Table of Contents

[Librosa 1](#_Toc130784978)

[Beat Tracking 1](#_Toc130784979)

[Visualizations 3](#_Toc130784980)

[Waveform 3](#_Toc130784981)

[Spectrogram 5](#_Toc130784982)

[Genre Identification 6](#_Toc130784983)

[References 12](#_Toc130784984)

# Librosa

This tutorial will explain how to work with *librosa*, a Python package used for analyzing and processing audio data.

Some *librosa* modules include:

* *librosa.beat*, which includes functions for estimating tempo and detecting beat events
* *librosa.decompose*, which includes functions for decomposing audio samples into their harmonic and percussive sections
* *librosa.display*, which allows us to visualize the audio sample using diagrams such as waveplots and spectrograms

## Beat Tracking

The following tutorial is from the official *librosa* quickstart tutorial: <https://librosa.org/doc/main/tutorial.html>.

The *librosa.beat* module contains functions for “estimating tempo and detecting beat events”. Tempo is the speed or pace at which a musical piece is played, and the ‘beat’ provides its sense of timing and rhythm. First, we import the package and add a path to our audio file. We can install the package using the command *pip install librosa*.

|  |
| --- |
| import librosa filename = 'intheairtonght.wav' |

The variable *filename* now contains a path toPhil Collins’ *In the Air Tonight*. We can then use the *librosa.load()* function to decode the audio file as a time series.

|  |
| --- |
| y, sr = librosa.load(filename) |

The variable *y* contains the time series, and the variable *sr* contains the sampling rate of this time series. As we may recall from COMP 3725, sampling rate is the number of samples taken from an analog signal to convert it into a digital signal. By default, the sampling rate is 22.05 kHz, but can be overridden by passing additional arguments into the *load()* function.

We then pass the time series and sampling rate into the *librosa*.*beat\_track* method, which estimates the audio’s tempo in beats per minute (BPM).

|  |
| --- |
| tempo, beat\_frames = librosa.beat.beat\_track(y=y, sr=sr) |

*librosa* estimates that the tempo for *In the Air Tonight* is 95.70 BPM, which is close to its actual tempo of 94 BPM (<https://getsongbpm.com/song/in-the-air-tonight/0VDrrV>).

We also create a *beat\_frames* variable, which contains an array of frame numbers corresponding to detected ‘beat events’. We can pass these beat frames into the *librosa.frames\_to\_time()* function to receive the timestamps of these ‘beat events’.

|  |
| --- |
| beat\_times = librosa.frames\_to\_time(beat\_frames, sr=sr) |

If we count the number of beat events within one minute (between 0 and 60 seconds), we get 93, which matches *librosa*’s estimated overall tempo.

[ 0.88235828 1.50929705 2.13623583 2.7631746 3.41333333

4.04027211 4.69043084 5.31736961 5.94430839 6.57124717

7.2214059 7.87156463 8.4985034 9.14866213 9.77560091

10.40253968 11.05269841 11.70285714 12.32979592 12.95673469

13.58367347 14.21061224 14.86077098 15.48770975 16.13786848

16.78802721 17.41496599 18.04190476 18.66884354 19.29578231

19.94594104 20.57287982 21.22303855 21.82675737 22.4769161

23.12707483 23.75401361 24.38095238 25.03111111 25.65804989

26.33142857 26.93514739 27.56208617 28.2122449 28.83918367

29.4893424 30.11628118 30.74321995 31.39337868 32.02031746

32.64725624 33.29741497 33.9475737 34.57451247 35.20145125

35.82839002 36.50176871 37.10548753 37.75564626 38.38258503

39.03274376 39.65968254 40.30984127 40.93678005 41.56371882

42.1906576 42.81759637 43.4677551 44.11791383 44.74485261

45.37179138 46.02195011 46.64888889 47.27582766 47.94920635

48.55292517 49.2030839 49.83002268 50.48018141 51.10712018

51.75727891 52.38421769 53.01115646 53.63809524 54.26503401

54.89197279 55.54213152 56.16907029 56.84244898 57.4461678

58.09632653 58.72326531 59.37342404 ]

We can further separate the audio time series into harmonic and percussive portions of the audio signal using *librosa.effects.hpss()*. As the percussion usually sets the rhythm for a piece of music, we can use this isolated time series to estimate a more accurate tempo.

|  |
| --- |
| y\_harmonic, y\_percussive = librosa.effects.hpss(y) |

The estimated tempo for *y\_percussive* is also 95.70 BPM, which is okay since the original estimate already matches the actual tempo. However, separating the audio signal may be necessary to capture the tempo for pieces of music with more asynchronous harmonies and percussion.

