the amplitude values corresponding to the frequency bins
    A_0 = sum(S[bin_0])
    A_1 = sum(S[bin_1])
    A_2 = sum(S[bin_2])
    A_3 = sum(S[bin_3])
    A_4 = sum(S[bin_4])
    A_5 = sum(S[bin_5])

    # Entering the features into the matrix
    X[i,0] = A_0
    X[i,1] = A_1
    X[i,2] = A_2
    X[i,3] = A_3
    X[i,4] = A_4
    X[i,5] = A_5

    y[i] = 0 # Entering the label into the vector, 0 for "guitar"
```

```
    i += 1 # Increasing the index

# Repeating for other instruments
# Nylon stringed acoustic guitar
for j in range(1, 150+1):
    s, _ = librosa.load(f"data/nylon_nylon_REC_{j}.wav") # Loading the audio␣
↪file as amplitude values for each sample
    s = s[0:n] # Cutting the recording to have duration d
    S = 2.0/n * np.abs(fft(s)[:n//2]) # The scaled absolute values of the␣
↪discrete Fourier transformation of the
                                      # recording on the positive frequency␣
↪axis, i.e. the amplitudes.
    freq = fftfreq(n=n, d=t)[:n//2] # The positive frequencies present in the␣
↪recording

    # Finding indices of the edge frequencies of the bins
    ind_0Hz, = np.where(np.isclose(freq, 0, atol=1/(t*n))) # Indices close to 0␣
↪Hz
    ind_100Hz, = np.where(np.isclose(freq, 100, atol=1/(t*n))) # Indices close␣
↪to 100 Hz
    ind_350Hz, = np.where(np.isclose(freq, 350, atol=1/(t*n))) # Indices close␣
↪to 350 Hz
    ind_1kHz, = np.where(np.isclose(freq, 1000, atol=1/(t*n))) # Indices close␣
↪to 1 kHz
    ind_2kHz, = np.where(np.isclose(freq, 2000, atol=1/(t*n))) # Indices close␣
↪to 2 kHz
    ind_5kHz, = np.where(np.isclose(freq, 5000, atol=1/(t*n))) # Indices close␣
↪to 5 kHz
    ind_10kHz, = np.where(np.isclose(freq, 10000, atol=1/(t*n))) # Indices␣
↪close to 10 kHz
    ind_20kHz, = np.where(np.isclose(freq, 20000, atol=1/(t*n))) # Indices␣
↪close to 20 kHz

    # Bins
    bin_0 = np.arange(ind_0Hz[0], ind_100Hz[0]) # [0 Hz, 100 Hz[
    bin_1 = np.arange(ind_100Hz[0], ind_350Hz[0]) # [100 Hz, 350 Hz[
    bin_2 = np.arange(ind_350Hz[0], ind_1kHz[0]) # [350 Hz, 1000 Hz[
    bin_3 = np.arange(ind_2kHz[0], ind_5kHz[0]) # [2 kHz, 5 kHz[
    bin_4 = np.arange(ind_5kHz[0], ind_10kHz[0]) # [5 kHz, 10 kHz[
    bin_5 = np.arange(ind_10kHz[0], ind_20kHz[0] + 1) # [10 kHz, 20 kHz]

    # Summing over the amplitude values corresponding to the frequency bins
    A_0 = sum(S[bin_0])
    A_1 = sum(S[bin_1])
    A_2 = sum(S[bin_2])
```

```
    A_3 = sum(S[bin_3])
    A_4 = sum(S[bin_4])
    A_5 = sum(S[bin_5])

    # Entering the features into the matrix
    X[i,0] = A_0
    X[i,1] = A_1
    X[i,2] = A_2
    X[i,3] = A_3
    X[i,4] = A_4
    X[i,5] = A_5

    y[i] = 0 # Entering the label into the vector, 0 for "guitar"

    i += 1 # Increasing the index

# Electric guitar 1
for j in range(1, 150+1):
    s, _ = librosa.load(f"data/egtr1_egtr1_REC_{j}.wav") # Loading the audio␣
↪file as amplitude values for each sample
    s = s[0:n] # Cutting the recording to have duration d
    S = 2.0/n * np.abs(fft(s)[:n//2]) # The scaled absolute values of the␣
↪discrete Fourier transformation of the
                                      # recording on the positive frequency␣
↪axis, i.e. the amplitudes.
    freq = fftfreq(n=n, d=t)[:n//2] # The positive frequencies present in the␣
↪recording

