Nadim Kawwa

Follow 205 Followers About

Can We Guess Musical Instruments With Machine Learning?

Nadim Kawwa Apr 27, 2019 · 8 min read

Music is pervasive in our daily lives, present across all cultures and constantly evolving. One might argue that the ability to interact with music is an exclusively human quality. Streaming giants such as Spotify and Apple (just to name a few) curate their content and offer personalized recommendations for their users. To do so they possess powerful tools that classify entries based on genre, instruments, and lyrics.

There is a real need for these businesses to get their predictions correct and without depleting their resources. Some sources suggest that Spotify adds 10,000 to 30,000 songs a day.

We are going to explore how we can classify musical instruments using machine learning algorithms written in python. We want to predict the instrument with the highest accuracy possible, but we also want to get there without too much effort, stopping when we reach a point of diminishing returns.

The idea is far from novel, there are several papers published on this topic and on even more advanced subjects such as music generation with AI. How this post differs is a focus on the following criteria businesses use to determine a winner among the algorithms:

- Accuracy
- Training Time
- Efficiency
- Reliability

The Dataset

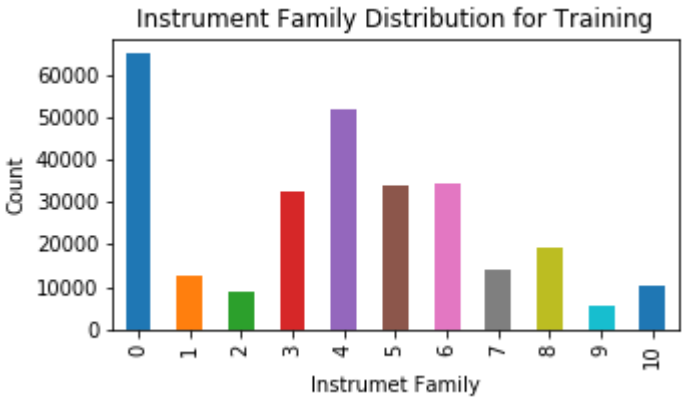
NSynth is a large-scale and high-quality dataset of annotated musical notes, and free to use courtesy of Google Inc. There are no missing values, no duplicate entries, and each observation is correctly labeled. This takes away the need to perform feats of data cleaning.

It is also massive: more than 300,000 four second, audio snippets stored as wave files (.wav). About 95% of the data is for training, 4% for validation, and a measly 1% for testing.

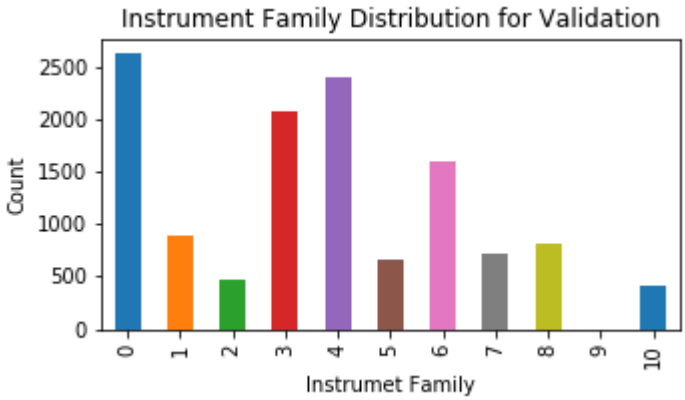
In total we have 11 instruments, labeled from 0 to 1, but as we find out later we can reduce that count to 10:

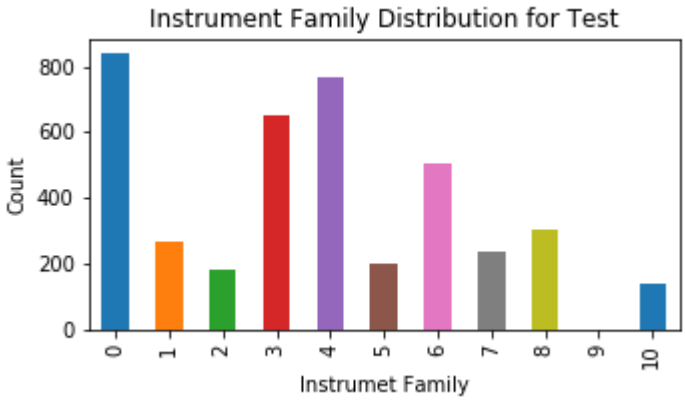
- bass
- brass
- flute
- guitar
- keyboard
- mallet
- organ
- reed
- string
- synth_lead
- vocal

In addition it is an imbalanced dataset, which will be an additional challenge in the road ahead. The plot below shows the distribution of instrument classes in the test dataset, notice we have over 60k bass files and barely 5k string instruments.



We have similar issues with the validation and testing data.





Notice that instrument #9 in the two plots above. Does it make sense to train for something we probably won't see? Instrument #9 is also underrepresented in the testing data. We will drop it from the testing data to simplify our analysis.

Understanding the Data: Anatomy of a Wave File

In order to classify the instruments we must first understand what are the characteristics of wave files, we shall refer to these characteristics as the *feature space*.