Here is the full program:

|  |
| --- |
| *# Beat tracking example* import librosa  *# 1. Get the file path to an included audio example* filename = 'intheairtonight.wav'  *# 2. Load the audio as a waveform `y` # Store the sampling rate as `sr`* y, sr = librosa.load(filename)  *# 3. Run the default beat tracker* tempo, beat\_frames = librosa.beat.beat\_track(y=y, sr=sr)  print('Estimated tempo: {:.2f} beats per minute'.format(tempo))  *# 4. Convert the frame indices of beat events into timestamps* beat\_times = librosa.frames\_to\_time(beat\_frames, sr=sr)  print(beat\_times)  *# 5. Separate harmonics and percussives into two waveforms* y\_harmonic, y\_percussive = librosa.effects.hpss(y) tempo, beat\_frames = librosa.beat.beat\_track(y=y\_percussive,  sr=sr)  print('Estimated percussive tempo: {:.2f} beats per minute'.format(tempo)) |

## Visualizations

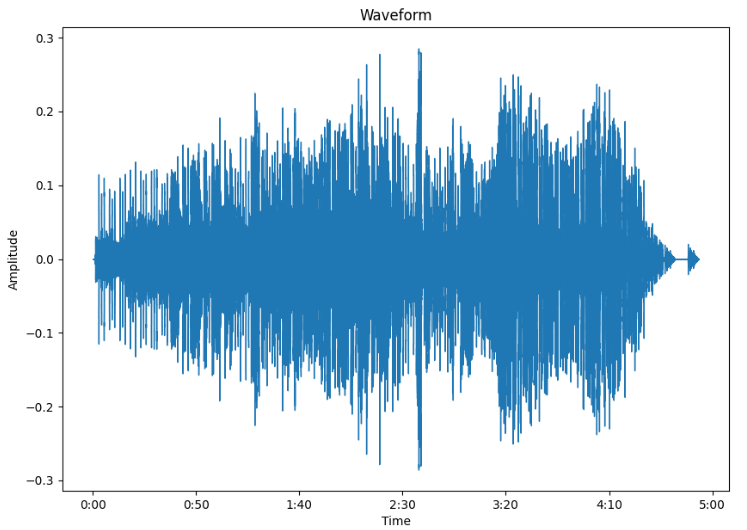
The *librosa.display* module offers functions which can create visualizations of an audio sample.

### Waveform

A waveform diagram shows how the amplitude – which corresponds to volume in decibels (dB) – of an audio signal changes over time. We can combine *librosa.display.waveshow* with *matplotlib.pyplot* to display a waveform diagram for a given audio sample.

Here is the code:

|  |
| --- |
| import librosa import librosa.display as dsp import matplotlib.pyplot as plt import numpy as np  time\_series, sample\_rate = librosa.load('intheairtonight.wav')  *# Waveform* fig, ax = plt.subplots(nrows=1, sharex=True,figsize=(10,7)) dsp.waveshow(time\_series, sr=sample\_rate, ax=ax) ax.set(title='Waveform') ax.label\_outer() plt.ylabel("Amplitude") plt.show() |



We can see that the average amplitude gets louder after the 3:20 minute-mark, which corresponds to the song’s famous breakdown. We also see that the song fades out during the end, which is typical of an 80’s song. The ending also repeats after some silence, which the YouTube video this file was converted from does for some reason.

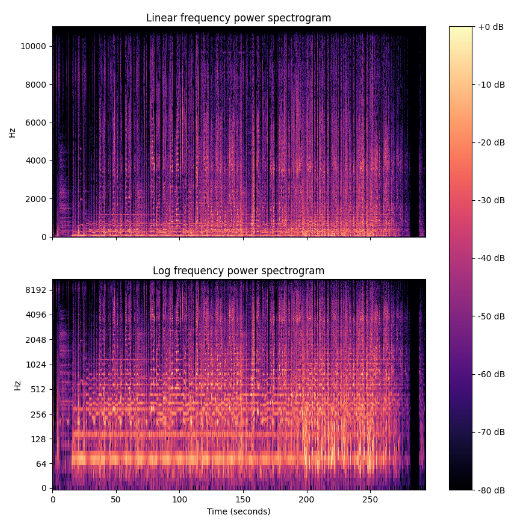
### Spectrogram

A spectrogram is a three-dimensional plot that visualizes the audio sample’s frequencies – which corresponds to pitch – over time. It also shows amplitude using colour.

We first use **Short-time Fourier Transform (STFT)** to decompose the audio sample into its frequency components. We then convert it from a linear scale to a logarithmic decibel scale using *librosa.amplitude\_to\_db()*. We can then show the spectrogram using *librosa.specshow()*.

Add the following code to the snippet above:

|  |
| --- |
| *# Spectrogram* d = librosa.stft(time\_series) D = librosa.amplitude\_to\_db(np.abs(d),ref=np.max) fig, ax = plt.subplots(2,1,sharex=True,figsize=(10,10)) img = dsp.specshow(D, y\_axis='linear', x\_axis='s',sr=sample\_rate,ax=ax[0]) ax[0].set(title='Linear frequency power spectrogram') ax[0].label\_outer() dsp.specshow(D,y\_axis='log',x\_axis='s',sr=sample\_rate,ax=ax[1]) ax[1].set(title='Log frequency power spectrogram') ax[1].label\_outer() fig.colorbar(img, ax=ax, format='%+2.f dB') plt.show() |



## Genre Identification

Now, let’s apply *librosa* for some sweet machine learning fun! We can apply various *librosa* functions to predict the musical genre of a given audio sample. The tutorial I will be using was created by Marc Saint-Félix, and can be found here: <https://towardsdatascience.com/music-genre-detection-with-deep-learning-cf89e4cb2ecc>.

To train our model, we’ll be using the GTZAN Dataset, a music genre classification dataset which contains 100 samples each of 10 different musical genres. It can be downloaded here: <https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification?select=Data>. Since each sample is over 30 seconds in length, we can crop them all to be 29 seconds. We can also generate more than the 999 samples in the dataset (1 was defective) by slicing each song into sub-parts. This will help to train our model using more information.

|  |
| --- |
| *# Sampling rate.* sr = 22050*# Let’s make sure all files have the same amount of samples, pick a duration right under 30 seconds.* TOTAL\_SAMPLES = 29 \* sr*# The dataset contains 999 files (1000–1 defective). Let’s make it bigger.*  *# X amount of slices => X times more training examples.* NUM\_SLICES = 10 SAMPLES\_PER\_SLICE = int(TOTAL\_SAMPLES / NUM\_SLICES) |

Since the audio files are already organized into folders named by their genre, we can easily classify each audio sample by converting their label into a number (e.g., 0 for ‘rock’, 1 for ‘pop’, etc.).

The information used to make predictions will be each audio file’s **Mel-frequency cepstral coefficients (MFCCs)**, which can be acquired by using the *librosa.feature.mfcc* module. MFCC is a technique used to extract feature information from audio files by transforming the sample to more closely mimic a human’s perception of sound. For example, one step of MFCC is to filter the audio logarithmically, since humans hear loudness on a logarithmic rather than linear scale (e.g., to double the perceived volume of sound, we need to use around 10 times as much energy).

We will then store this information in a .json file to speed up the process on subsequent runs.

|  |
| --- |
| *# Testing the model on never seen before data.* make\_prediction(model, Xtest, ytest, 24)  *# Let's browse each file, slice it and generate the 13 band mfcc for each slice.* for i, (dirpath, dirnames, filenames) in enumerate(os.walk(source\_path)):   for file in filenames:  song, sr = librosa.load(os.path.join(dirpath, file), duration=29)   for s in range(NUM\_SLICES):  start\_sample = SAMPLES\_PER\_SLICE \* s  end\_sample = start\_sample + SAMPLES\_PER\_SLICE  mfcc = librosa.feature.mfcc(y=song[start\_sample:end\_sample], sr=sr, n\_mfcc=13)  mfcc = mfcc.T  mydict["labels"].append(i - 1)  mydict["mfcc"].append(mfcc.tolist())  *# Let's write the dictionary in a json file.* with open(json\_path, 'w') as f:  json.dump(mydict, f)  f.close() |

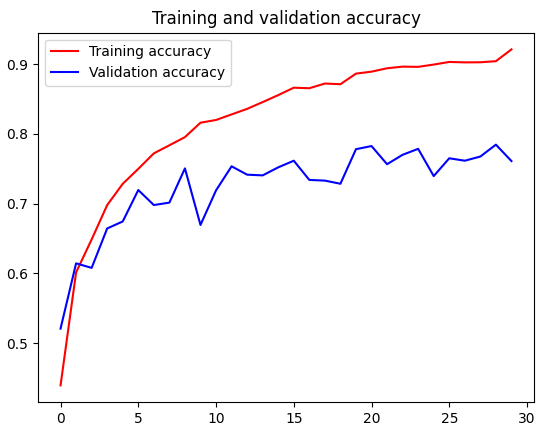
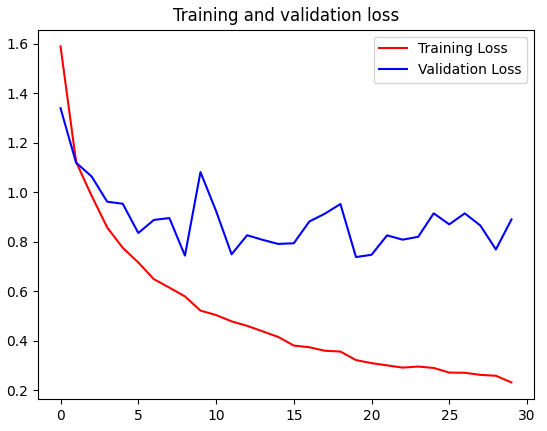
Finally, we need to train and predict using our model. First, we split the data into training, testing, and validation inputs.

|  |
| --- |
| def prepare\_datasets(inputs, targets, split\_size):  *# Creating a validation set and a test set.* inputs\_train, inputs\_val, targets\_train, targets\_val = train\_test\_split(inputs, targets, test\_size=split\_size)  inputs\_train, inputs\_test, targets\_train, targets\_test = train\_test\_split(inputs\_train, targets\_train,  test\_size=split\_size)   *# Our CNN model expects 3D input shape.* inputs\_train = inputs\_train[..., np.newaxis]  inputs\_val = inputs\_val[..., np.newaxis]  inputs\_test = inputs\_test[..., np.newaxis]   return inputs\_train, inputs\_val, inputs\_test, targets\_train, targets\_val, targets\_test |

We then create a 3-layer **Convolutional Neural Network (CNN)** using *keras*. The first layer is a 2D convolutional layer that uses the Rectified Linear Unit (ReLU) activation function, the second layer is a max pooling layer, and the third layer normalizes the activations of the previous layer.

|  |
| --- |
| def design\_model(input\_shape):  *# Let's design the model architecture.* model = tf.keras.models.Sequential([   tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input\_shape=input\_shape),  tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same'),  tf.keras.layers.BatchNormalization(),   tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),  tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same'),  tf.keras.layers.BatchNormalization(),   tf.keras.layers.Conv2D(32, (2, 2), activation='relu'),  tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same'),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.Dropout(0.3),   tf.keras.layers.Flatten(),  tf.keras.layers.Dense(64, activation='relu'),  tf.keras.layers.Dense(len(np.unique(targets)), activation='softmax')  ])   return model |

We can then compile, fit, and predict using the model. In the end, our model will have a validation accuracy somewhere around 75%.

Using a random file from our test split, we can see that the model successfully predicts its musical genre:



Here is the full program:

|  |
| --- |
| import os import json import numpy as np import librosa import matplotlib.pyplot as plt from sklearn.model\_selection import train\_test\_split import tensorflow as tf  *# Dataset location* SOURCE\_PATH = 'Dataset/genres\_original/'  *# Path to labels and processed data file, json format.* JSON\_PATH = 'data.json'  *# Sampling rate.* sr = 22050  *# Let's make sure all files have the same amount of samples, pick a duration right under 30 seconds.* TOTAL\_SAMPLES = 29 \* sr  *# The dataset contains 999 files. Lets make it bigger. # X amount of slices => X times more training examples.* NUM\_SLICES = 10 SAMPLES\_PER\_SLICE = int(TOTAL\_SAMPLES / NUM\_SLICES)   def preprocess\_data(source\_path, json\_path):  *# Let's create a dictionary of labels and processed data.* mydict = {  "labels": [],  "mfcc": []  }   *# Let's browse each file, slice it and generate the 13 band mfcc for each slice.* for i, (dirpath, dirnames, filenames) in enumerate(os.walk(source\_path)):   for file in filenames:  song, sr = librosa.load(os.path.join(dirpath, file), duration=29)   for s in range(NUM\_SLICES):  start\_sample = SAMPLES\_PER\_SLICE \* s  end\_sample = start\_sample + SAMPLES\_PER\_SLICE  mfcc = librosa.feature.mfcc(y=song[start\_sample:end\_sample], sr=sr, n\_mfcc=13)  mfcc = mfcc.T  mydict["labels"].append(i - 1)  mydict["mfcc"].append(mfcc.tolist())   *# Let's write the dictionary in a json file.* with open(json\_path, 'w') as f:  json.dump(mydict, f)  f.close()   def load\_data(json\_path):  with open(json\_path, 'r') as f:  data = json.load(f)  f.close()   *# Let's load our data into numpy arrays for TensorFlow compatibility.* X = np.array(data["mfcc"])  y = np.array(data["labels"])  print(X.shape)   return X, y   def prepare\_datasets(inputs, targets, split\_size):  *# Creating a validation set and a test set.* inputs\_train, inputs\_val, targets\_train, targets\_val = train\_test\_split(inputs, targets, test\_size=split\_size)  inputs\_train, inputs\_test, targets\_train, targets\_test = train\_test\_split(inputs\_train, targets\_train,  test\_size=split\_size)   *# Our CNN model expects 3D input shape.* inputs\_train = inputs\_train[..., np.newaxis]  inputs\_val = inputs\_val[..., np.newaxis]  inputs\_test = inputs\_test[..., np.newaxis]   return inputs\_train, inputs\_val, inputs\_test, targets\_train, targets\_val, targets\_test   def design\_model(input\_shape):  *# Let's design the model architecture.* model = tf.keras.models.Sequential([   tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input\_shape=input\_shape),  tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same'),  tf.keras.layers.BatchNormalization(),   tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),  tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same'),  tf.keras.layers.BatchNormalization(),   tf.keras.layers.Conv2D(32, (2, 2), activation='relu'),  tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same'),  tf.keras.layers.BatchNormalization(),  tf.keras.layers.Dropout(0.3),   tf.keras.layers.Flatten(),  tf.keras.layers.Dense(64, activation='relu'),  tf.keras.layers.Dense(len(np.unique(targets)), activation='softmax')  ])   return model   def make\_prediction(model, X, y, idx):  genre\_dict = {  0: "blues",  1: "classical",  2: "country",  3: "disco",  4: "hiphop",  5: "jazz",  6: "metal",  7: "pop",  8: "reggae",  9: "rock",  }   predictions = model.predict(X)  genre = np.argmax(predictions[idx])   print("\n---Now testing the model for one audio file---\nThe model predicts: {}, and it's actually: {}.\n".format(  genre\_dict[genre], genre\_dict[y[idx]]))   def plot\_performance(hist):  acc = hist.history['acc']  val\_acc = hist.history['val\_acc']  loss = hist.history['loss']  val\_loss = hist.history['val\_loss']   epochs = range(len(acc))   plt.plot(epochs, acc, 'r', label='Training accuracy')  plt.plot(epochs, val\_acc, 'b', label='Validation accuracy')  plt.title('Training and validation accuracy')  plt.legend()  plt.figure()   plt.plot(epochs, loss, 'r', label='Training Loss')  plt.plot(epochs, val\_loss, 'b', label='Validation Loss')  plt.title('Training and validation loss')  plt.legend()   plt.show()   if \_\_name\_\_ == "\_\_main\_\_":  preprocess\_data(source\_path=SOURCE\_PATH, json\_path=JSON\_PATH)   inputs, targets = load\_data(json\_path=JSON\_PATH)   Xtrain, Xval, Xtest, ytrain, yval, ytest = prepare\_datasets(inputs, targets, 0.2)   input\_shape = (Xtrain.shape[1], Xtrain.shape[2], 1)  model = design\_model(input\_shape)   *# Selection of the optimizer, loss type and metrics for performance evaluation.* model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=0.001),  loss='sparse\_categorical\_crossentropy',  metrics=['acc']  )   model.summary()   *# Training the model.* history = model.fit(Xtrain,  ytrain,  validation\_data=(Xval, yval),  epochs=30,  batch\_size=32)   plot\_performance(history) |

# References

Lendave, V. (2021, July 29). *Hands-on guide to Librosa for Handling Audio files*. Analytics

India Magazine. Retrieved March 27, 2023, from https://analyticsindiamag.com/hands-on-

guide-to-librosa-for-handling-audio-files/

McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and

Oriol Nieto. “librosa: Audio and music signal analysis in python.” In Proceedings of the

14th python in science conference, pp. 18-25. 2015.

Saint-Félix, M. (2021, August 11). *Music genre detection with Deep Learning*. Medium.

Retrieved March 27, 2023, from https://towardsdatascience.com/music-genre-detection-

with-deep-learning-cf89e4cb2ecc