    # Finding indices of the edge frequencies of the bins
    ind_0Hz, = np.where(np.isclose(freq, 0, atol=1/(t*n))) # Indices close to 0␣
↪Hz
    ind_100Hz, = np.where(np.isclose(freq, 100, atol=1/(t*n))) # Indices close␣
↪to 100 Hz
    ind_350Hz, = np.where(np.isclose(freq, 350, atol=1/(t*n))) # Indices close␣
↪to 350 Hz
    ind_1kHz, = np.where(np.isclose(freq, 1000, atol=1/(t*n))) # Indices close␣
↪to 1 kHz
    ind_2kHz, = np.where(np.isclose(freq, 2000, atol=1/(t*n))) # Indices close␣
↪to 2 kHz
    ind_5kHz, = np.where(np.isclose(freq, 5000, atol=1/(t*n))) # Indices close␣
↪to 5 kHz
    ind_10kHz, = np.where(np.isclose(freq, 10000, atol=1/(t*n))) # Indices␣
↪close to 10 kHz
    ind_20kHz, = np.where(np.isclose(freq, 20000, atol=1/(t*n))) # Indices␣
↪close to 20 kHz
```

```
    # Bins
    bin_0 = np.arange(ind_0Hz[0], ind_100Hz[0]) # [0 Hz, 100 Hz[
    bin_1 = np.arange(ind_100Hz[0], ind_350Hz[0]) # [100 Hz, 350 Hz[
    bin_2 = np.arange(ind_350Hz[0], ind_1kHz[0]) # [350 Hz, 1000 Hz[
    bin_3 = np.arange(ind_2kHz[0], ind_5kHz[0]) # [2 kHz, 5 kHz[
    bin_4 = np.arange(ind_5kHz[0], ind_10kHz[0]) # [5 kHz, 10 kHz[
    bin_5 = np.arange(ind_10kHz[0], ind_20kHz[0] + 1) # [10 kHz, 20 kHz]

    # Summing over the amplitude values corresponding to the frequency bins
    A_0 = sum(S[bin_0])
    A_1 = sum(S[bin_1])
    A_2 = sum(S[bin_2])
    A_3 = sum(S[bin_3])
    A_4 = sum(S[bin_4])
    A_5 = sum(S[bin_5])

    # Entering the features into the matrix
    X[i,0] = A_0
    X[i,1] = A_1
    X[i,2] = A_2
    X[i,3] = A_3
    X[i,4] = A_4
    X[i,5] = A_5

    y[i] = 0 # Entering the label into the vector, 0 for "guitar"

    i += 1 # Increasing the index

# Electric guitar 2
for j in range(1, 150+1):
    s, _ = librosa.load(f"data/egtr2_egtr2_REC_{j}.wav") # Loading the audio␣
↪file as amplitude values for each sample
    s = s[0:n] # Cutting the recording to have duration d
    S = 2.0/n * np.abs(fft(s)[:n//2]) # The scaled absolute values of the␣
↪discrete Fourier transformation of the
                                      # recording on the positive frequency␣
↪axis, i.e. the amplitudes.
    freq = fftfreq(n=n, d=t)[:n//2] # The positive frequencies present in the␣
↪recording

    # Finding indices of the edge frequencies of the bins
    ind_0Hz, = np.where(np.isclose(freq, 0, atol=1/(t*n))) # Indices close to 0␣
↪Hz
    ind_100Hz, = np.where(np.isclose(freq, 100, atol=1/(t*n))) # Indices close␣
↪to 100 Hz
```

```
    ind_350Hz, = np.where(np.isclose(freq, 350, atol=1/(t*n))) # Indices close␣
↪to 350 Hz
    ind_1kHz, = np.where(np.isclose(freq, 1000, atol=1/(t*n))) # Indices close␣
↪to 1 kHz
    ind_2kHz, = np.where(np.isclose(freq, 2000, atol=1/(t*n))) # Indices close␣
↪to 2 kHz
    ind_5kHz, = np.where(np.isclose(freq, 5000, atol=1/(t*n))) # Indices close␣
↪to 5 kHz
    ind_10kHz, = np.where(np.isclose(freq, 10000, atol=1/(t*n))) # Indices␣
↪close to 10 kHz
    ind_20kHz, = np.where(np.isclose(freq, 20000, atol=1/(t*n))) # Indices␣
↪close to 20 kHz