For dissecting wave files we rely heavily on Librosa, a python package for music and audio analysis. We randomly select wave files from the data and explore the features of the sample. Loading a file is very easy in Librosa:

```
#import the package
import librosa

#define the file path
bass_file= 'audio/bass_electronic_018-047-075.wav'

#load the waveform y and sampling rate s
```

```
y, sr = librosa.load(bass_file, sr=None)
# To preserve the native sampling rate of the file, use sr=None
```

librosa_waveform.py hosted with ♡ by GitHub view raw

We define a sound as vibration traveling through the air. Each sound has a frequency defined as the number of occurrences of a repeating event per unit of time and measured in Hertz (Hz) or inverse seconds (s-1). With that in mind we can get our hands dirty with the data and explore some features.

Waveform and Sampling Rate

A sound is a continuous time signal that is usually represented by a waveform; this continuous signal is sampled and converted into a discrete time signal.

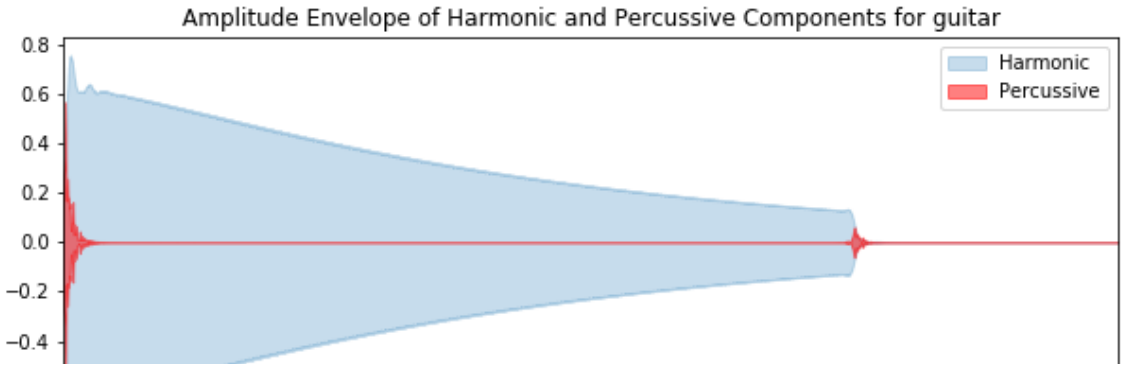
Sampling rate is the number of samples of audio recorded per second.The sampling rate determines the maximum audio frequency that can be reproduced.

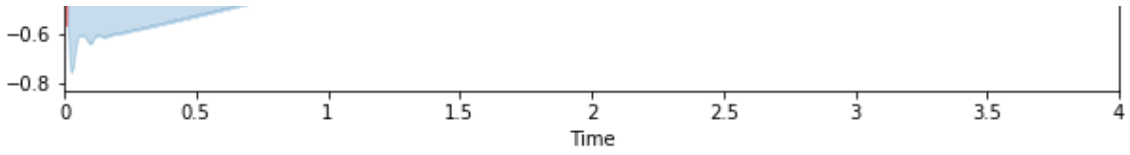
A harmonic sound is one where series of sounds within which the fundamental frequency of each of them is an integral multiple of the lowest fundamental frequency. A percussion instrument is any object you can bang on.

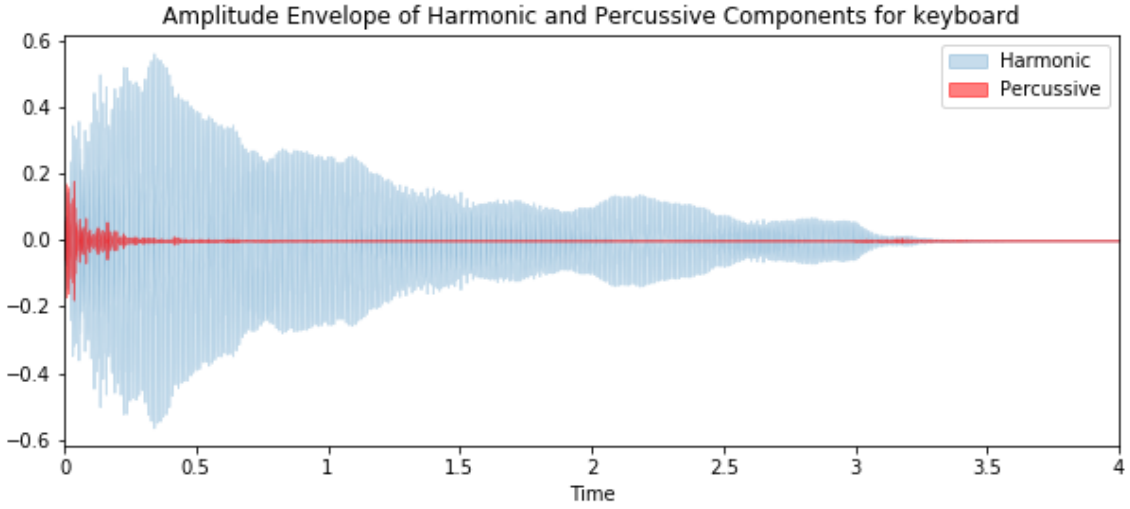
We can decompose wave files into harmonic and percussive components with a single line of code:

