

# Classification and feature engineering

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON



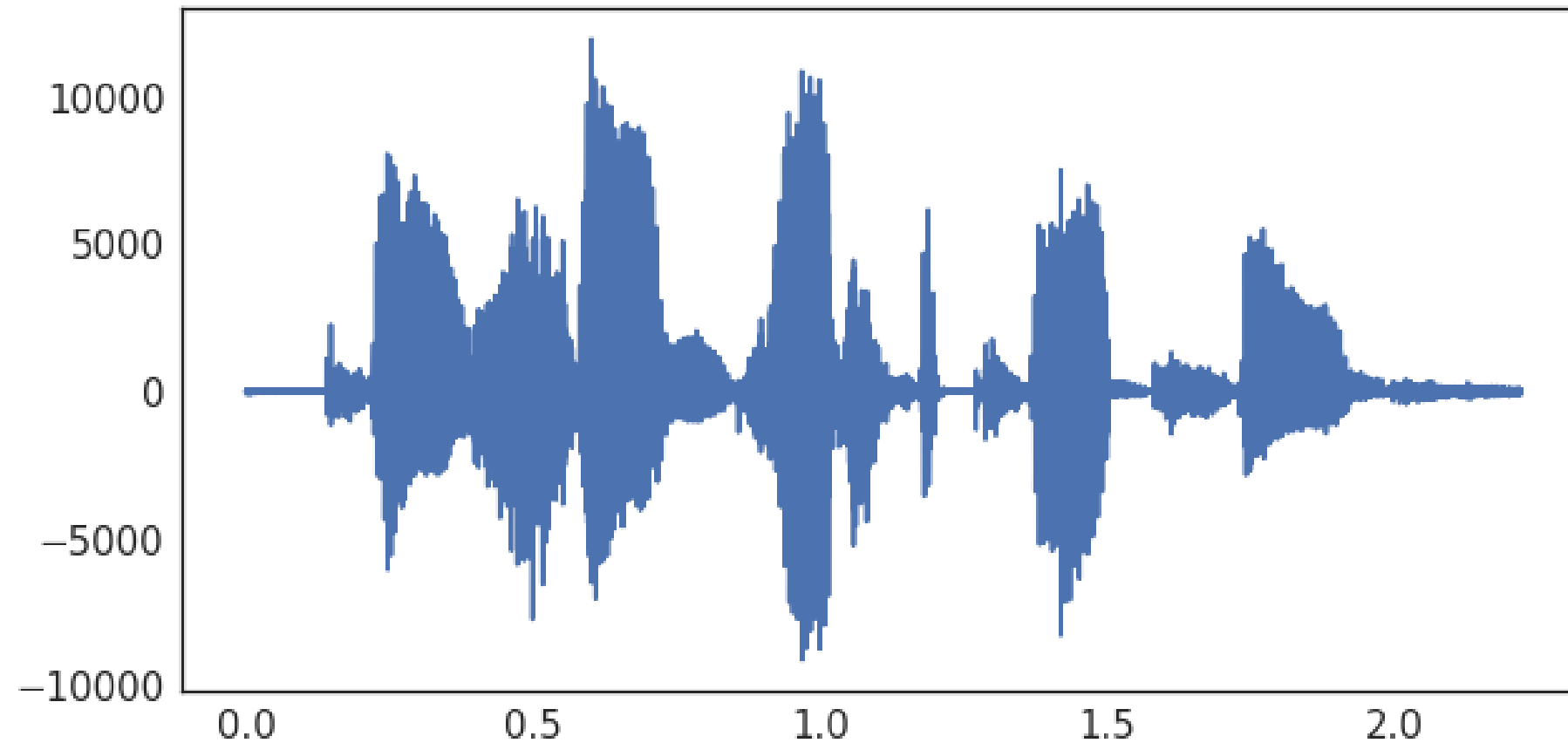
**Chris Holdgraf**

Fellow, Berkeley Institute for Data  
Science

# Always visualize raw data before fitting models

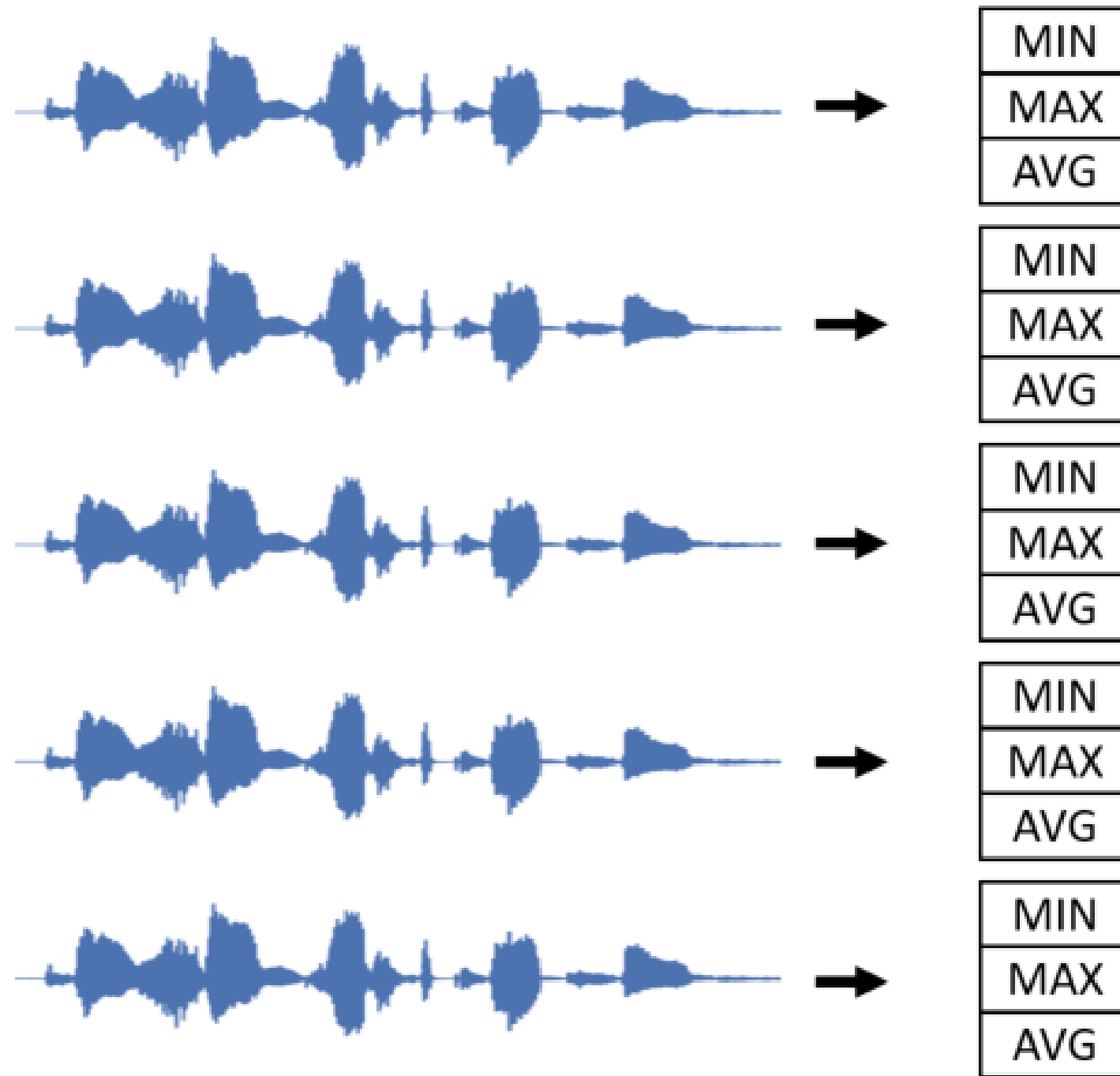
# Visualize your timeseries data!

```
ixs = np.arange(audio.shape[-1])  
time = ixs / sfreq  
fig, ax = plt.subplots()  
ax.plot(time, audio)
```