    # Bins
    bin_0 = np.arange(ind_0Hz[0], ind_100Hz[0]) # [0 Hz, 100 Hz[
    bin_1 = np.arange(ind_100Hz[0], ind_350Hz[0]) # [100 Hz, 350 Hz[
    bin_2 = np.arange(ind_350Hz[0], ind_1kHz[0]) # [350 Hz, 1000 Hz[
    bin_3 = np.arange(ind_2kHz[0], ind_5kHz[0]) # [2 kHz, 5 kHz[
    bin_4 = np.arange(ind_5kHz[0], ind_10kHz[0]) # [5 kHz, 10 kHz[
    bin_5 = np.arange(ind_10kHz[0], ind_20kHz[0] + 1) # [10 kHz, 20 kHz]

    # Summing over the amplitude values corresponding to the frequency bins
    A_0 = sum(S[bin_0])
    A_1 = sum(S[bin_1])
    A_2 = sum(S[bin_2])
    A_3 = sum(S[bin_3])
    A_4 = sum(S[bin_4])
    A_5 = sum(S[bin_5])

    # Entering the features into the matrix
    X[i,0] = A_0
    X[i,1] = A_1
    X[i,2] = A_2
    X[i,3] = A_3
    X[i,4] = A_4
    X[i,5] = A_5

    y[i] = 0 # Entering the label into the vector, 0 for "guitar"

    i += 1 # Increasing the index

# Bass 1
for j in range(1, 300+1):
    s, _ = librosa.load(f"data/bass1_bass1_REC_{j}.wav") # Loading the audio␣
↪file as amplitude values for each sample
    s = s[0:n] # Cutting the recording to have duration d
```

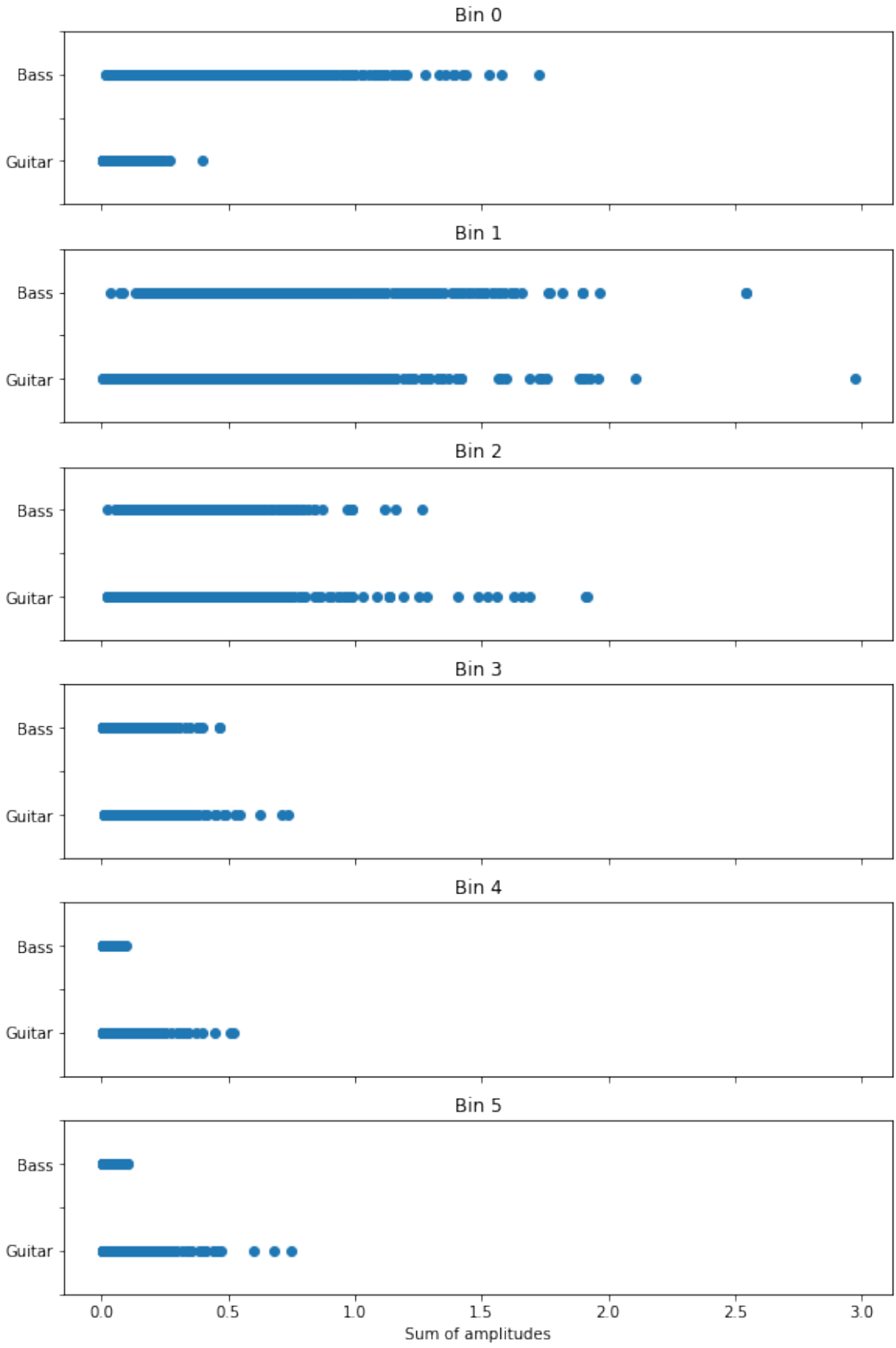
```
   S = 2.0/n * np.abs(fft(s)[:n//2]) # The scaled absolute values of the␣
↪discrete Fourier transformation of the
                                     # recording on the positive frequency␣
↪axis, i.e. the amplitudes.
   freq = fftfreq(n=n, d=t)[:n//2] # The positive frequencies present in the␣
↪recording