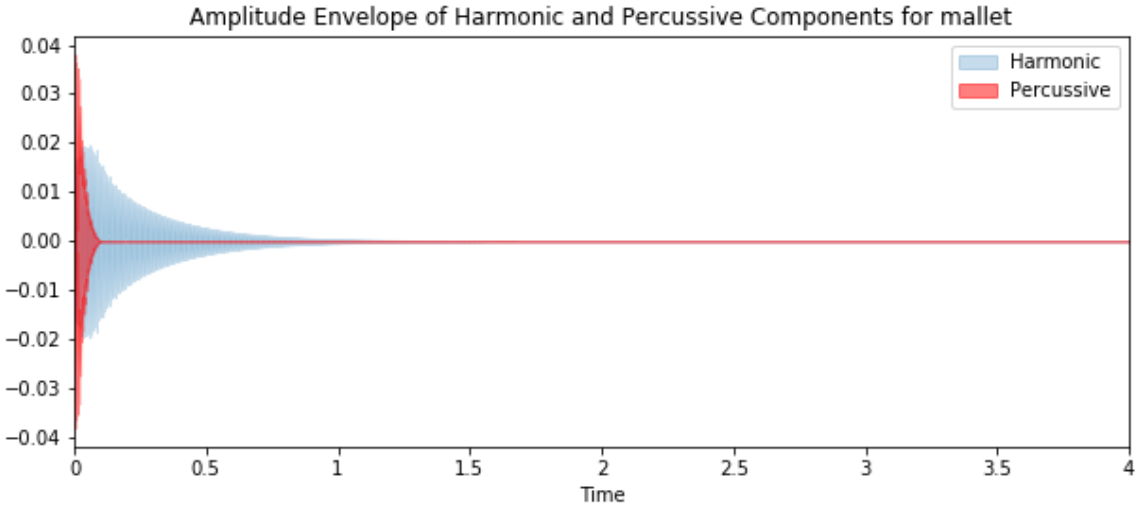
```
y_harmonic, y_percussive = librosa.effects.hpss(bass_file)
```

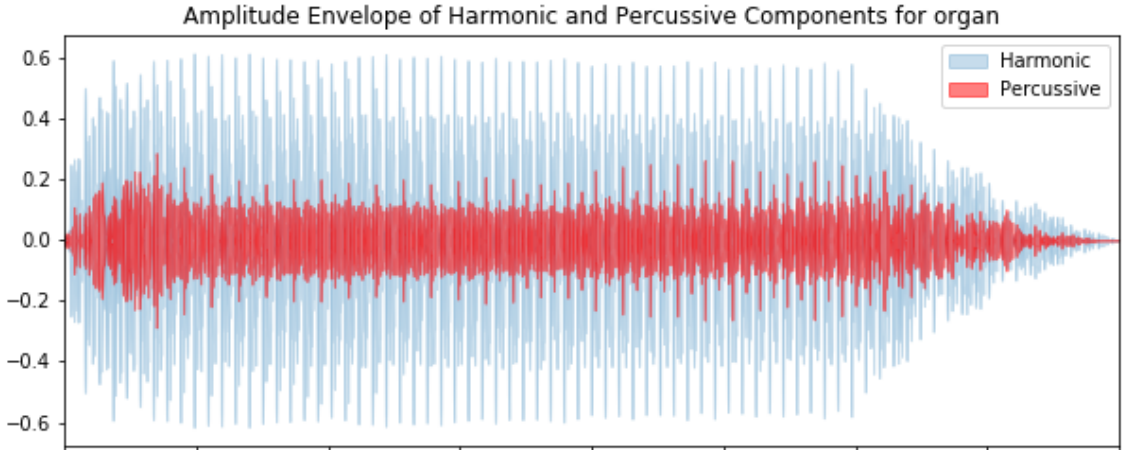
From the plots below, we can begin to tell instruments apart and see the percussive and harmonic qualities.




−0.6
−0.8
0
0.5
1
1.5
2
2.5
3
3.5
4
Time


Amplitude Envelope of Harmonic and Percussive Components for keyboard
Harmonic
Percussive
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
0.5
1
1.5
2
2.5
3
3.5
4
Time


Amplitude Envelope of Harmonic and Percussive Components for mallet
Harmonic
Percussive
0.04
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
0
0.5
1
1.5
2
2.5
3
3.5
4
Time

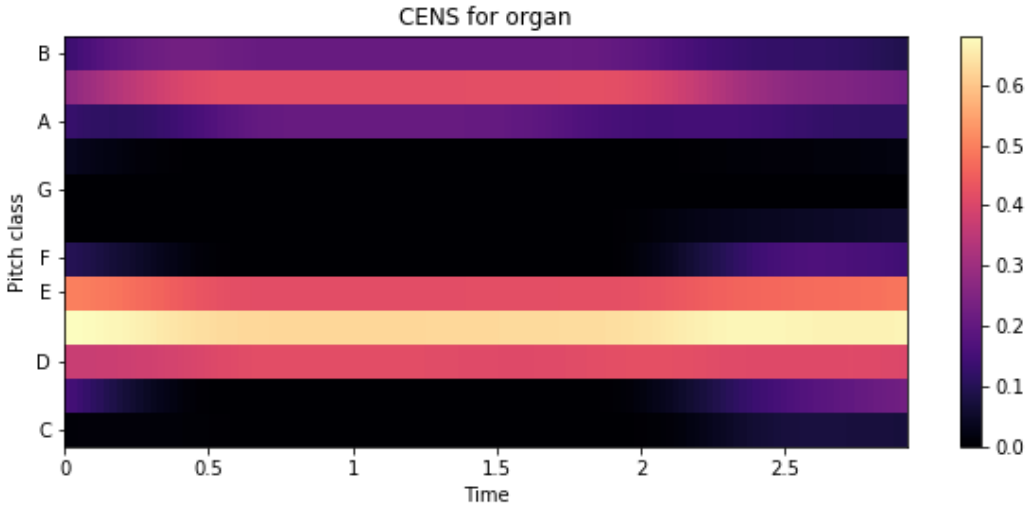

Amplitude Envelope of Harmonic and Percussive Components for organ
Harmonic
Percussive
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