# What features to use?

- Using raw timeseries data is too noisy for classification
- We need to calculate features!
- An easy start: summarize your audio data



# Calculating multiple features

```
print(audio.shape)  
# (n_files, time)
```

```
(20, 7000)
```

```
means = np.mean(audio, axis=-1)  
maxs = np.max(audio, axis=-1)  
stds = np.std(audio, axis=-1)  
  
print(means.shape)  
# (n_files,)
```

```
(20,)
```

# Fitting a classifier with scikit-learn

- We've just collapsed a 2-D dataset (samples x time) into several features of a 1-D dataset (samples)
- We can combine each feature, and use it as an input to a model
- If we have a label for each sample, we can use scikit-learn to create and fit a classifier

# Preparing your features for scikit-learn

```
# Import a linear classifier
from sklearn.svm import LinearSVC

# Note that means are reshaped to work with scikit-learn
X = np.column_stack([means, maxs, stds])
y = labels.reshape([-1, 1])
model = LinearSVC()
model.fit(X, y)
```



# Scoring your scikit-learn model

```
from sklearn.metrics import accuracy_score

# Different input data
predictions = model.predict(X_test)

# Score our model with % correct
# Manually
percent_score = sum(predictions == labels_test) / len(labels_test)
# Using a sklearn scorer
percent_score = accuracy_score(labels_test, predictions)
```

# Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

# Improving the features we use for classification

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

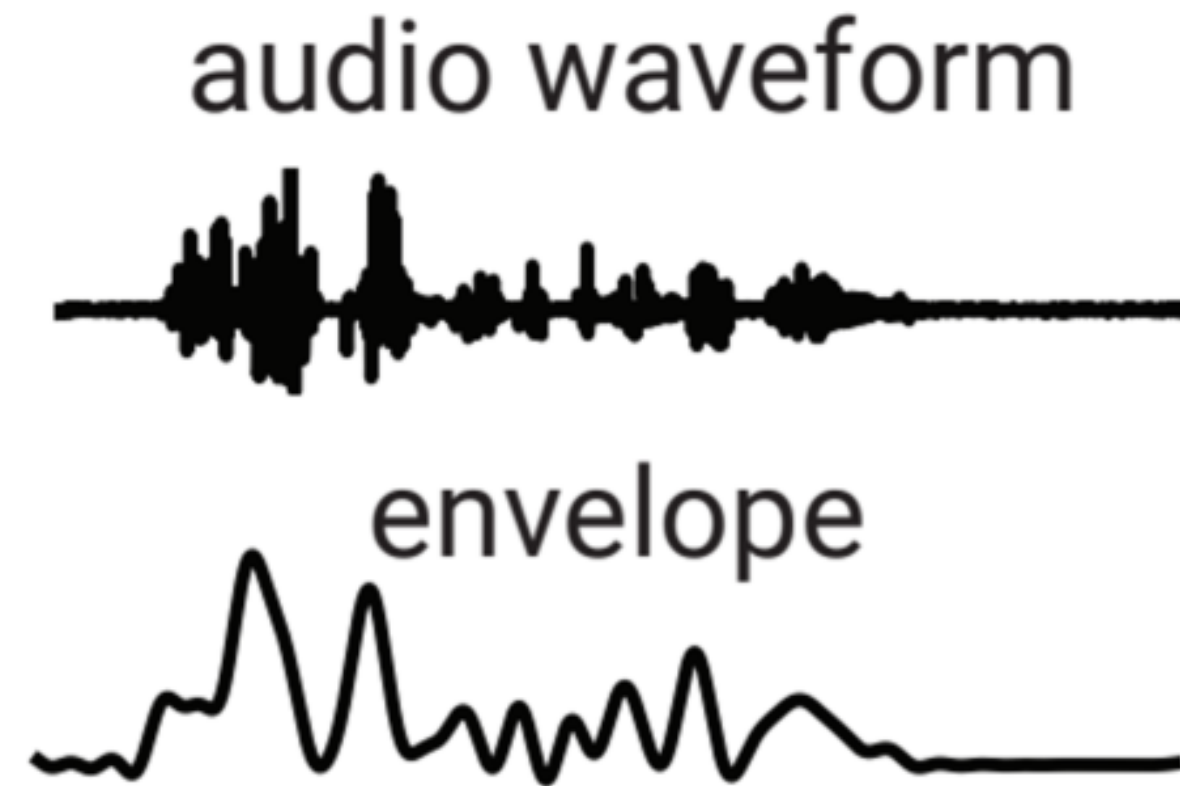


**Chris Holdgraf**

Fellow, Berkeley Institute for Data  
Science

# The auditory envelope

- Smooth the data to calculate the auditory envelope
- Related to the total amount of audio energy present at each moment of time