   # Finding indices of the edge frequencies of the bins
   ind_0Hz, = np.where(np.isclose(freq, 0, atol=1/(t*n))) # Indices close to 0␣
↪Hz
   ind_100Hz, = np.where(np.isclose(freq, 100, atol=1/(t*n))) # Indices close␣
↪to 100 Hz
   ind_350Hz, = np.where(np.isclose(freq, 350, atol=1/(t*n))) # Indices close␣
↪to 350 Hz
   ind_1kHz, = np.where(np.isclose(freq, 1000, atol=1/(t*n))) # Indices close␣
↪to 1 kHz
   ind_2kHz, = np.where(np.isclose(freq, 2000, atol=1/(t*n))) # Indices close␣
↪to 2 kHz
   ind_5kHz, = np.where(np.isclose(freq, 5000, atol=1/(t*n))) # Indices close␣
↪to 5 kHz
   ind_10kHz, = np.where(np.isclose(freq, 10000, atol=1/(t*n))) # Indices␣
↪close to 10 kHz
   ind_20kHz, = np.where(np.isclose(freq, 20000, atol=1/(t*n))) # Indices␣
↪close to 20 kHz

   # Bins
   bin_0 = np.arange(ind_0Hz[0], ind_100Hz[0]) # [0 Hz, 100 Hz[
   bin_1 = np.arange(ind_100Hz[0], ind_350Hz[0]) # [100 Hz, 350 Hz[
   bin_2 = np.arange(ind_350Hz[0], ind_1kHz[0]) # [350 Hz, 1000 Hz[
   bin_3 = np.arange(ind_2kHz[0], ind_5kHz[0]) # [2 kHz, 5 kHz[
   bin_4 = np.arange(ind_5kHz[0], ind_10kHz[0]) # [5 kHz, 10 kHz[
   bin_5 = np.arange(ind_10kHz[0], ind_20kHz[0] + 1) # [10 kHz, 20 kHz]

   # Summing over the amplitude values corresponding to the frequency bins
   A_0 = sum(S[bin_0])
   A_1 = sum(S[bin_1])
   A_2 = sum(S[bin_2])
   A_3 = sum(S[bin_3])
   A_4 = sum(S[bin_4])
   A_5 = sum(S[bin_5])

   # Entering the features into the matrix
   X[i,0] = A_0
   X[i,1] = A_1
   X[i,2] = A_2
   X[i,3] = A_3
```

```
    X[i,4] = A_4
    X[i,5] = A_5

    y[i] = 1 # Entering the label into the vector, 1 for "bass"

    i += 1 # Increasing the index

# Bass 2
for j in range(1, 300+1):
    s, _ = librosa.load(f"data/bass2_bass2_REC_{j}.wav") # Loading the audio␣
↪file as amplitude values for each sample
    s = s[0:n] # Cutting the recording to have duration d
    S = 2.0/n * np.abs(fft(s)[:n//2]) # The scaled absolute values of the␣
↪discrete Fourier transformation of the
                                      # recording on the positive frequency␣
↪axis, i.e. the amplitudes.
    freq = fftfreq(n=n, d=t)[:n//2] # The positive frequencies present in the␣
↪recording

    # Finding indices of the edge frequencies of the bins
    ind_0Hz, = np.where(np.isclose(freq, 0, atol=1/(t*n))) # Indices close to 0␣
↪Hz
    ind_100Hz, = np.where(np.isclose(freq, 100, atol=1/(t*n))) # Indices close␣
↪to 100 Hz
    ind_350Hz, = np.where(np.isclose(freq, 350, atol=1/(t*n))) # Indices close␣
↪to 350 Hz
    ind_1kHz, = np.where(np.isclose(freq, 1000, atol=1/(t*n))) # Indices close␣
↪to 1 kHz
    ind_2kHz, = np.where(np.isclose(freq, 2000, atol=1/(t*n))) # Indices close␣
↪to 2 kHz
    ind_5kHz, = np.where(np.isclose(freq, 5000, atol=1/(t*n))) # Indices close␣
↪to 5 kHz
    ind_10kHz, = np.where(np.isclose(freq, 10000, atol=1/(t*n))) # Indices␣
↪close to 10 kHz
    ind_20kHz, = np.where(np.isclose(freq, 20000, atol=1/(t*n))) # Indices␣
↪close to 20 kHz

    # Bins
    bin_0 = np.arange(ind_0Hz[0], ind_100Hz[0]) # [0 Hz, 100 Hz[
    bin_1 = np.arange(ind_100Hz[0], ind_350Hz[0]) # [100 Hz, 350 Hz[
    bin_2 = np.arange(ind_350Hz[0], ind_1kHz[0]) # [350 Hz, 1000 Hz[
    bin_3 = np.arange(ind_2kHz[0], ind_5kHz[0]) # [2 kHz, 5 kHz[
    bin_4 = np.arange(ind_5kHz[0], ind_10kHz[0]) # [5 kHz, 10 kHz[
    bin_5 = np.arange(ind_10kHz[0], ind_20kHz[0] + 1) # [10 kHz, 20 kHz]

    # Summing over the amplitude values corresponding to the frequency bins
```

```
    A_0 = sum(S[bin_0])
    A_1 = sum(S[bin_1])
    A_2 = sum(S[bin_2])
    A_3 = sum(S[bin_3])
    A_4 = sum(S[bin_4])
    A_5 = sum(S[bin_5])

    # Entering the features into the matrix
    X[i,0] = A_0
    X[i,1] = A_1
    X[i,2] = A_2
    X[i,3] = A_3
    X[i,4] = A_4
    X[i,5] = A_5

    y[i] = 1 # Entering the label into the vector, 1 for "bass"

    i += 1 # Increasing the index
```

2 Plotting for Intuition

[3]:
```
# Plotting the sums of amplitudes in different bins against the labels
fig, ax = plt.subplots(6, figsize=(8,12), constrained_layout=True, sharex=True)
ax[5].set_xlabel("Sum of amplitudes")
for i in np.arange(6):
    ax[i].scatter(X[:,i], y)

    ax[i].set_title(f"Bin {i}")
    labels = [item.get_text() for item in ax[i].get_yticklabels()]
    labelvalues = [item for item in ax[i].get_yticks()]
    labels[labelvalues.index(0)] = 'Guitar'
    labels[labelvalues.index(1)] = 'Bass'
    ax[i].set_yticks(labelvalues)
    ax[i].set_yticklabels(labels)
```


Bin 0
Bass
Guitar
Bin 1
Bass
Guitar
Bin 2
Bass
Guitar
Bin 3
Bass
Guitar
Bin 4
Bass
Guitar
Bin 5
Bass
Guitar
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Sum of amplitudes

3 Applying Machine Learning

[92]:
```
# Separating a third of the features and labels as a test set, which is not␣
↪used for training or validating
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.
↪33, random_state=42)

cv = KFold(n_splits=20, shuffle=True, random_state=42) # Creating a cross␣
↪validator with 20 splits

training_accuracies_logreg = [] # Storing the training accuracies obtained with␣
↪logistic regression
validation_accuracies_logreg = [] # Storing the validation accuracies obtained␣
↪with logistic regression

training_accuracies_dectree = [] # Storing the training accuracies obtained␣
↪with a decision tree
validation_accuracies_dectree = [] # Storing the validation accuracies obtained␣
↪with a decision tree

# Iterating through the indices of the train and validation sets
for train_index, val_index in cv.split(y_train_val):

    X_train, X_val = X_train_val[train_index], X_train_val[val_index]
    y_train, y_val = y_train_val[train_index], y_train_val[val_index]

    # Method 1. Logistic Regression

    # Fitting a classifier to the training set using logistic regression
    clf_logreg = LogisticRegression()
    clf_logreg.fit(X_train, y_train)

    y_pred_train_logreg = clf_logreg.predict(X_train) # Predicting the training␣
↪set labels
    training_accuracies_logreg.append(accuracy_score(y_train,␣
↪y_pred_train_logreg)) # Computing the training accuracy

    y_pred_val_logreg = clf_logreg.predict(X_val) # Predicting the validation␣
↪set labels
    validation_accuracies_logreg.append(accuracy_score(y_val,␣
↪y_pred_val_logreg)) # Computing the validation accuracy

    # Method 2. Decision Tree
```

```
    # Fitting a classifier to the training set using a decision tree

    clf_dectree = DecisionTreeClassifier(criterion="entropy", max_depth=7)
    clf_dectree.fit(X_train, y_train)

    y_pred_train_dectree = clf_dectree.predict(X_train) # Predicting the␣
 ↪training set labels
    training_accuracies_dectree.append(accuracy_score(y_train,␣
 ↪y_pred_train_dectree)) # Computing the training accuracy

    y_pred_val_dectree = clf_dectree.predict(X_val) # Predictiong the␣
 ↪validation set labels
    validation_accuracies_dectree.append(accuracy_score(y_val,␣
 ↪y_pred_val_dectree)) # Computing the validation accuracy



# Print the accuracy values for the training, validation, and test sets

# Logistic regression

print("Logistic regression")

# Average training accuracy
print("Average training accuracy", sum(training_accuracies_logreg)/
 ↪len(training_accuracies_logreg))

# Average validation accuracy
print("Average validation accuracy", sum(validation_accuracies_logreg)/
 ↪len(validation_accuracies_logreg))

# Test accuracy
y_pred_test_logreg = clf_logreg.predict(X_test)
print("Test accuracy", accuracy_score(y_test, y_pred_test_logreg), "\n")


# Decision tree

print("Decision tree")

# Average training accuracy
print("Average training accuracy", sum(training_accuracies_dectree)/
 ↪len(training_accuracies_dectree))

# Average validation accuracy
```

```
print("Average validation accuracy", sum(validation_accuracies_dectree)/
 ↪len(validation_accuracies_dectree))

# Test accuracy
y_pred_test_dectree = clf_dectree.predict(X_test)
print("Test accuracy", accuracy_score(y_test, y_pred_test_dectree))
```

```
Logistic regression
Average training accuracy 0.8832809830306108
Average validation accuracy 0.881859756097561
Test accuracy 0.8686868686868687

Decision tree
Average training accuracy 0.97748099263722
Average validation accuracy 0.9266158536585364
Test accuracy 0.9191919191919192
```