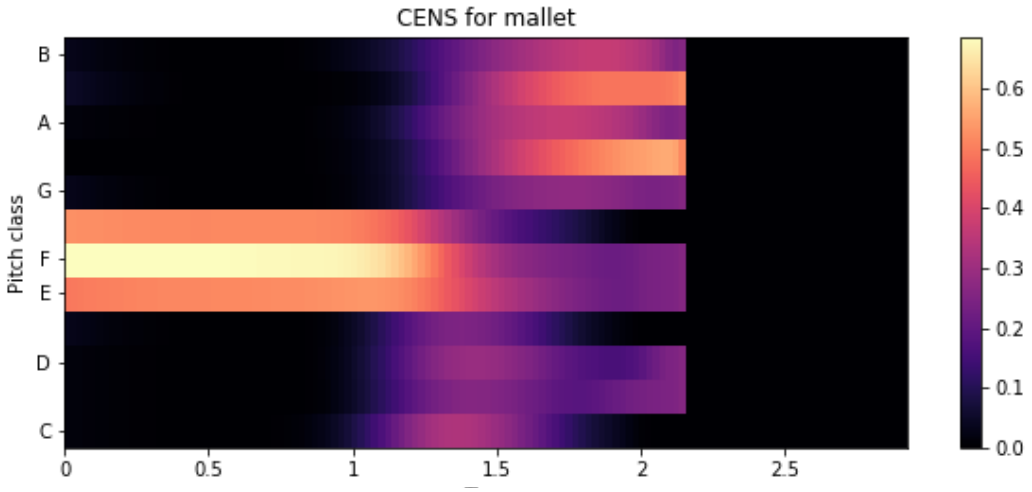
Chroma Energy

A pitch is the quality of a sound governed by the rates of vibrations producing it. Two pitches are perceived as similar in "color" if they differ by an octave. Based on this observation, a pitch can be separated into two components, which are referred to as tone height and chroma.

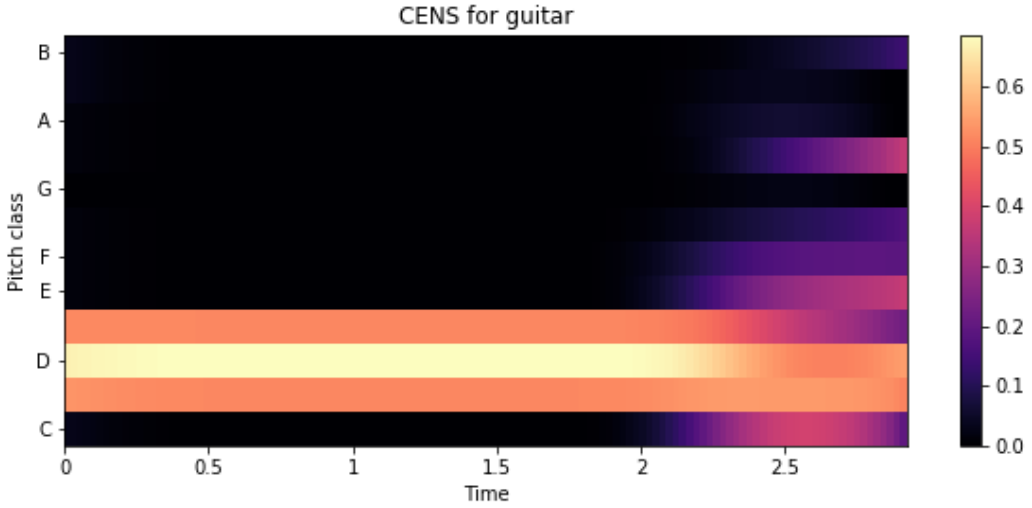
Assuming an equal-tempered scale, the chromas correspond to the set {C, C♯, D, . . . , B} that consists of the twelve pitch spelling attribute as used in Western music notation.

The plots below display the chromas by color bands. Each color bands corresponds to the notes that the instrument hits. We also notice that some notes are more activated then others and some decay suddenly.





Time



Mel Spectrogram

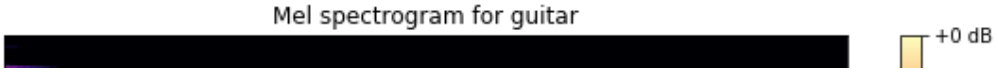
A spectrogram is a basic tool in audio spectral analysis and other fields. It has been applied extensively in speech analysis. The spectrogram can be defined as an intensity plot (usually on a log scale, such as dB) of the Short-Time Fourier Transform (STFT) magnitude.

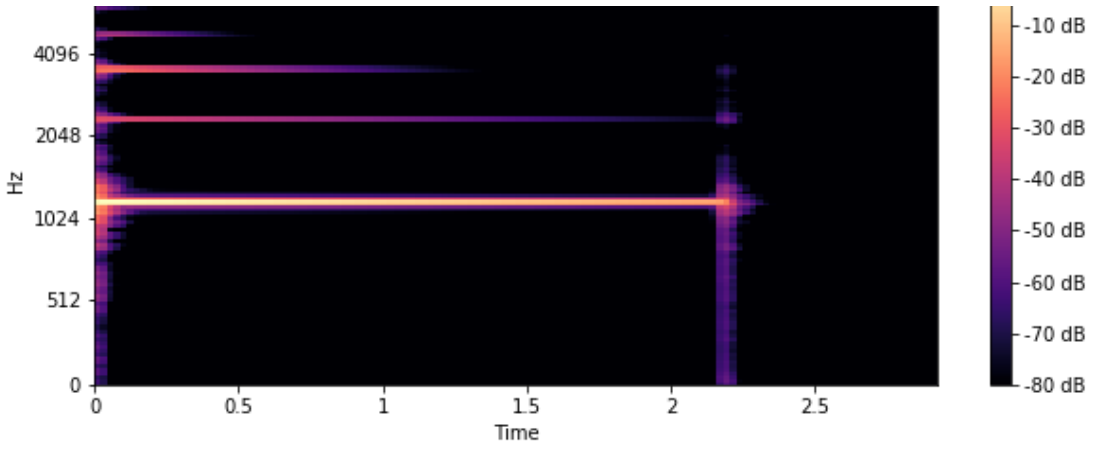
The STFT is simply a sequence of Fast Fourier Transforms (FFT) of windowed data segments, where the windows are usually allowed to overlap in time, typically by 25–50%. It is an important representation of audio data because human hearing is based on a kind of real-time spectrogram encoded by the cochlea of the inner ear. We can predict that the spectrogram will probably play a major role in prediction.

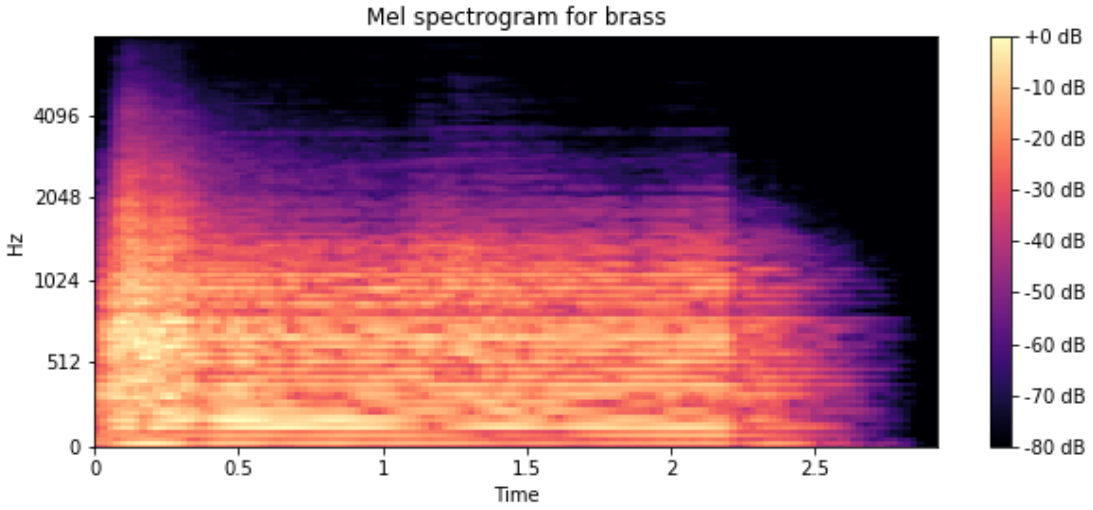
The mel is a unit of pitch. The mel scale is a scale of pitches judged by listeners to be equal in distance one from another. We note that the mel scale is based on empirical evidence and has no theoretical basis.

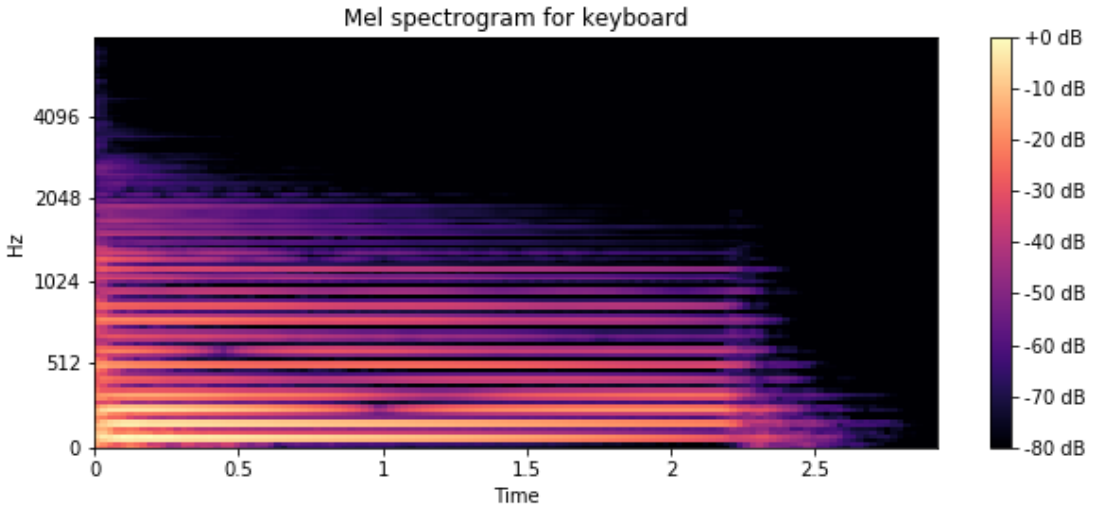
Librosa is again handy here and outputs the spectrogram with a single line of code:

```
librosa.feature.melscpetrogram(y, sr, n_mels=128, fmax=8000)
```









Additional Features

There are other features that were evaluated such as spectral rolloff frequency, spectral contrast, and mel-frequency cepstrum coefficients (MFCC). However we won't dive

deeper into exploring them.

Data Preparation: Wrangling

We now have an idea of what makes a wave file. With no categorical variables to encode, we want to extract the following features:

- Is the instrument harmonic?
- MFCC
- Spectrogram
- Spectral contrast

The first hurdle is the data because there's a lot of it, maybe too much. Let's make a simplification right off the bat and draw only 5000 samples from each class. That way we go from 290,000 samples to just 50,000. Keep the objective in mind:

Make as many reasonable simplifications as possible, save up on time, and still reach reasonable results.

We can execute these steps in the function below:

```
import numpy as np
import librosa

def feature_extract(file):
    """
    Define function that takes in a file an returns features in an array
    """

    #get wave representation
    y, sr = librosa.load(file)

    #determine if instruemnt is harmonic or percussive by comparing means
    y_harmonic, y_percussive = librosa.effects.hpss(y)
    if np.mean(y_harmonic)>np.mean(y_percussive):
        harmonic=1
    else:
        harmonic=0

```

```
    #Mel-frequency cepstral coefficients (MFCCs)
    mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
    #temporal averaging
    mfcc=np.mean(mfcc,axis=1)

    #get the mel-scaled spectrogram
    spectrogram = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128,fmax=8000)
    #temporally average spectrogram
    spectrogram = np.mean(spectrogram, axis = 1)

    #compute chroma energy
    chroma = librosa.feature.chroma_cens(y=y, sr=sr)
    #temporally average chroma
    chroma = np.mean(chroma, axis = 1)

    #compute spectral contrast
    contrast = librosa.feature.spectral_contrast(y=y, sr=sr)
    contrast = np.mean(contrast, axis= 1)

    return [harmonic, mfcc, spectrogram, chroma, contrast]
```

feature_extract.py hosted with ♡ by GitHub view raw

There's a lot going in the block of code above, and it might seem daunting. However the key takeaway is the following expression:

```
spectrogram = np.mean(spectrogram, axis=1)
```

Recall that the spectrogram shows the activated frequencies for each time frame. How do we fit an *m x n* array into a table? The solution is temporally averaging the spectrogram: At each time step we take the average of the frequencies.

Prediction Accuracy

Naive Bayes

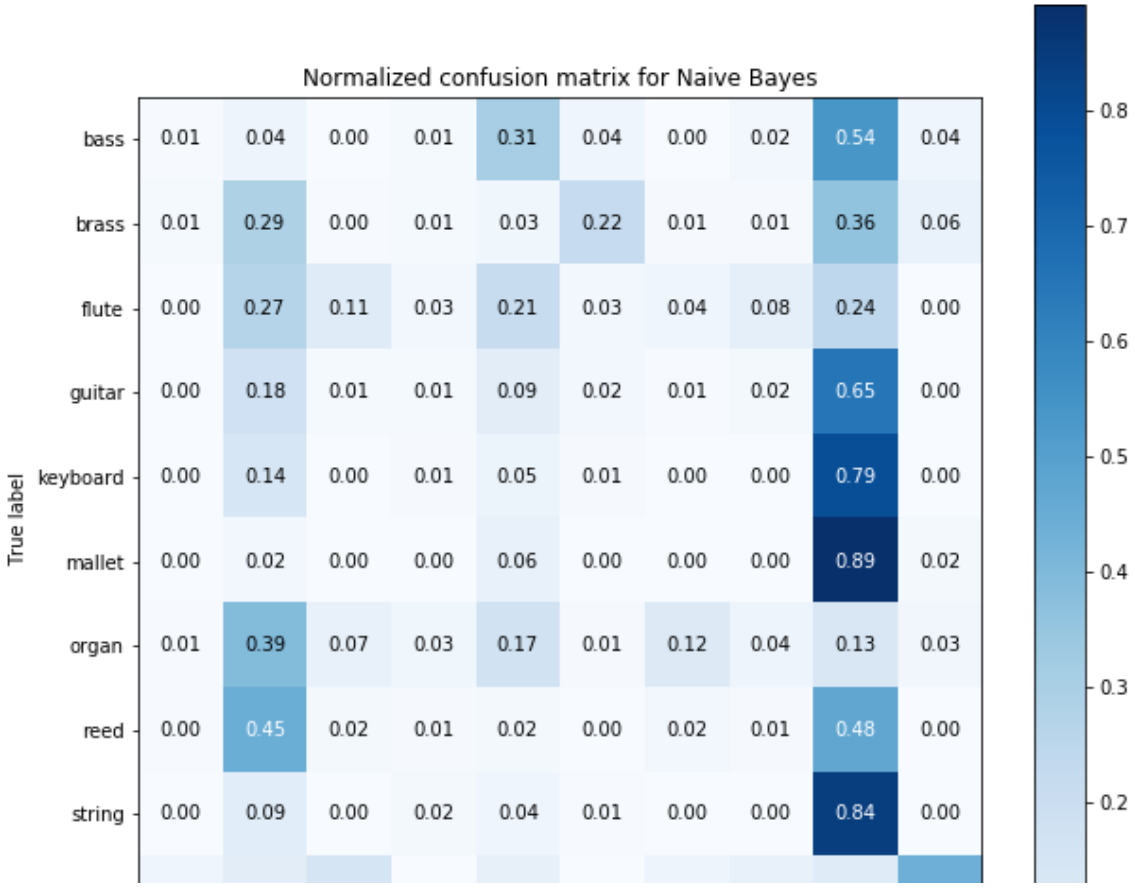
The first runner up for prediction is Naive Bayes (NB). Scikit-learn makes it incredibly easy to set up the prediction regardless of the algorithm:

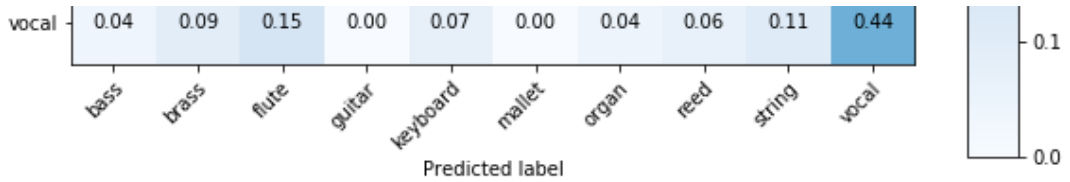
```
#imports
from sklearn.naive_bayes import GaussianNB
```

```
import numpy as np

#instantiate the classifier
clf_NB = GaussianNB()

#fit to training data
clf_NB.fit(X_train, y_train)

#make a prediction
y_pred_NB = clf_NB.predict(X_test)

#report the percentage of true predictions
accuracy_NB = np.mean(y_pred_NB == y_test)
```

NSynth_NB.py hosted with ♡ by GitHub view raw

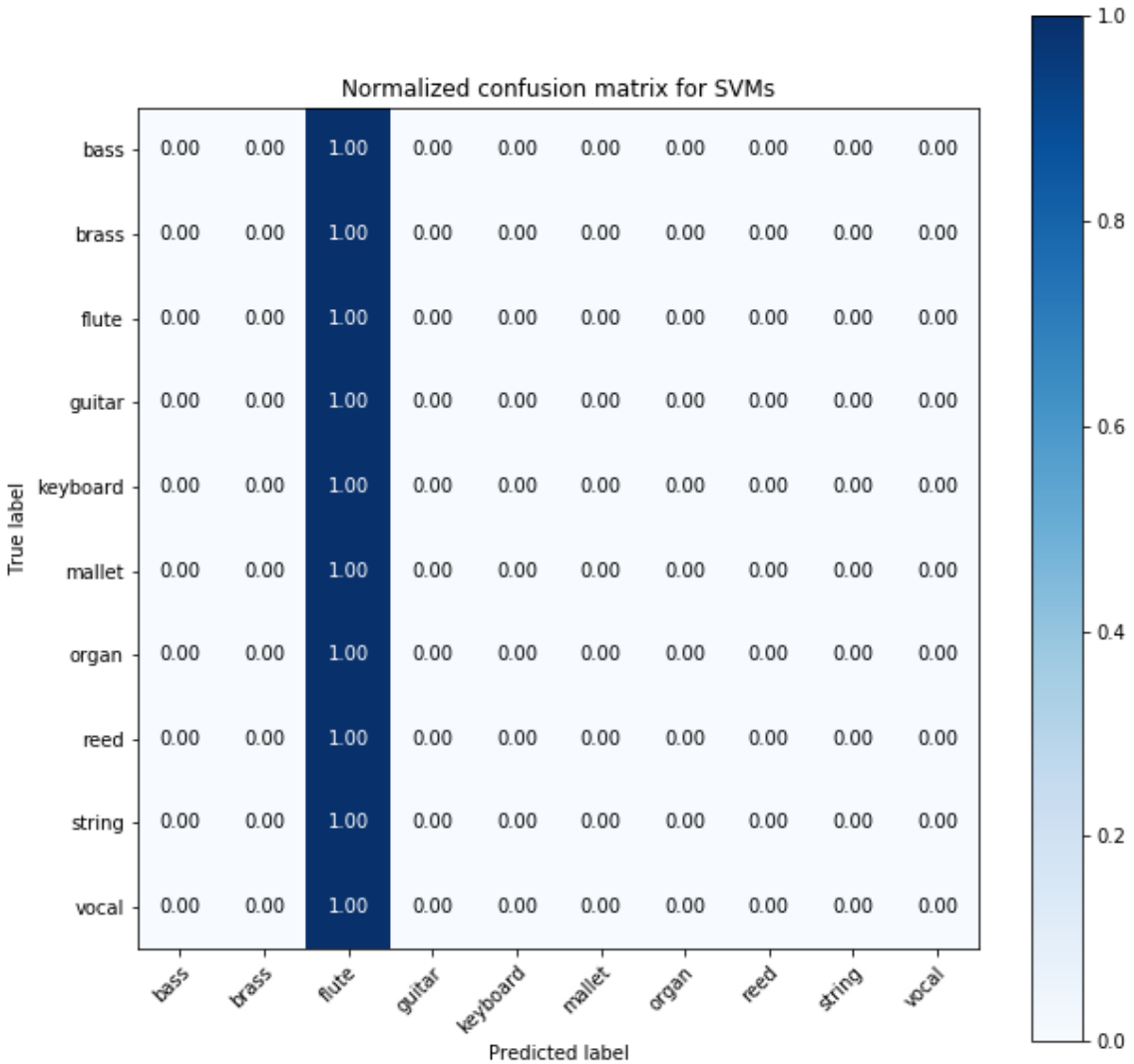
The results are presented as a confusion matrix. The higher the values in the diagonal the better our prediction. NB has an accuracy of **13%**, slightly better than a random guess.





Support Vector Machines

Support vector machines are a popular supervised learning algorithm. However from the way our features and parameters are set up, we predict everything to be a flute (fail).



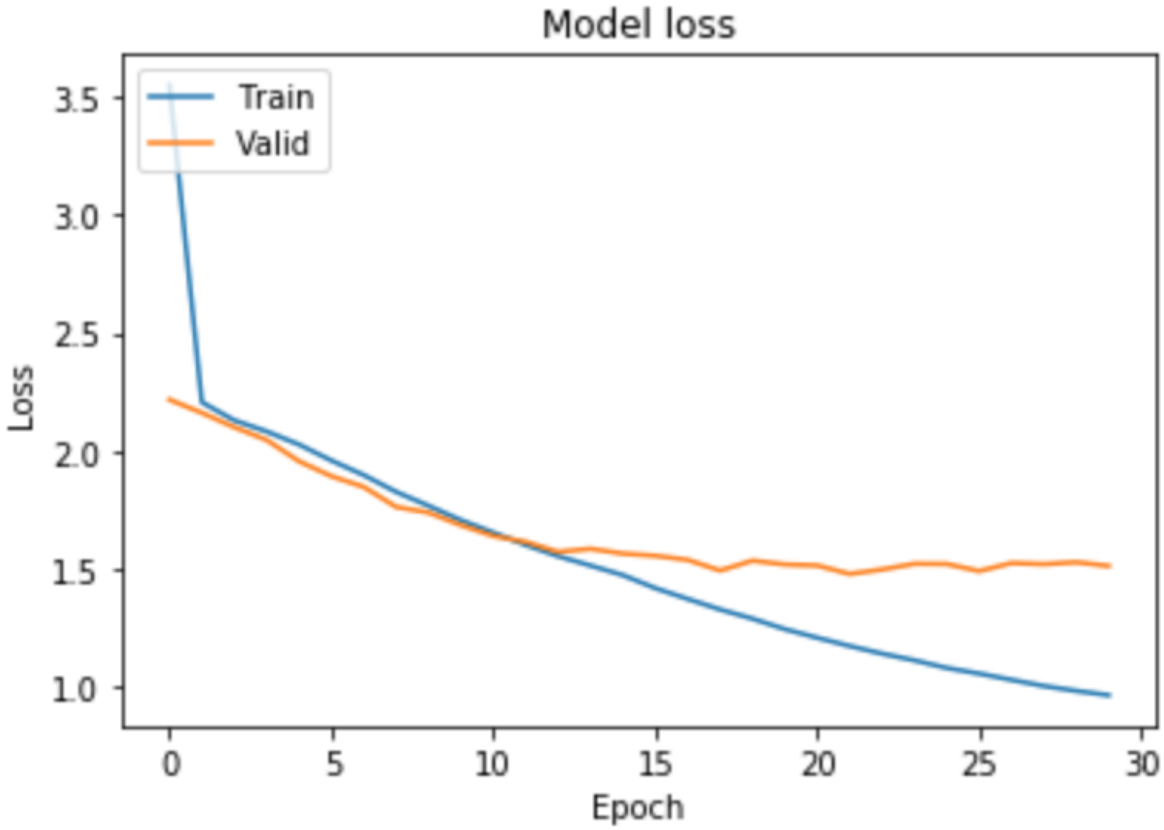
Random Forests

Random decision forests (RF) are an ensemble learning method and often the winners of machine learning competitions. We obtain **65%** accuracy after fine tuning some parameters and clearly outperform the other two.

Neural Networks

Neural nets are often seen as the holy grail of AI with much deserved hype; for this project we use convolutional neural networks (CNN). With Keras we can easily set up our own CNN in the style of CIFAR-10.

The accuracy ebbs at **55%** and after certain training epochs we stop making gains. Considering the plot below, starting at the 15th training epoch we can see diminishing returns for the validation data.



Training Time and Efficiency

We now have four competing algorithms, so which one is the most efficient? We can start by discarding NB and SVMs because of low accuracy. This leaves us with RF and CNN.

The CNN took about 15 hours of training on an AWS GPU instance. Training on the cloud is not free and can become very expensive, limiting the amount of time we can spend on the task.

On the other hand RF trains in less than 3 hours, just make sure you keep your machine plugged to a power outlet. In addition, the time complexity of RF is O(f*s*log(f)) where f is the number of features and s is the number of samples.

Reliability

By now it's clear that RF is the winner across all three metrics. However how reliable are our results? Can we expect a consistent performance? We perform a sensitivity test and use another batch of the training data of equal size and Lo and behold! accuracy is now **85%!**

The vast difference in results suggests that our model can't be expected to provide consistent results. There are many more parameters that might affect RF performance such as the maximum depth of a tree and the number of estimators. One idea to get a real feel of reliability is bootstrapping: randomly sampling the data with replacement, and report the accuracy of each batch.

Conclusion

We found that we can indeed classify musical instruments without much computational effort and the results far outperform random guessing. In addition, the fanciest algorithm is not always the best performer.

This post is based on a much larger project found on my GitHub profile and comes with an in depth report that explains the machine learning process step by step.

Machine Learning Data Science Music Neural Networks Python