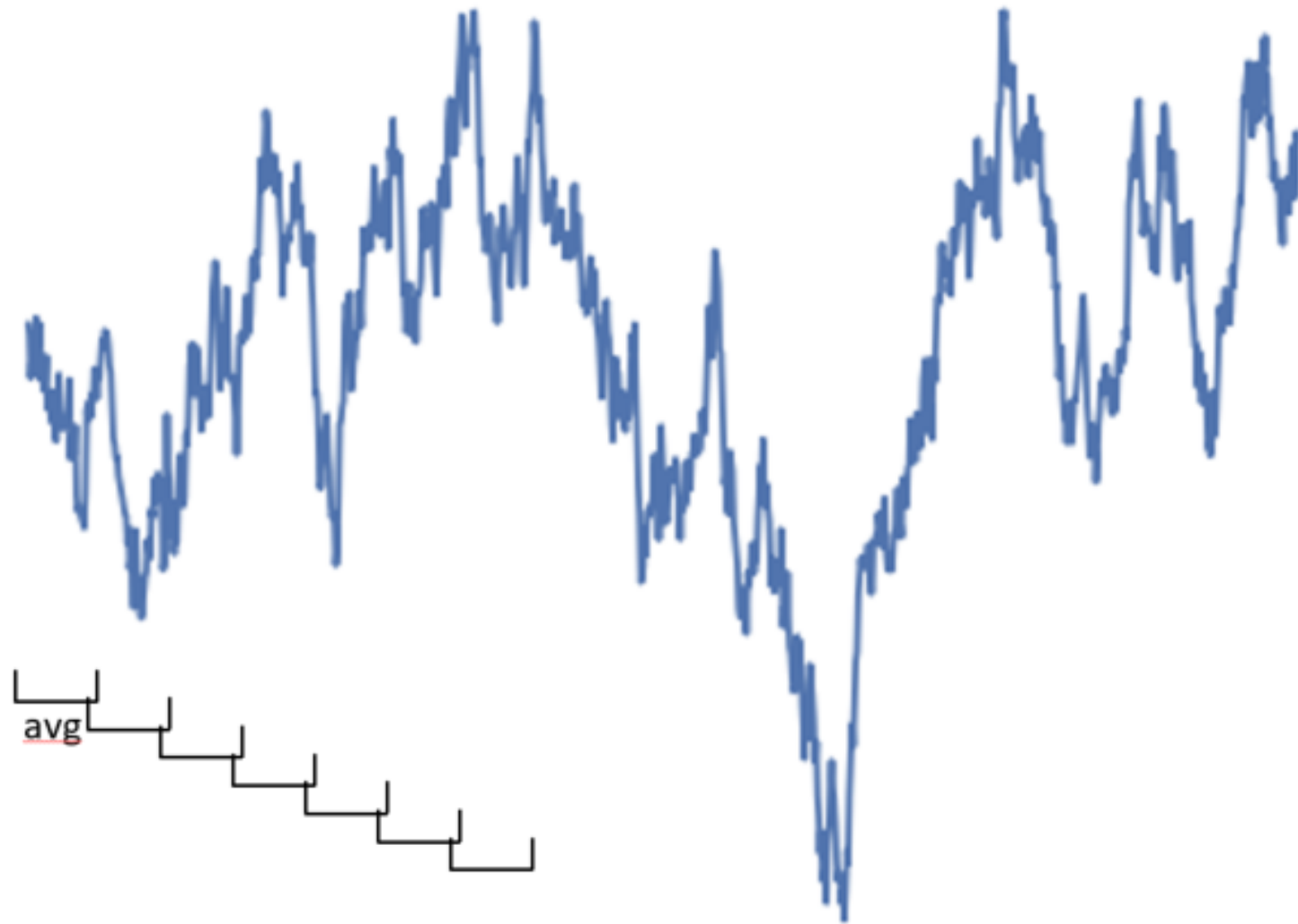


# Smoothing over time

- Instead of averaging over *all* time, we can do a *local* average
- This is called *smoothing* your timeseries
- It removes short-term noise, while retaining the general pattern

# Smoothing your data

Noisy data



Smoothed with rolling window



# Calculating a rolling window statistic

```
# Audio is a Pandas DataFrame
print(audio.shape)
# (n_times, n_audio_files)
```

```
(5000, 20)
```

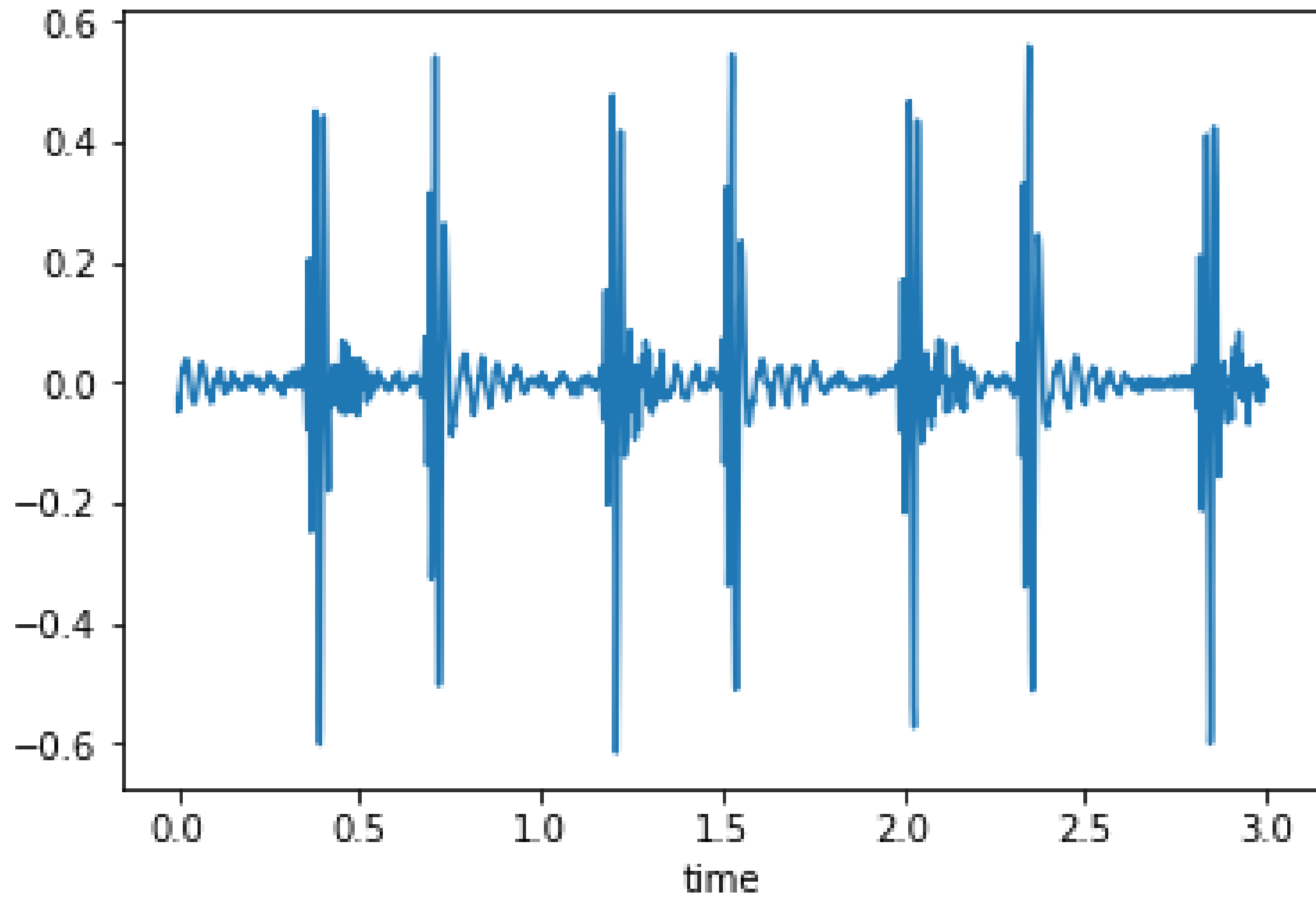
```
# Smooth our data by taking the rolling mean in a window of 50 samples
window_size = 50
windowed = audio.rolling(window=window_size)
audio_smooth = windowed.mean()
```

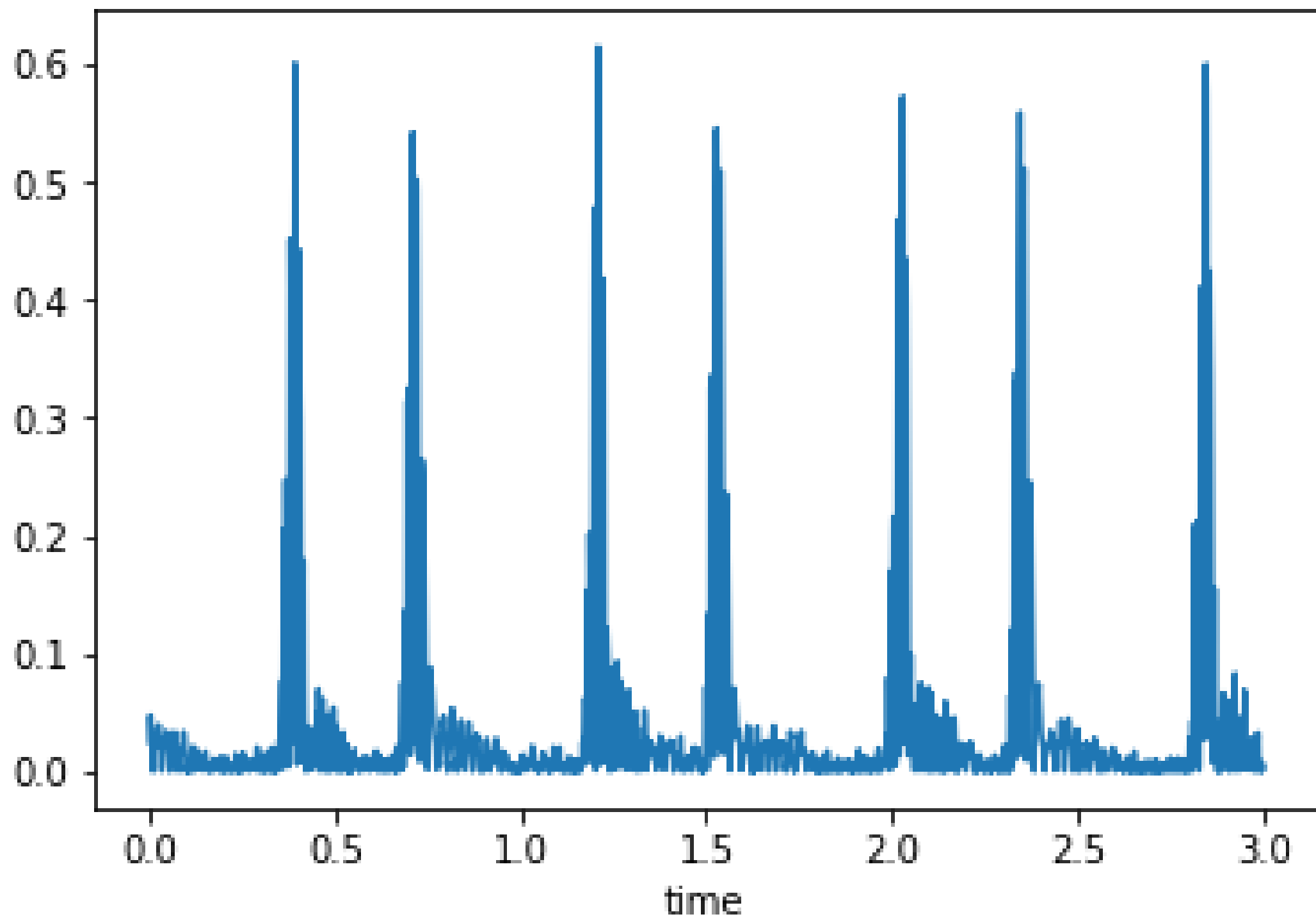
# Calculating the auditory envelope

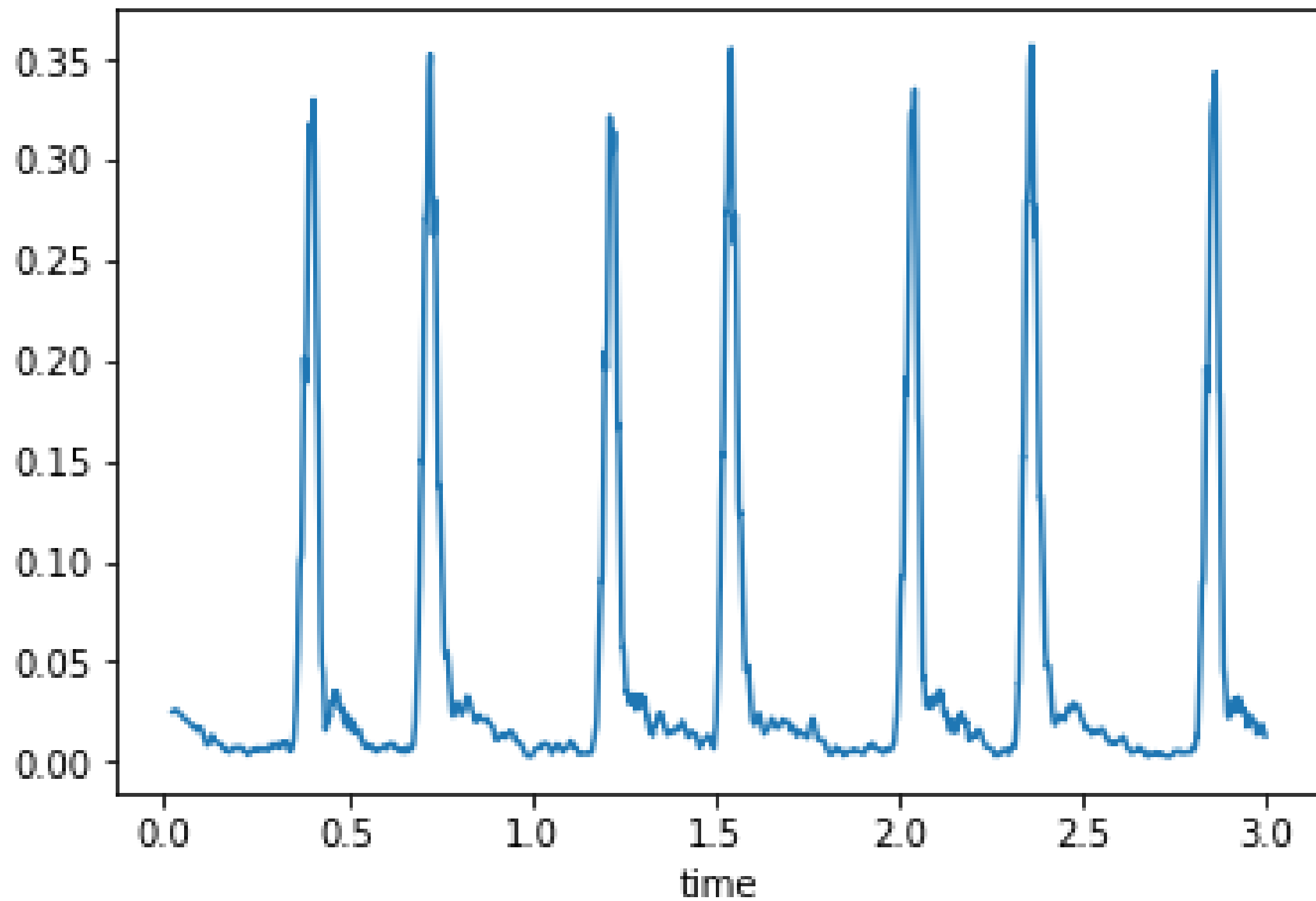
- First *rectify* your audio, then smooth it

```
audio_rectified = audio.apply(np.abs)
audio_envelope = audio_rectified.rolling(50).mean()
```









# Feature engineering the envelope

```
# Calculate several features of the envelope, one per sound
envelope_mean = np.mean(audio_envelope, axis=0)
envelope_std = np.std(audio_envelope, axis=0)
envelope_max = np.max(audio_envelope, axis=0)

# Create our training data for a classifier
X = np.column_stack([envelope_mean, envelope_std, envelope_max])
```

# Preparing our features for scikit-learn

```
X = np.column_stack([envelope_mean, envelope_std, envelope_max])  
y = labels.reshape([-1, 1])
```

# Cross validation for classification

- `cross_val_score` automates the process of:
  - Splitting data into training / validation sets
  - Fitting the model on training data
  - Scoring it on validation data
  - Repeating this process

# Using cross\_val\_score

```
from sklearn.model_selection import cross_val_score

model = LinearSVC()
scores = cross_val_score(model, X, y, cv=3)
print(scores)
```

```
[0.60911642 0.59975305 0.61404035]
```

# Auditory features: The Tempogram

- We can summarize more complex temporal information with timeseries-specific functions
- `librosa` is a great library for auditory and timeseries feature engineering
- Here we'll calculate the *tempogram*, which estimates the tempo of a sound over time
- We can calculate summary statistics of tempo in the same way that we can for the envelope



# Computing the tempogram

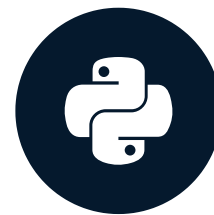
```
# Import librosa and calculate the tempo of a 1-D sound array
import librosa as lr
audio_tempo = lr.beat.tempo(audio, sr=sfreq,
                           hop_length=2**6, aggregate=None)
```

# Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

# The spectrogram - spectral changes to sound over time

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON



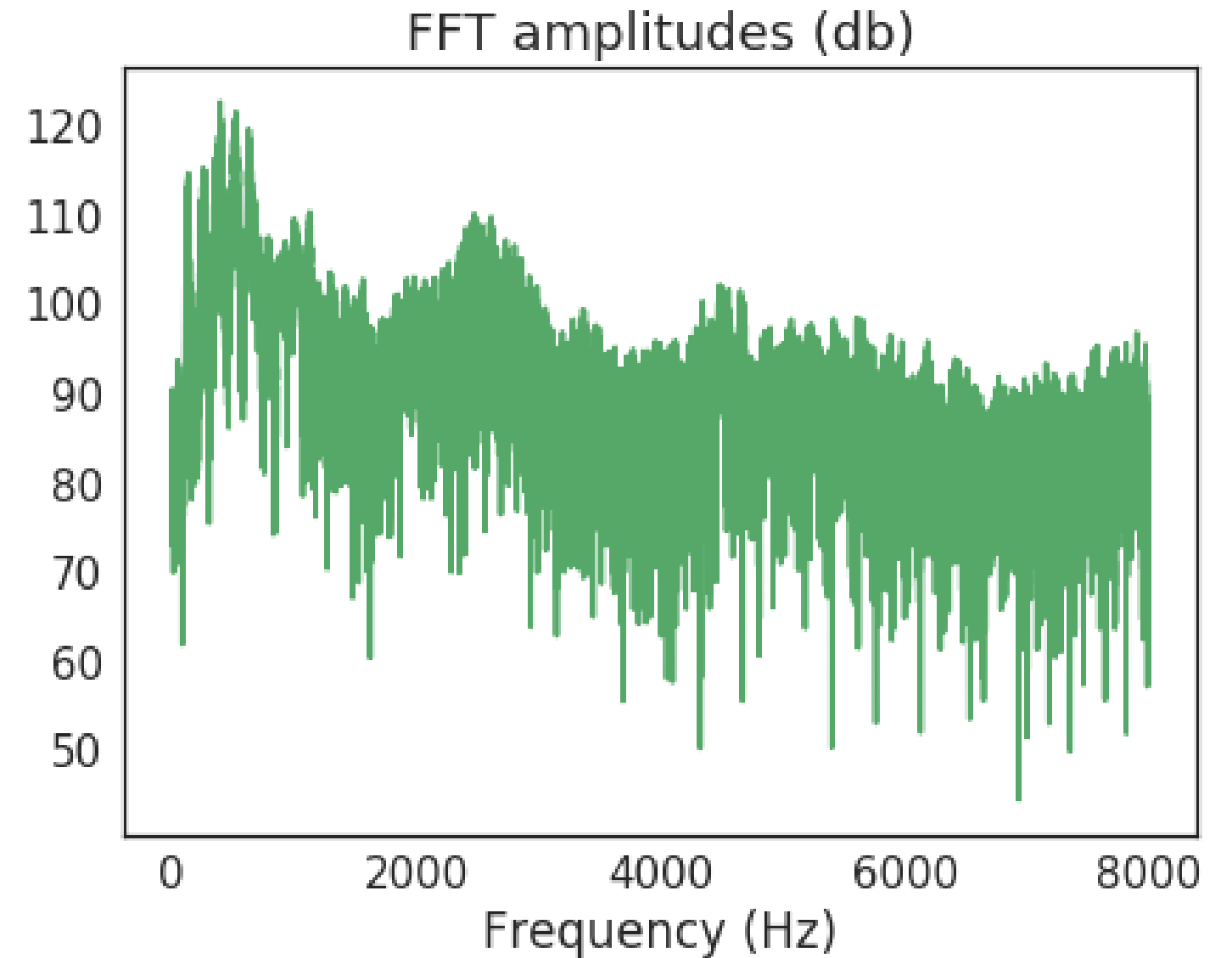
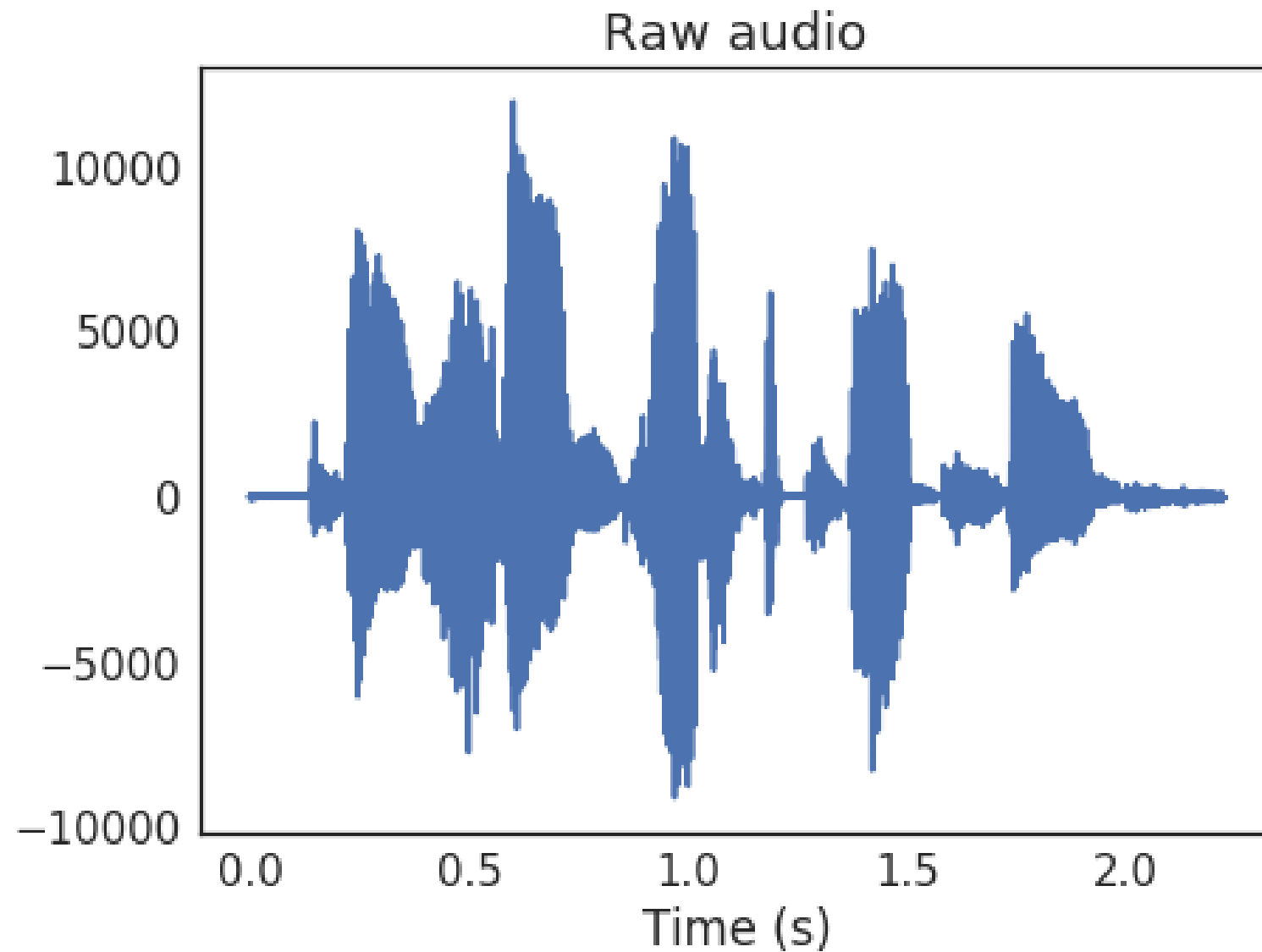
**Chris Holdgraf**

Fellow, Berkeley Institute for Data  
Science

# Fourier transforms

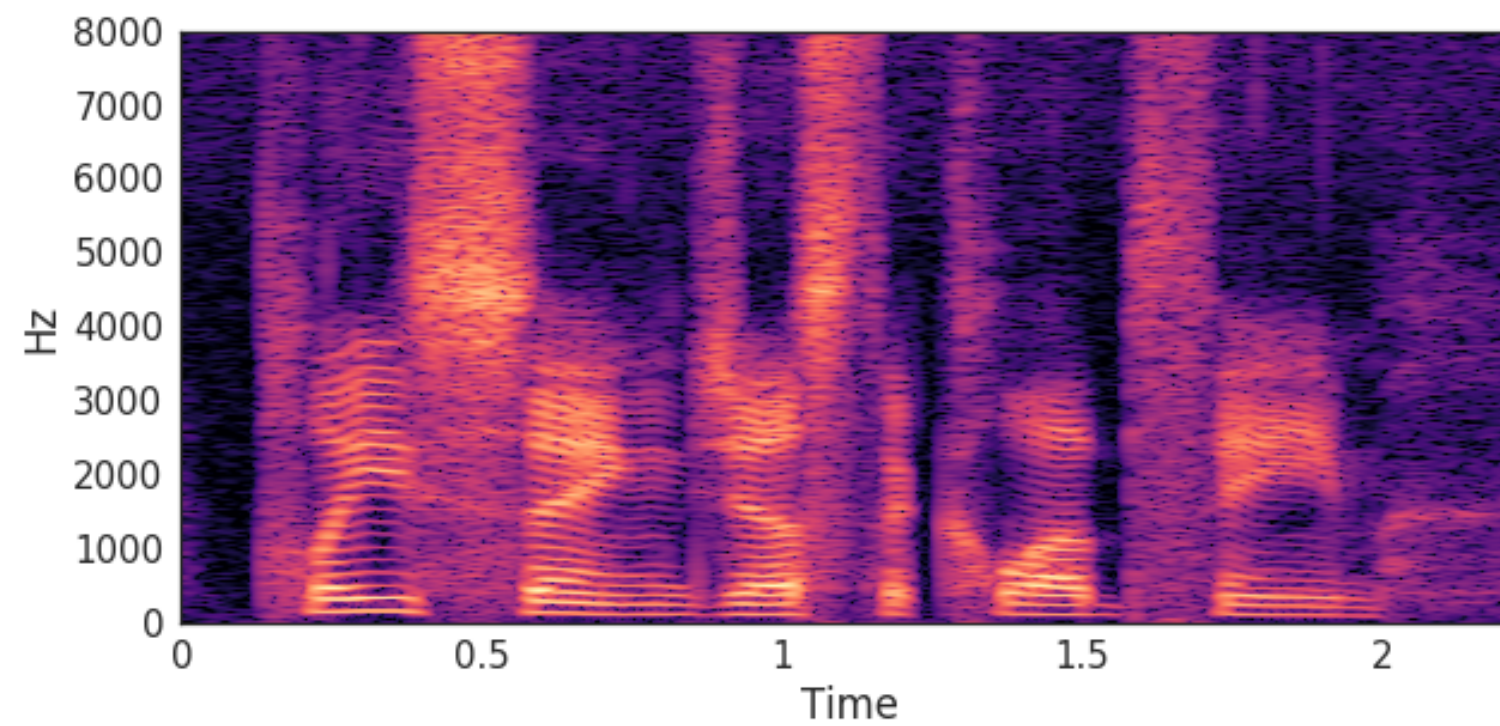
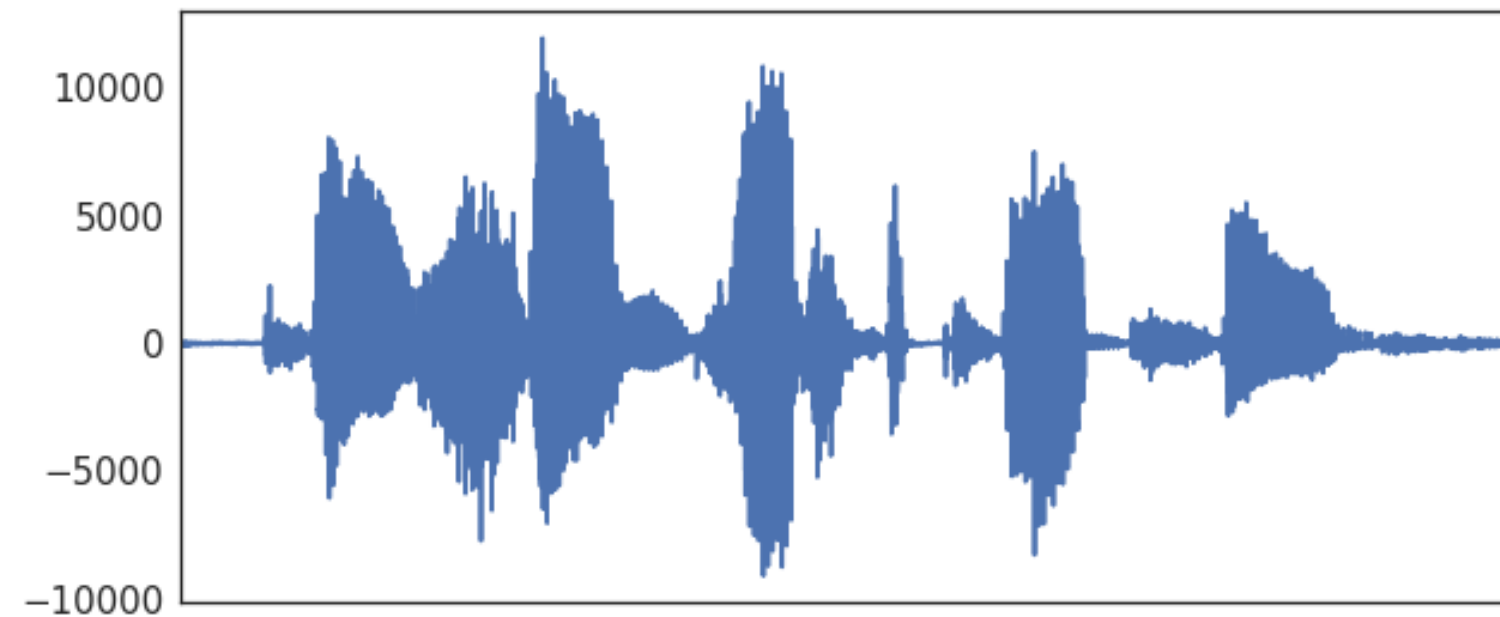
- Timeseries data can be described as a combination of quickly-changing things and slowly-changing things
- At each moment in time, we can describe the relative presence of fast- and slow-moving components
- The simplest way to do this is called a *Fourier Transform*
- This converts a single timeseries into an array that describes the timeseries as a combination of oscillations

# A Fourier Transform (FFT)



# Spectrograms: combinations of windows Fourier transforms

- A spectrogram is a collection of windowed Fourier transforms over time
- Similar to how a rolling mean was calculated:
  1. Choose a window size and shape
  2. At a timepoint, calculate the FFT for that window
  3. Slide the window over by one
  4. Aggregate the results
- Called a *Short-Time Fourier Transform* (STFT)



# Calculating the STFT

- We can calculate the STFT with `librosa`
- There are several parameters we can tweak (such as window size)
- For our purposes, we'll convert into *decibels* which normalizes the average values of all frequencies
- We can then visualize it with the `specshow()` function



# Calculating the STFT with code

```
# Import the functions we'll use for the STFT
from librosa.core import stft, amplitude_to_db
from librosa.display import specshow

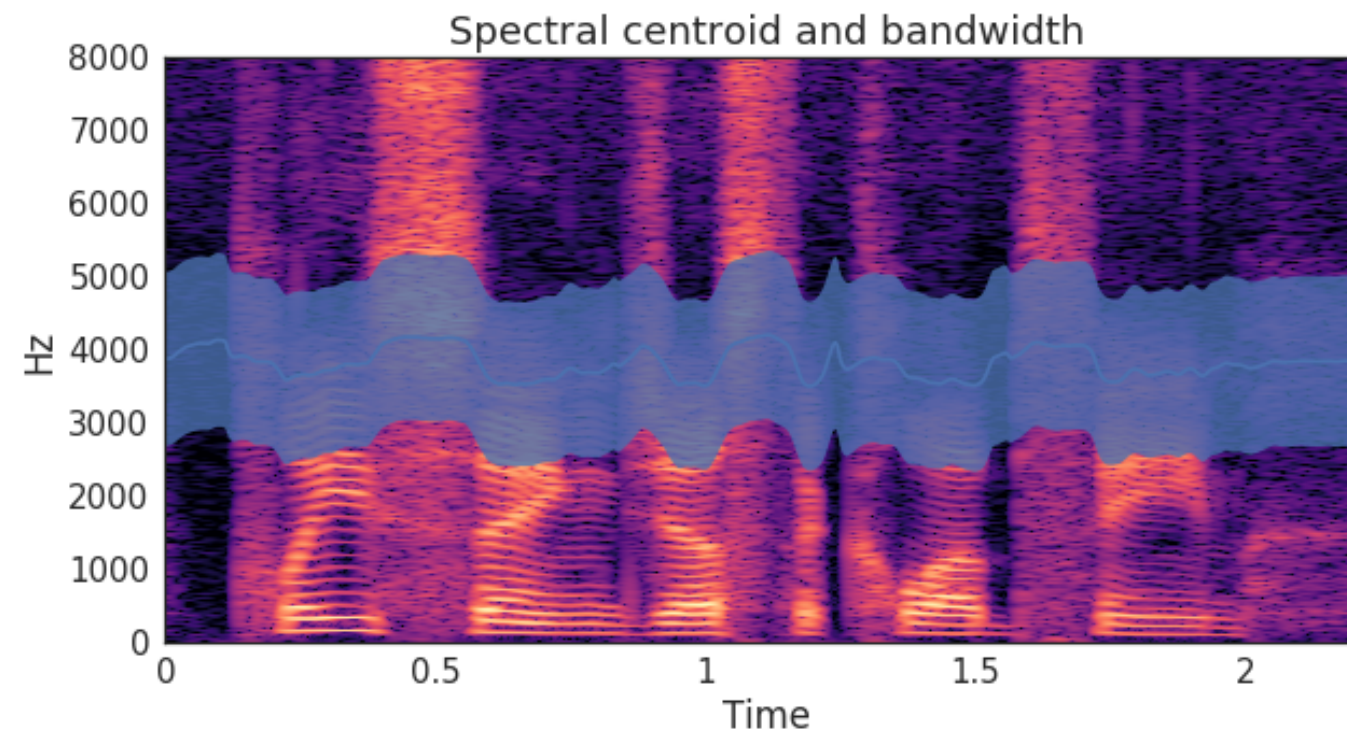
# Calculate our STFT
HOP_LENGTH = 2**4
SIZE_WINDOW = 2**7
audio_spec = stft(audio, hop_length=HOP_LENGTH, n_fft=SIZE_WINDOW)

# Convert into decibels for visualization
spec_db = amplitude_to_db(audio_spec)

# Visualize
specshow(spec_db, sr=sfreq, x_axis='time',
         y_axis='hz', hop_length=HOP_LENGTH)
```

# Spectral feature engineering

- Each timeseries has a different spectral pattern.
- We can calculate these spectral patterns by analyzing the spectrogram.
- For example, **spectral bandwidth** and **spectral centroids** describe where most of the energy is at each moment in time



# Calculating spectral features

```
# Calculate the spectral centroid and bandwidth for the spectrogram
bandwidths = lr.feature.spectral_bandwidth(S=spec)[0]
centroids = lr.feature.spectral_centroid(S=spec)[0]

# Display these features on top of the spectrogram
ax = specshow(spec, x_axis='time', y_axis='hz', hop_length=HOP_LENGTH)
ax.plot(times_spec, centroids)
ax.fill_between(times_spec, centroids - bandwidths / 2,
               centroids + bandwidths / 2, alpha=0.5)
```

# Combining spectral and temporal features in a classifier

```
centroids_all = []
bandwidths_all = []
for spec in spectrograms:
    bandwidths = lr.feature.spectral_bandwidth(S=lr.db_to_amplitude(spec))
    centroids = lr.feature.spectral_centroid(S=lr.db_to_amplitude(spec))
    # Calculate the mean spectral bandwidth
    bandwidths_all.append(np.mean(bandwidths))
    # Calculate the mean spectral centroid
    centroids_all.append(np.mean(centroids))

# Create our X matrix
X = np.column_stack([means, stds, maxs, tempo_mean,
                    tempo_max, tempo_std, bandwidths_all, centroids_all])
```

# Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON