Timeseries kinds and applications

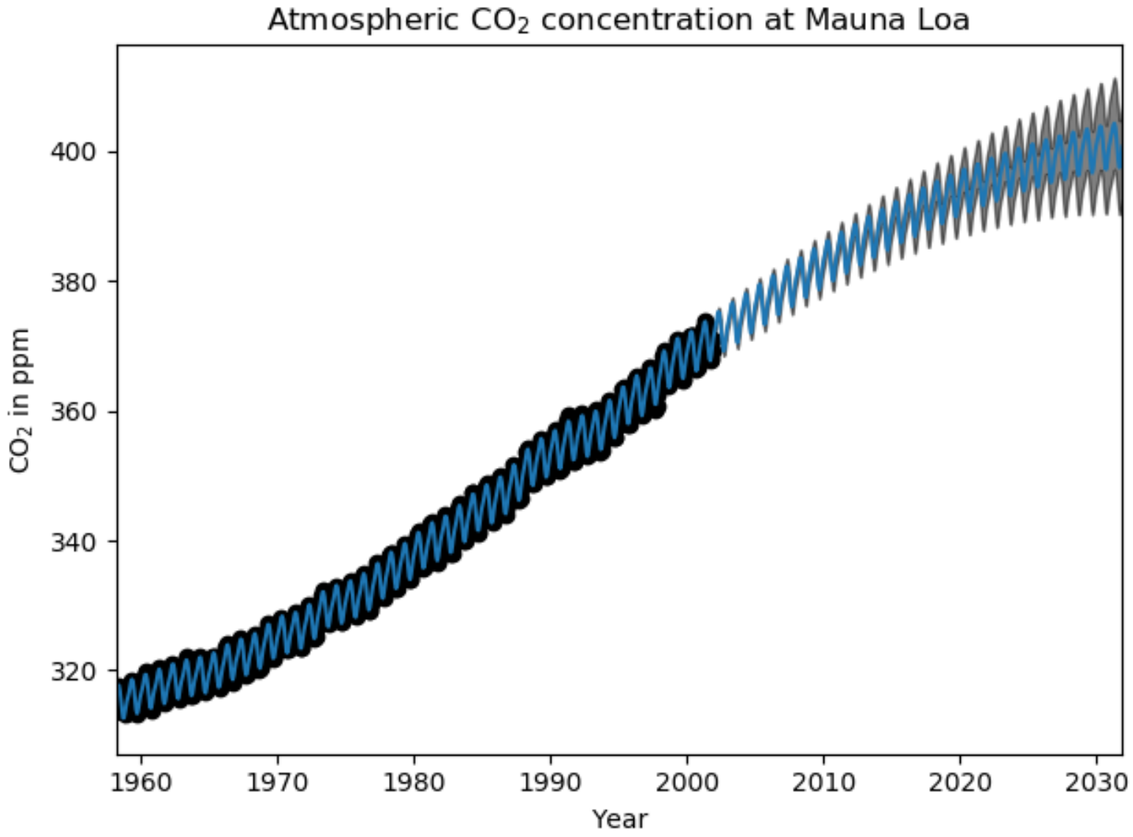
MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

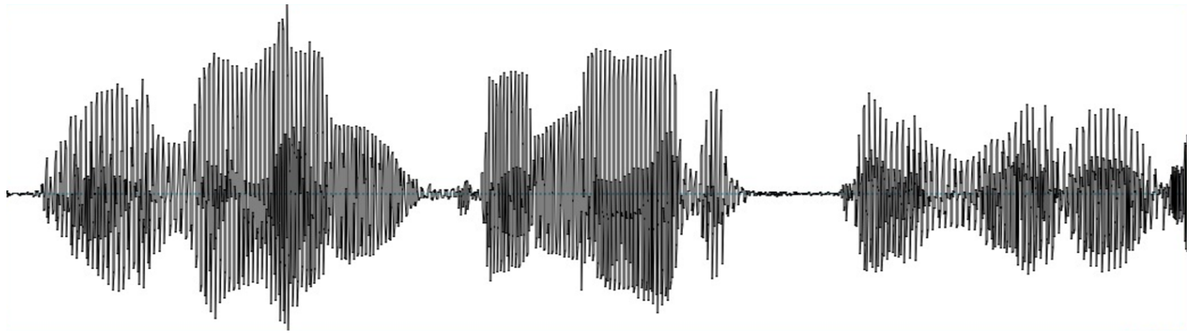
Chris Holdgraf

Fellow, Berkeley Institute for Data Science

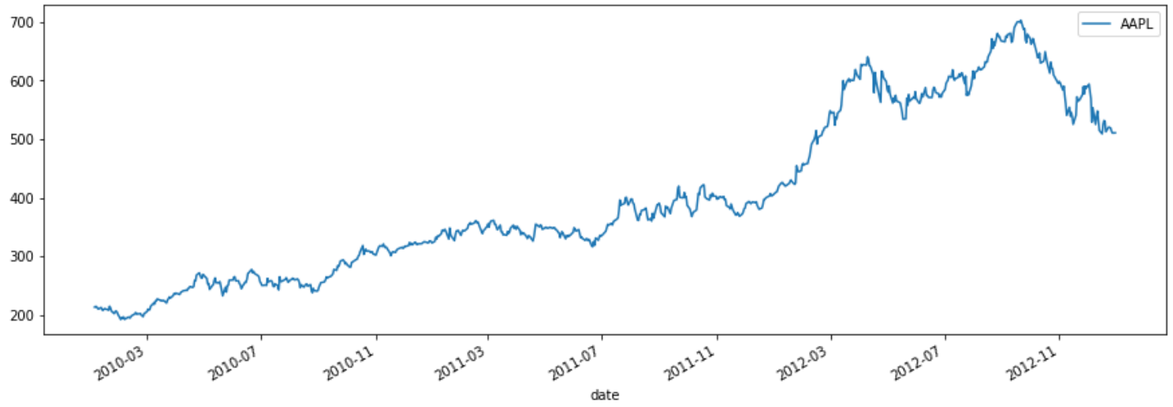
datacamp

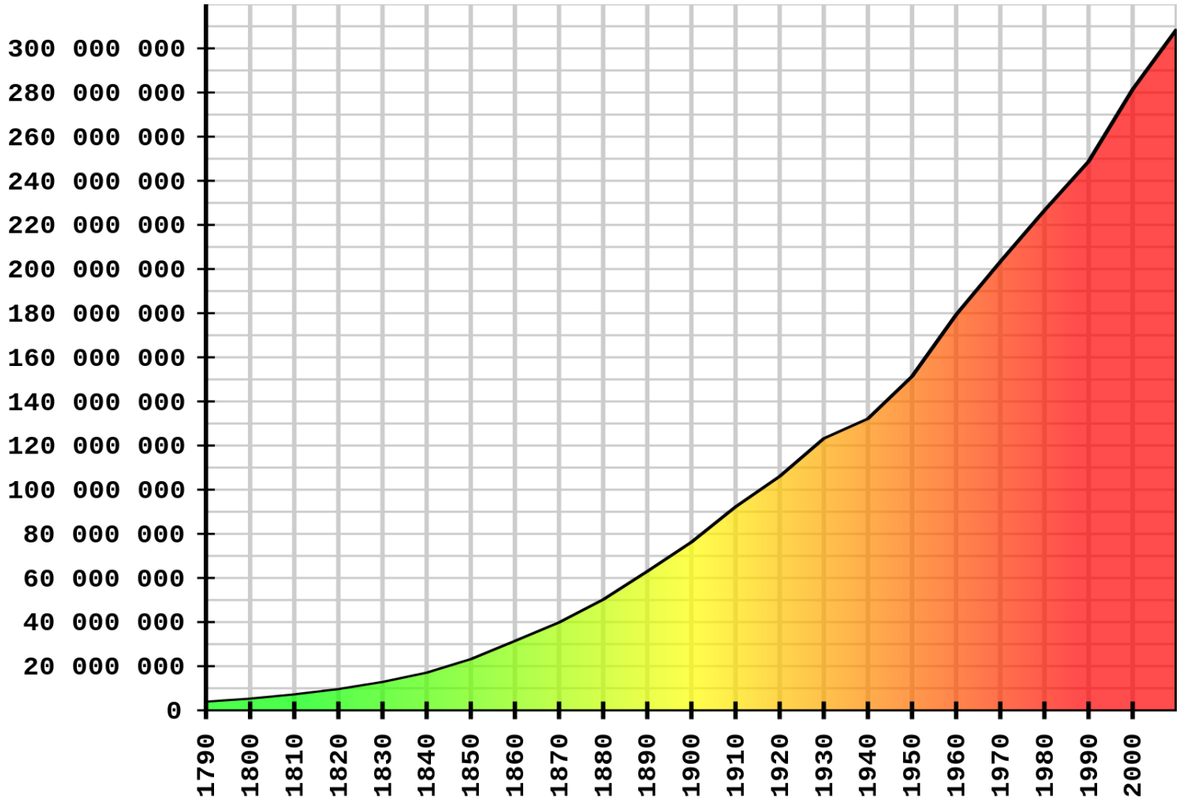
Time Series


Atmospheric CO_2 concentration at Mauna Loa
CO_2 in ppm
400
380
360
340
320
1960
1970
1980
1990
2000
2010
2020
2030
Year



Time Series


AAPL
700
600
500
400
300
200
2010-03
2010-07
2010-11
2011-03
2011-07
2011-11
2012-03
2012-07
2012-11
date


300 000 000
280 000 000
260 000 000
240 000 000
220 000 000
200 000 000
180 000 000
160 000 000
140 000 000
120 000 000
100 000 000
80 000 000
60 000 000
40 000 000
20 000 000
0
1790
1800
1810
1820
1830
1840
1850
1860
1870
1880
1890
1900
1910
1920
1930
1940
1950
1960
1970
1980
1990
2000

What makes a time series?

Datapoint	Datapoint	Datapoint	Datapoint	Datapoint	Datapoint
1	34	12	54	76	40

Timepoint	Timepoint	Timepoint	Timepoint	Timepoint	Timepoint
2:00	2:01	2:02	2:03	2:04	2:05

Timepoint	Timepoint	Timepoint	Timepoint	Timepoint	Timepoint
Jan	Feb	March	April	May	Jun

Timepoint	Timepoint	Timepoint	Timepoint	Timepoint	Timepoint
1e-9	2e-9	3e-9	4e-9	5e-9	6e-9

Reading in a time series with Pandas

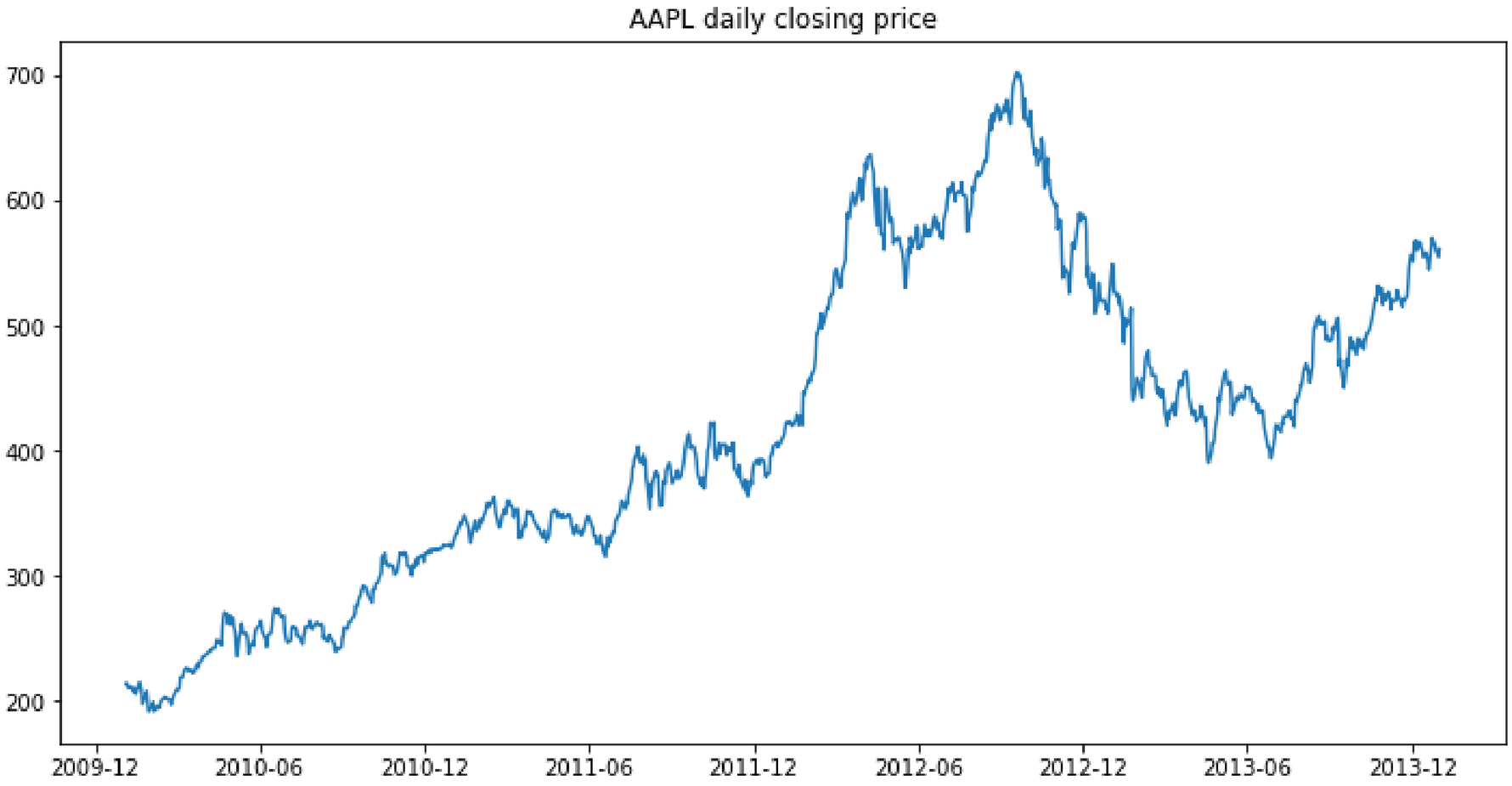
```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('data.csv')
data.head()
```

```
          date symbol       close       volume
0   2010-01-04   AAPL  214.009998  123432400.0
46  2010-01-05   AAPL  214.379993  150476200.0
92  2010-01-06   AAPL  210.969995  138040000.0
138 2010-01-07   AAPL  210.580000  119282800.0
184 2010-01-08   AAPL  211.980005  111902700.0
```

Plotting a pandas timeseries

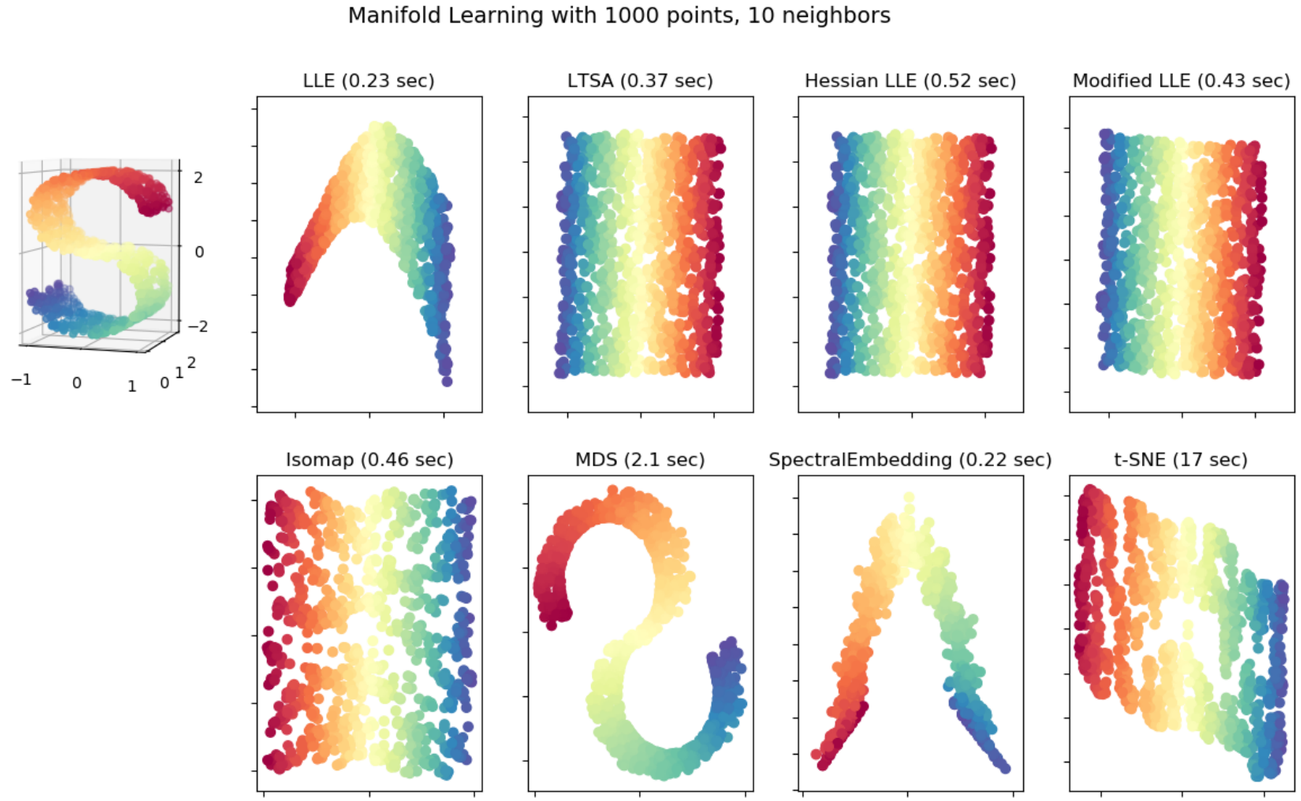
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(12, 6))
data.plot('date', 'close', ax=ax)
ax.set(title="AAPL daily closing price")
```

A timeseries plot



Why machine learning?

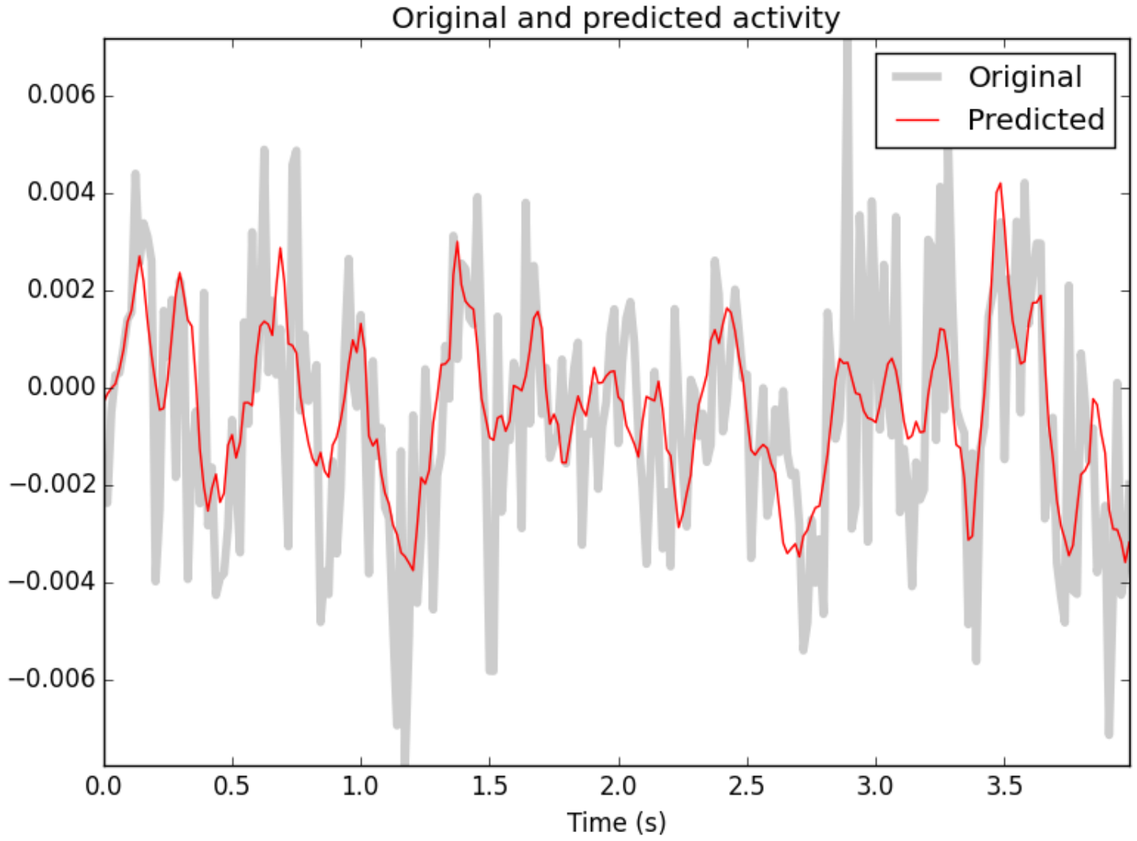
We can use really big data and really complicated data



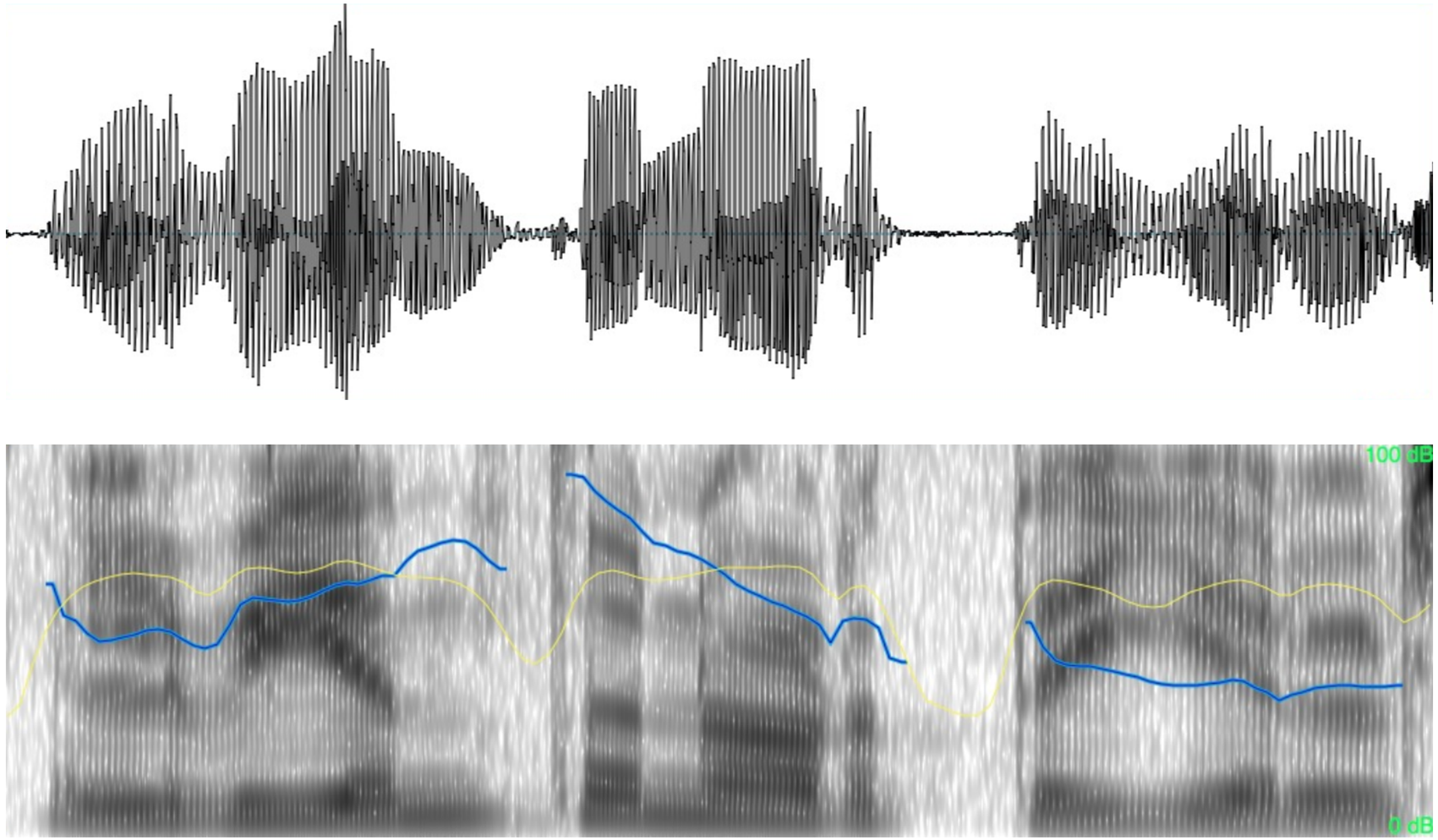
Why machine learning?

We can...

- Predict the future
- Automate this process



Why combine these two?



A machine learning pipeline

- Feature extraction
- Model fitting
- Prediction and validation

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Machine learning basics

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

datacamp

Always begin by looking at your data

```
array.shape
```

```
(10, 5)
```

```
array[:3]
```

```
array([[ 0.735528  ,  1.00122818, -0.28315978],
       [-0.94478393,  0.18658748, -0.00241224],
       [-0.74822942, -1.46636618,  0.69835096]])
```

Always begin by looking at your data

```
df.head()
```

```
       col1      col2      col3
0  0.735528  1.001228 -0.283160
1 -0.944784  0.186587 -0.002412
2 -0.748229 -1.466366  0.698351
3  1.038589 -0.171248  0.831457
4 -0.161904  0.003972 -0.321933
```

Always visualize your data

Make sure it looks the way you'd expect.

```
# Using matplotlib
fig, ax = plt.subplots()
ax.plot(...)

# Using pandas
fig, ax = plt.subplots()
df.plot(..., ax=ax)
```

Scikit-learn

Scikit-learn is the most popular machine learning library in Python

```
from sklearn.svm import LinearSVC
```

Preparing data for scikit-learn

- `scikit-learn` expects a particular structure of data:

  `(samples, features)`

- Make sure that your data is *at least two-dimensional*
- Make sure the first dimension is *samples*

If your data is not shaped properly

- If the axes are swapped:

```
array.T.shape
```

```
(10, 3)
```

If your data is not shaped properly

- If we're missing an axis, use `.reshape()` :

```
array.shape
```

```
(10,)
```

```
array.reshape([-1, 1]).shape
```

```
(10, 1)
```

- `-1` will automatically fill that axis with remaining values

Fitting a model with scikit-learn

```
# Import a support vector classifier
from sklearn.svm import LinearSVC

# Instantiate this model
model = LinearSVC()

# Fit the model on some data
model.fit(X, y)
```

It is common for `y` to be of shape `(samples, 1)`

Investigating the model

```
# There is one coefficient per input feature
model.coef_
```

```
array([[ 0.69417875, -0.5289162 ]])
```

Predicting with a fit model

```
# Generate predictions
predictions = model.predict(X_test)
```

Let's practice

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

datacamp

Combining timeseries data with machine learning

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

datacamp

Getting to know our data

- The datasets that we'll use in this course are all freely-available online
- There are many datasets available to download on the web, the ones we'll use come from Kaggle

The Heartbeat Acoustic Data

- Many recordings of heart sounds from different patients
- Some had normally-functioning hearts, others had abnormalities
- Data comes in the form of audio files + labels for each file
- Can we find the "abnormal" heart beats?

Loading auditory data

```
from glob import glob
files = glob('data/heartbeat-sounds/files/*.wav')


print(files)
```

```
['data/heartbeat-sounds/proc/files/murmur__201101051104.wav',
 ...
 'data/heartbeat-sounds/proc/files/murmur__201101051114.wav']
```

Reading in auditory data

```
import librosa as lr
# `load` accepts a path to an audio file
audio, sfreq = lr.load('data/heartbeat-sounds/proc/files/murmur__201101051104.wav')

print(sfreq)
```

```
2205
```

In this case, the sampling frequency is `2205` , meaning there are `2205` samples per second

Inferring time from samples

- If we know the sampling rate of a timeseries, then we know the timestamp of each datapoint *relative to the first datapoint*
- Note: this assumes the sampling rate is fixed and no data points are lost

Creating a time array (I)

- Create an array of indices, one for each sample, and divide by the sampling frequency

```
indices = np.arange(0, len(audio))
time = indices / sfreq
```

Creating a time array (II)

- Find the time stamp for the *N-1*th data point. Then use `linspace()` to interpolate from zero to that time

```
final_time = (len(audio) - 1) / sfreq
time = np.linspace(0, final_time, sfreq)
```

The New York Stock Exchange dataset

- This dataset consists of company stock values for 10 years
- Can we detect any patterns in historical records that allow us to predict the value of companies in the future?

Looking at the data

```
data = pd.read_csv('path/to/data.csv')


data.columns
```

```
Index(['date', 'symbol', 'close', 'volume'], dtype='object')
```

```
data.head()
```

```
         date symbol       close       volume
0  2010-01-04   AAPL  214.009998  123432400.0
1  2010-01-04    ABT   54.459951   10829000.0
2  2010-01-04    AIG   29.889999    7750900.0
3  2010-01-04   AMAT   14.300000   18615100.0
4  2010-01-04   ARNC   16.650013   11512100.0
```

Timeseries with Pandas DataFrames

- We can investigate the object type of each column by accessing the `dtypes` attribute

```
df['date'].dtypes
```

```
0    object
1    object
2    object
dtype: object
```

Converting a column to a time series

- To ensure that a column within a DataFrame is treated as time series, use the `to_datetime()` function

```
df['date'] = pd.to_datetime(df['date'])


df['date']
```

```
0   2017-01-01
1   2017-01-02
2   2017-01-03
Name: date, dtype: datetime64[ns]
```

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Classification and feature engineering

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

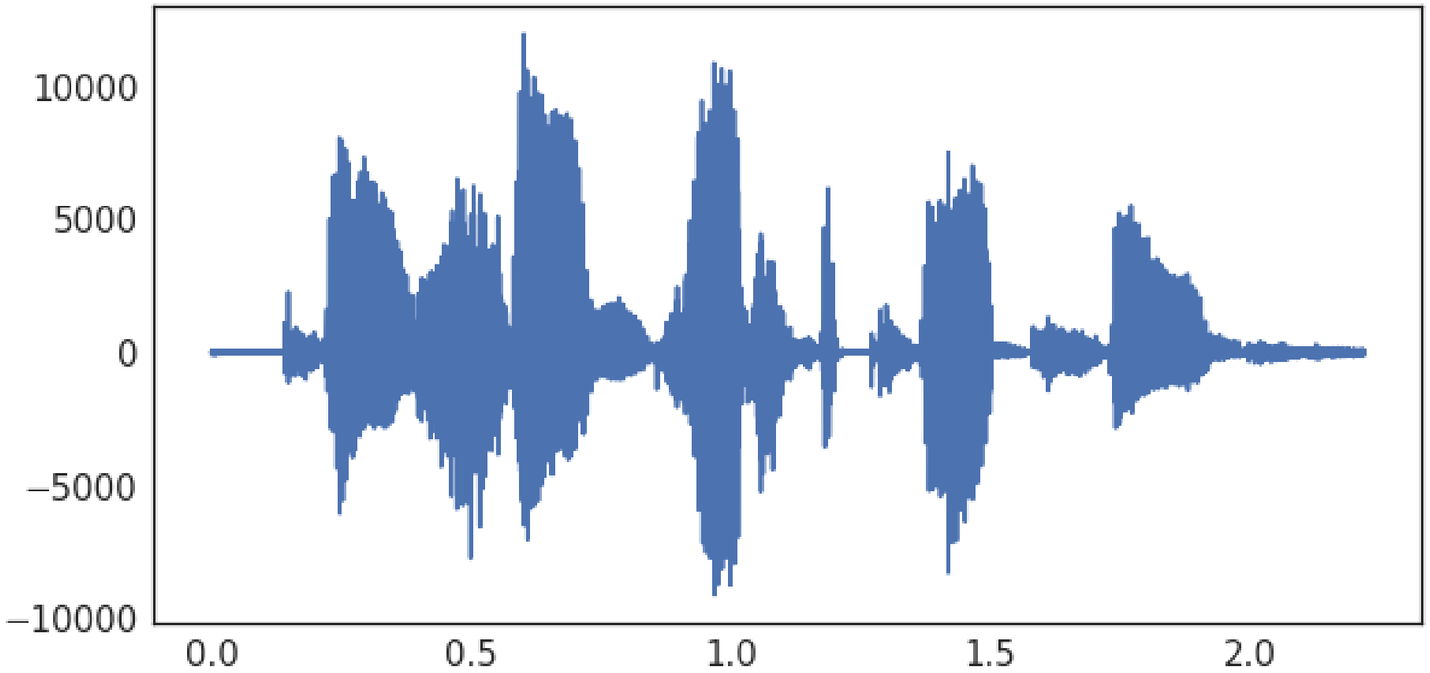
Fellow, Berkeley Institute for Data Science

datacamp

Always visualize raw data before fitting models

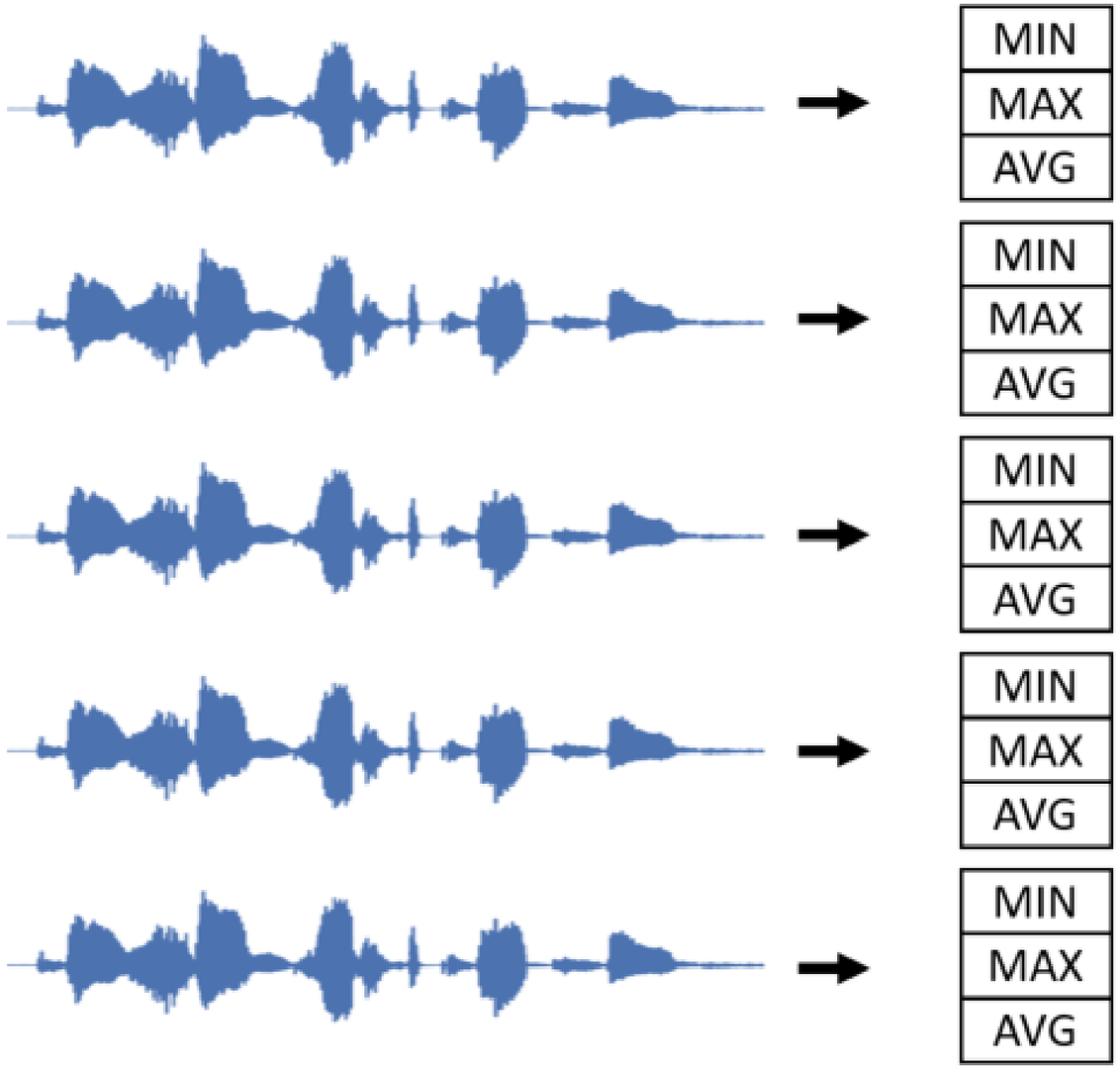
Visualize your timeseries data!

```
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


MIN
MAX
AVG
MIN
MAX
AVG
MIN
MAX
AVG
MIN
MAX
AVG
MIN
MAX
AVG

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

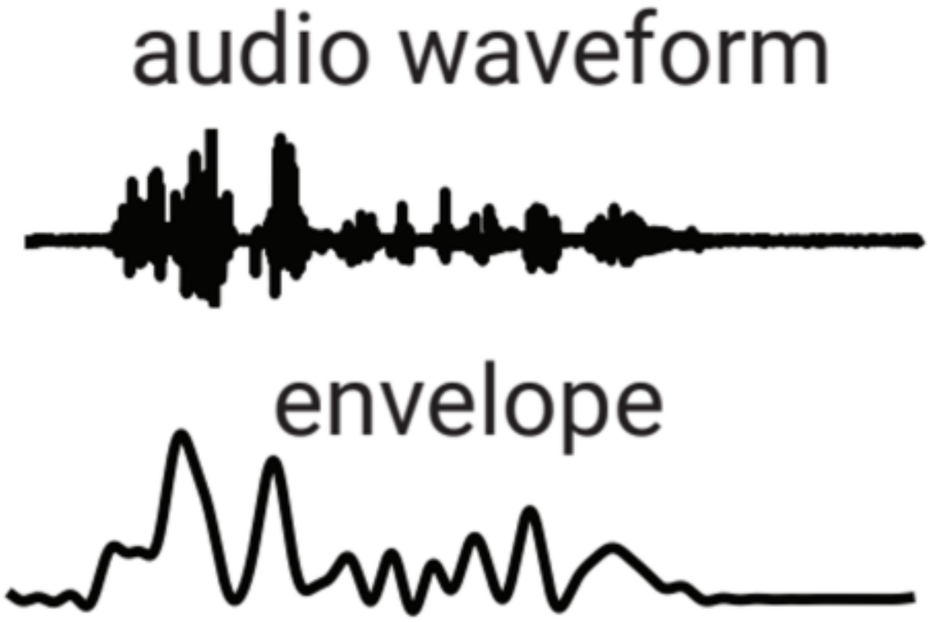
MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf
Fellow, Berkeley Institute for Data Science

datacamp

The auditory envelope

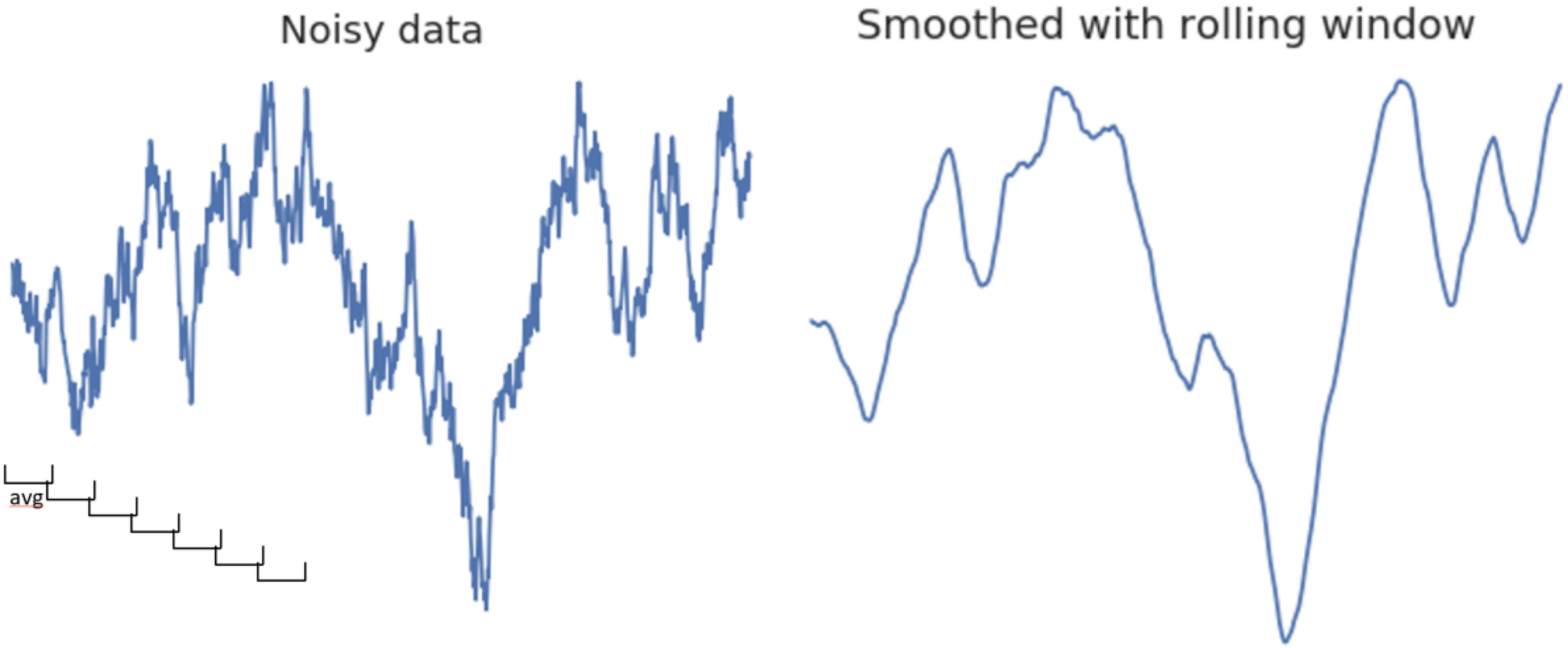
- Smooth the data to calculate the auditory envelope
- Related to the total amount of audio energy present at each moment of time



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



Calculating a rolling window statistic

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

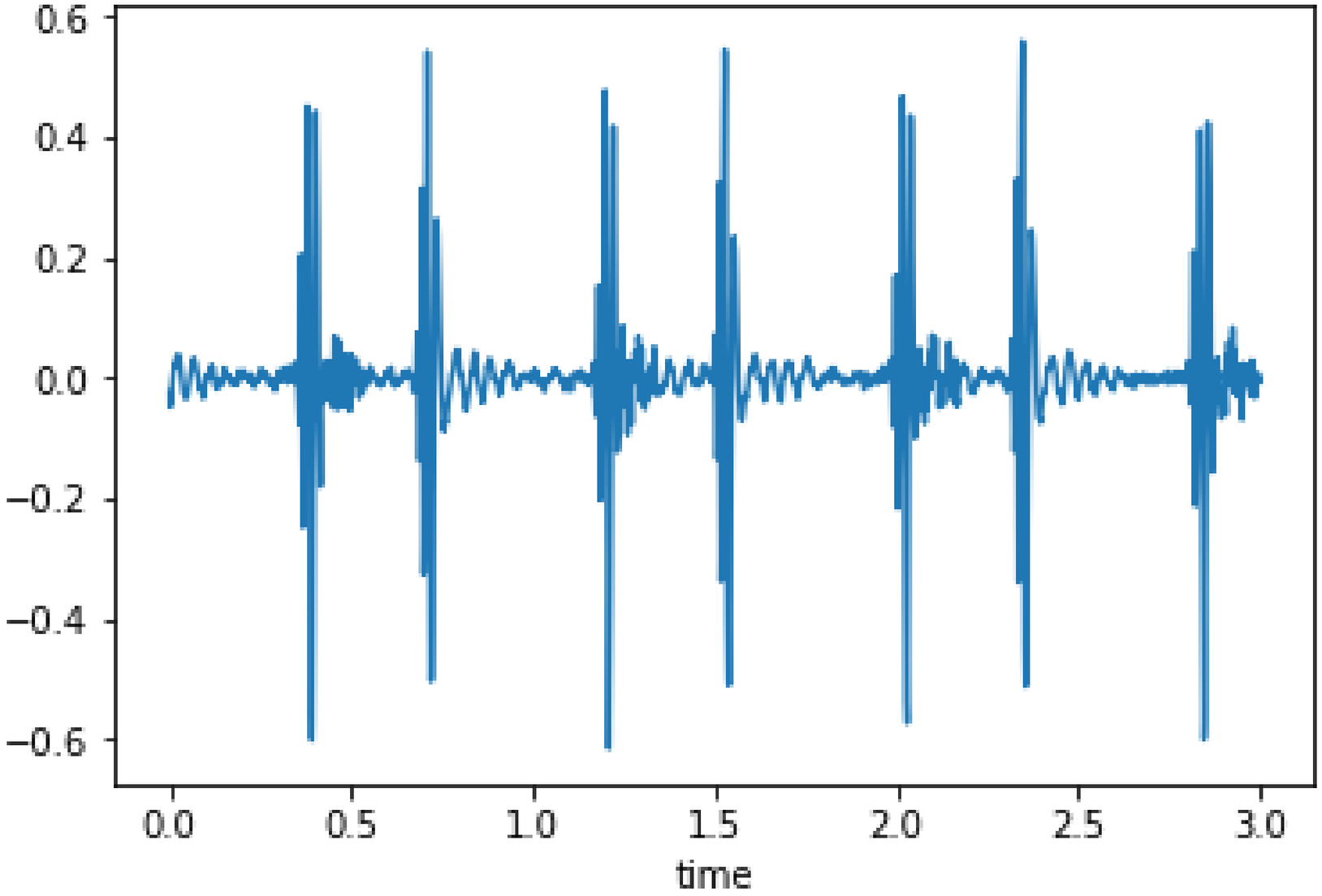
```
(5000, 20)
```

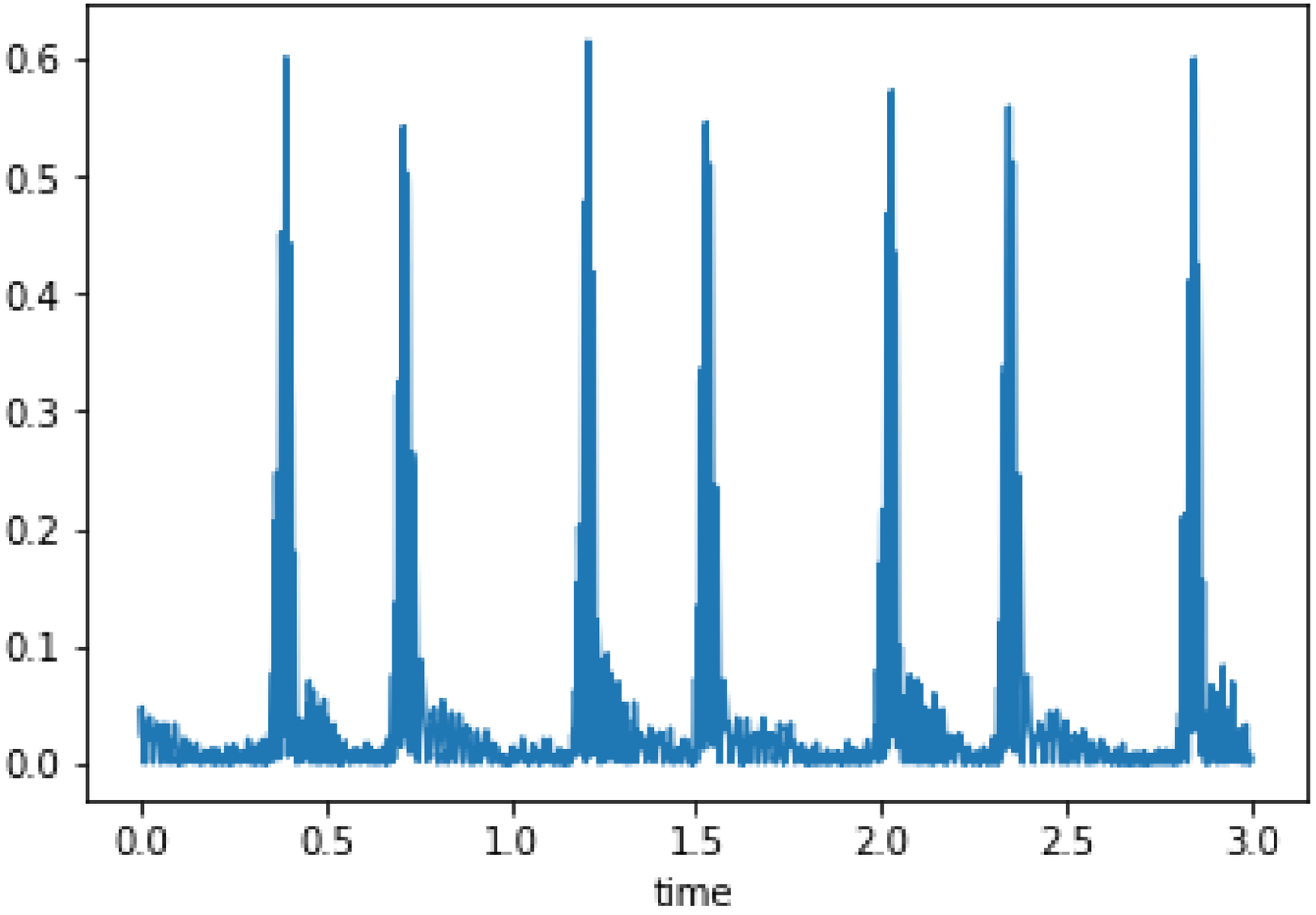
```
# Smooth our data by taking the rolling mean in a window of 50 samples
window_size = 50
windowed = audio.rolling(window=window_size)
audio_smooth = windowed.mean()
```

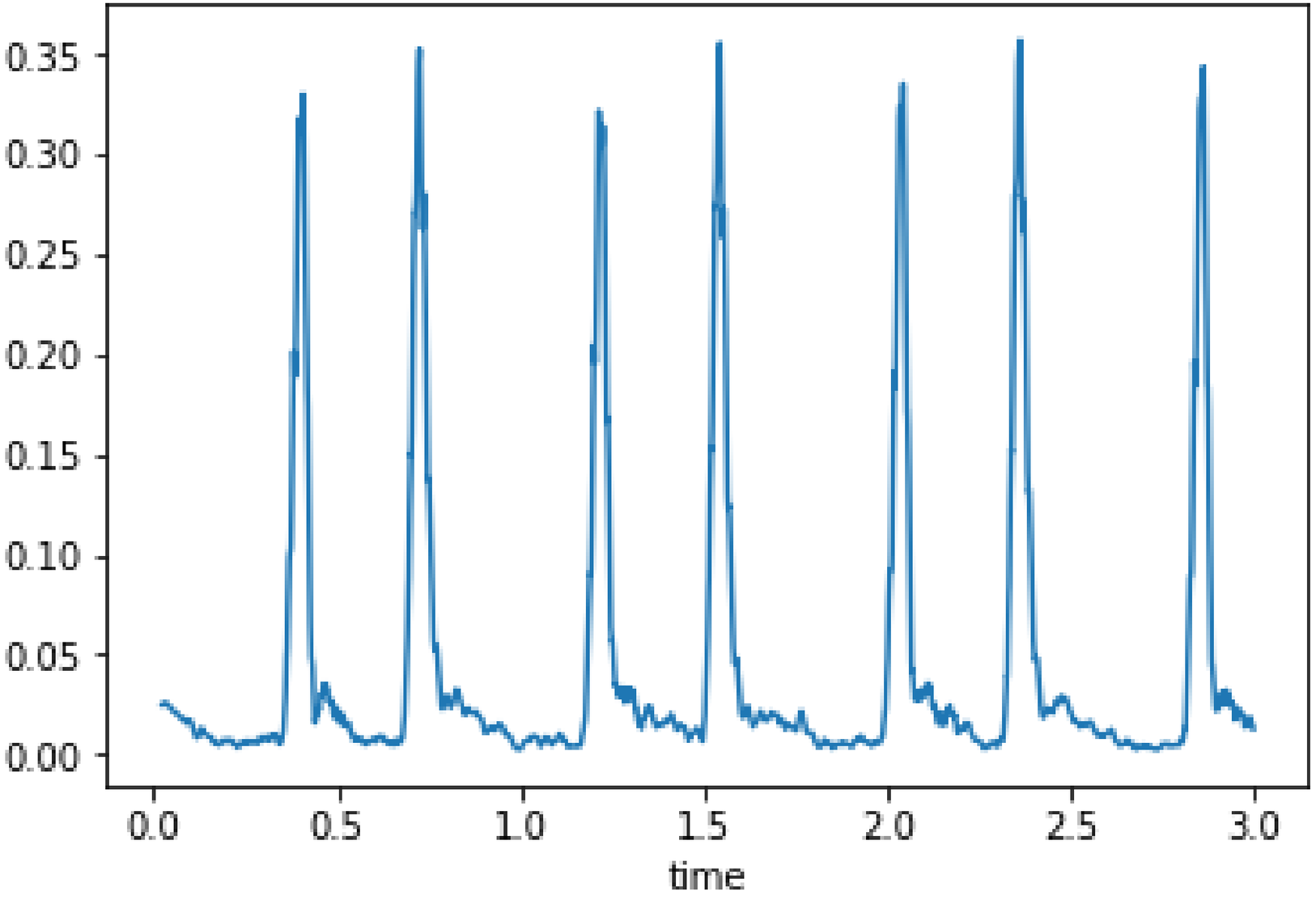
Calculating the auditory envelope

- First *rectify* your audio, then smooth it

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


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time


0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0
time

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

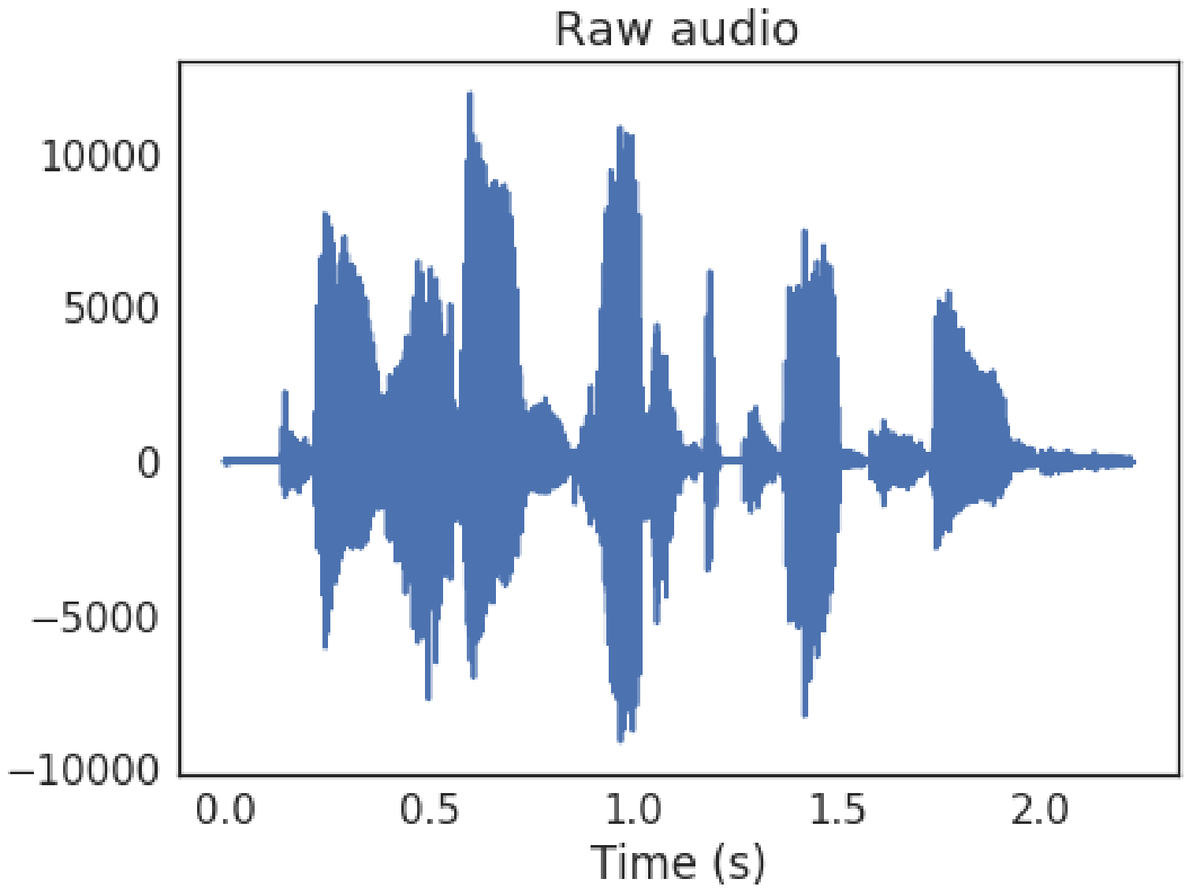
Chris Holdgraf
Fellow, Berkeley Institute for Data Science

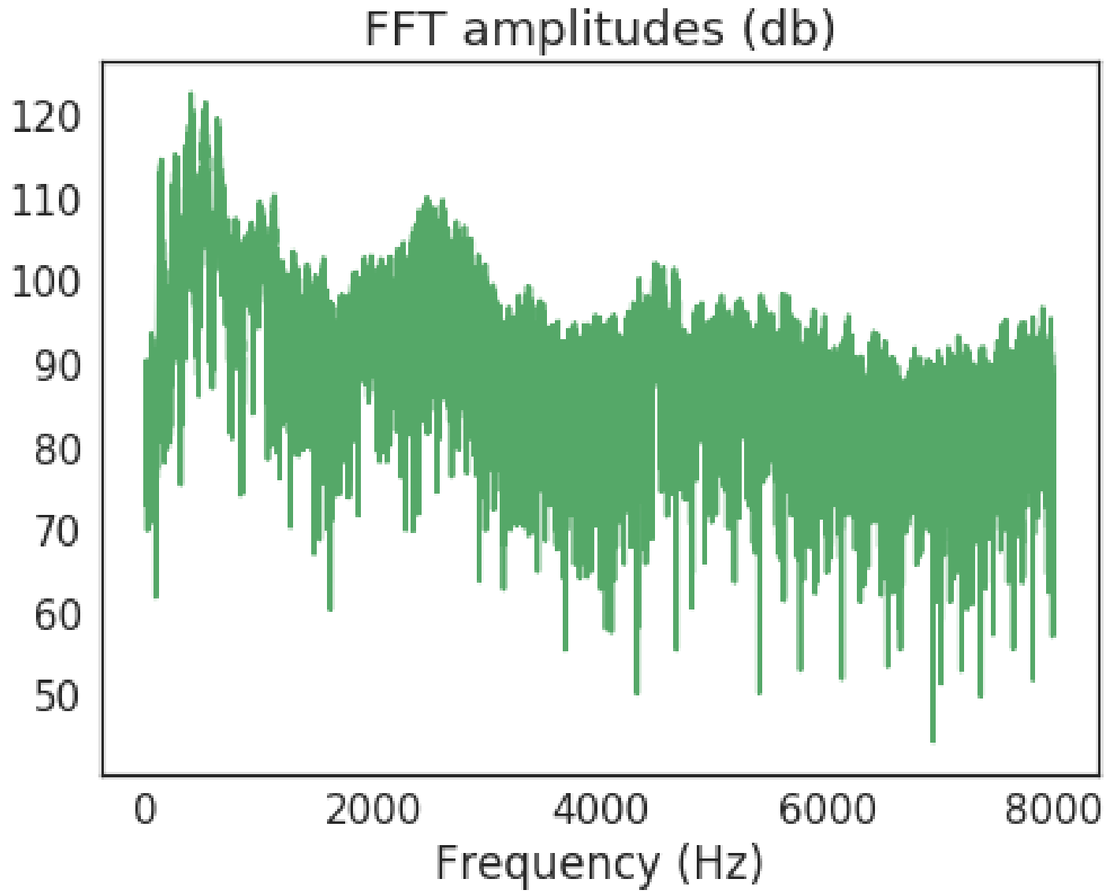
datacamp

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

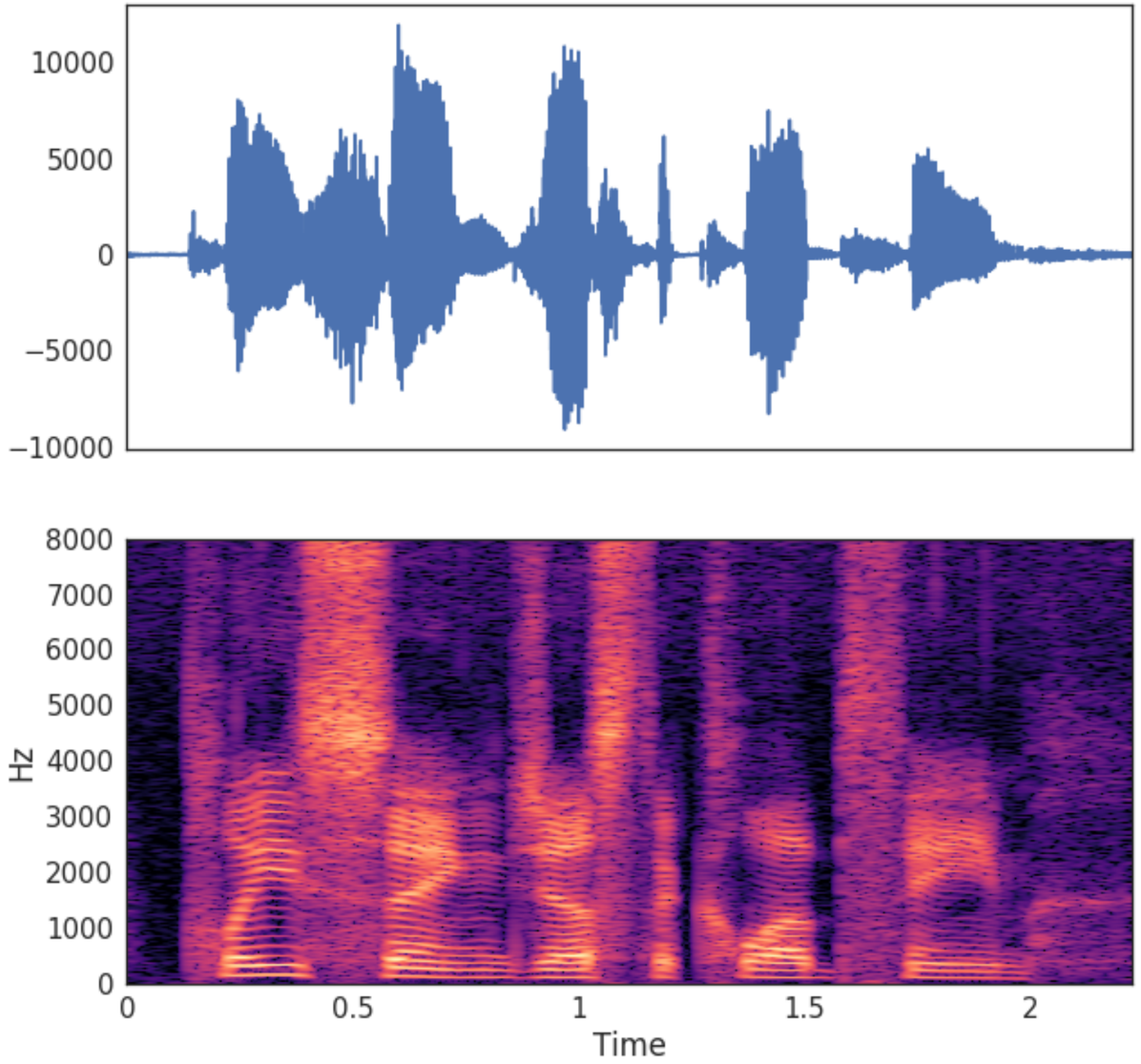
A Fourier Transform (FFT)





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)


10000
5000
0
−5000
−10000
8000
7000
6000
5000
4000
3000
2000
1000
0
Hz
0
0.5
1
1.5
2
Time

Calculating the STFT

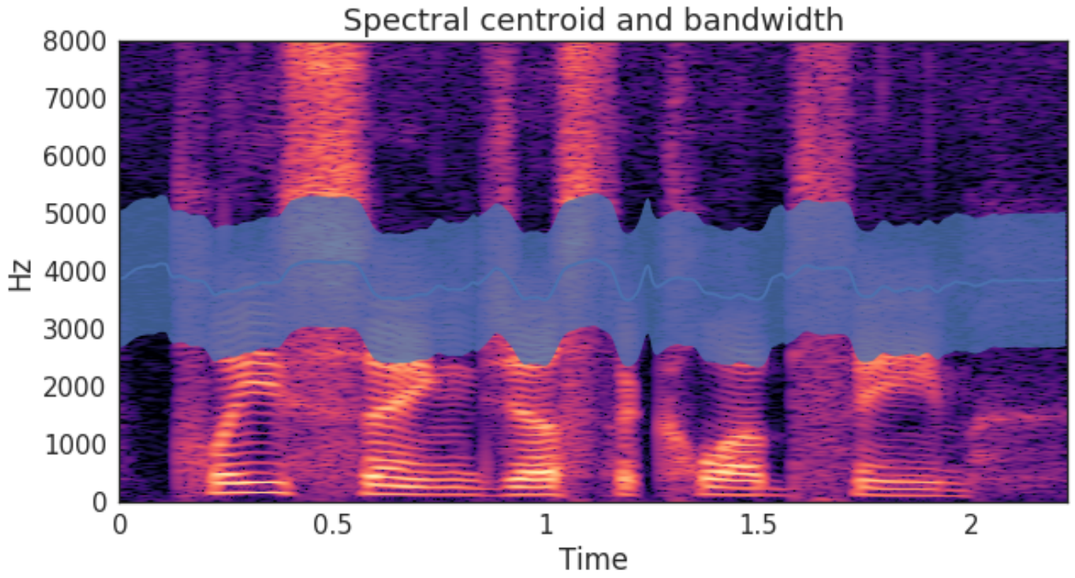
- We can calculate the STFT with `librosa`
- There are several parameters we can tweak (such as window size)
- For our purposes, we'll convert into *decibels* which normalizes the average values of all frequencies
- We can then visualize it with the `specshow()` function

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

Predicting data over time

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

Classification vs. Regression

CLASSIFICATION

```
classification_model.predict(X_test)
```

```
array([0, 1, 1, 0])
```

REGRESSION

```
regression_model.predict(X_test)
```

```
array([0.2, 1.4, 3.6, 0.6])
```

Correlation and regression

- Regression is similar to calculating correlation, with some key differences
  - **Regression**: A process that results in a formal model of the data
  - **Correlation**: A statistic that describes the data. Less information than regression model.

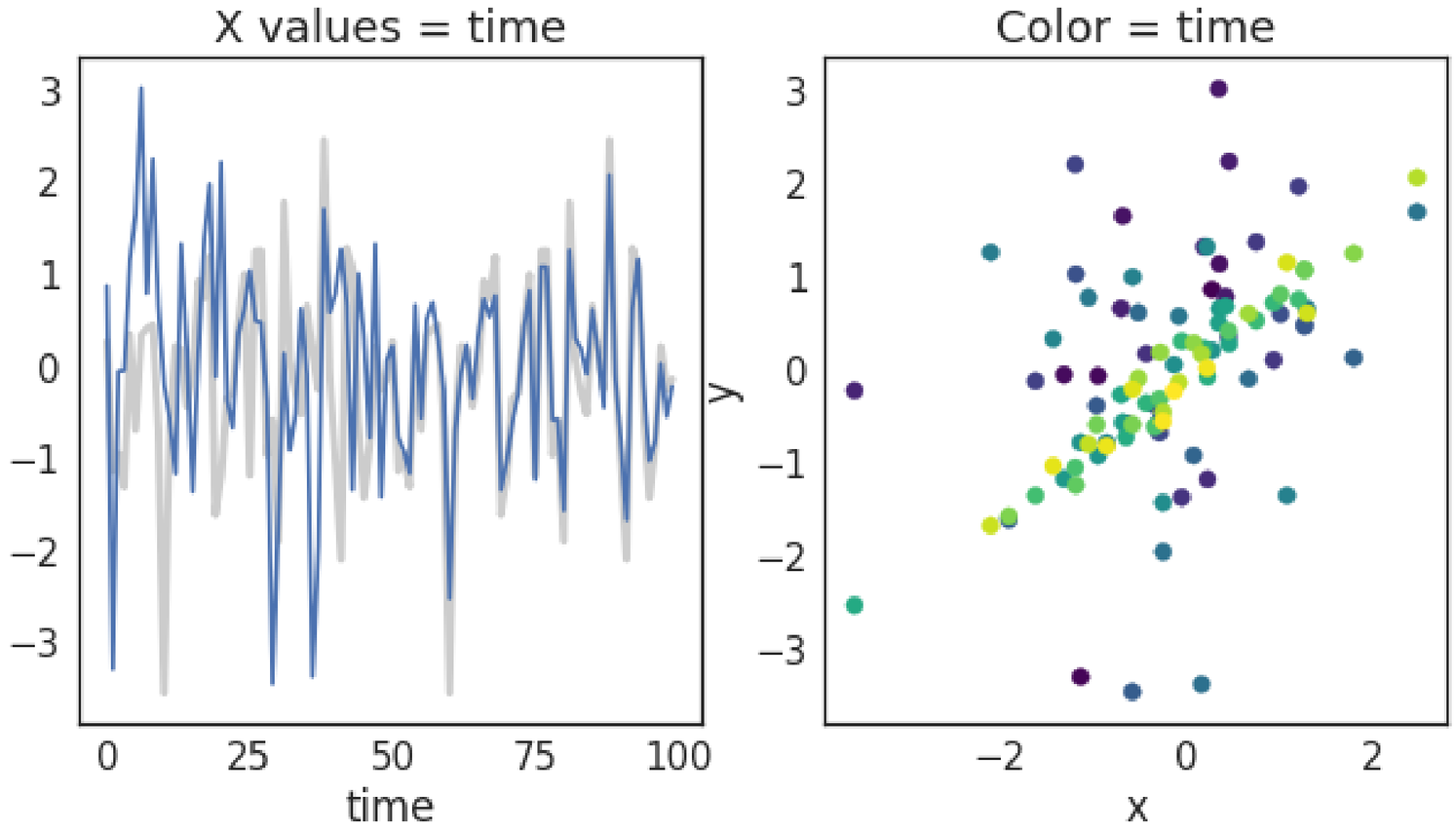
Correlation between variables often changes over time

- Timeseries often have patterns that change over time
- Two timeseries that seem correlated at one moment may not remain so over time

Visualizing relationships between timeseries

```
fig, axs = plt.subplots(1, 2)

# Make a line plot for each timeseries
axs[0].plot(x, c='k', lw=3, alpha=.2)
axs[0].plot(y)
axs[0].set(xlabel='time', title='X values = time')

# Encode time as color in a scatterplot
axs[1].scatter(x_long, y_long, c=np.arange(len(x_long)), cmap='viridis')
axs[1].set(xlabel='x', ylabel='y', title='Color = time')
```

Visualizing two timeseries



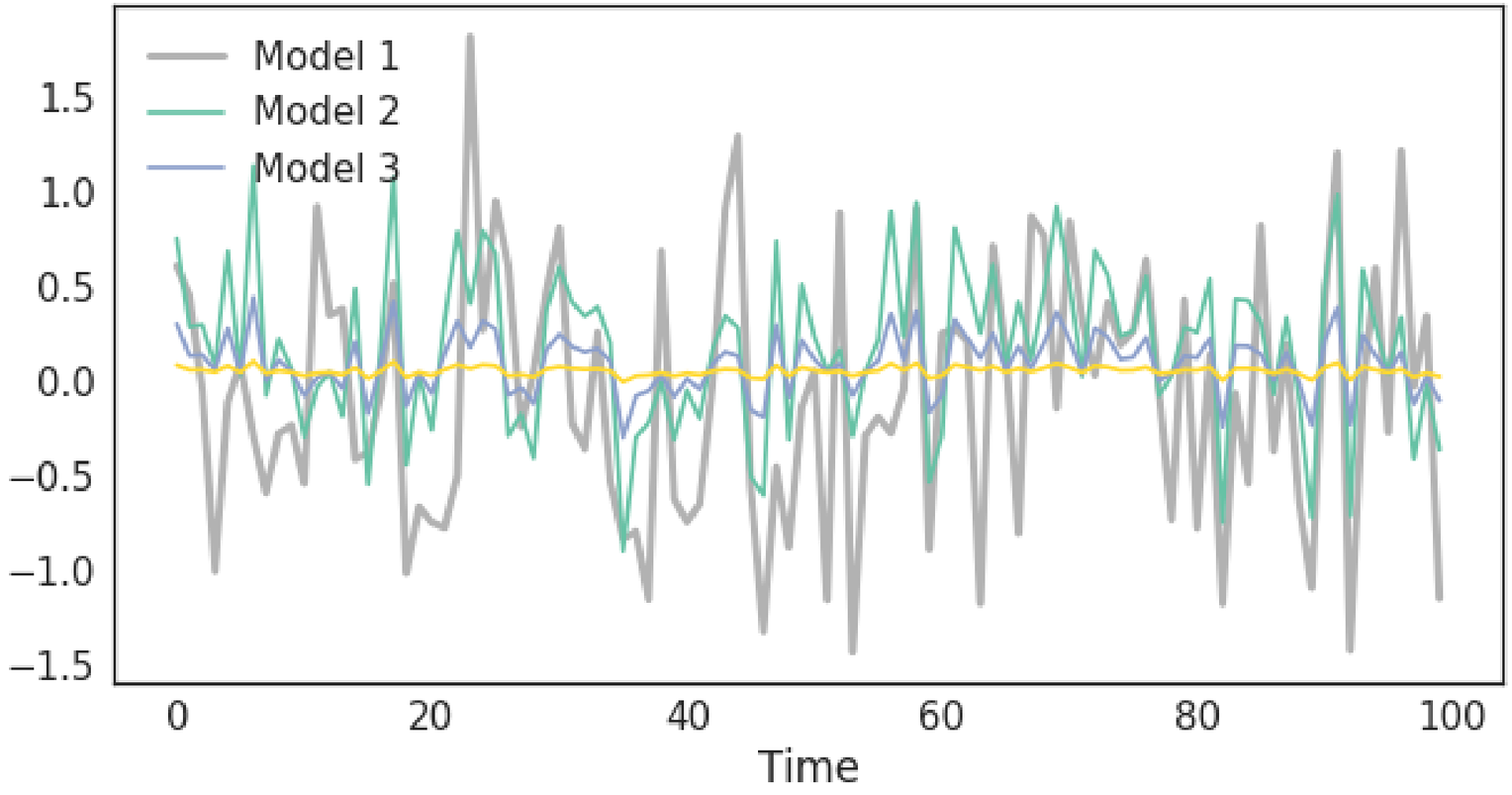
Regression models with scikit-learn

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)
model.predict(X)
```

Visualize predictions with scikit-learn

```
alphas = [.1, 1e2, 1e3]
ax.plot(y_test, color='k', alpha=.3, lw=3)
for ii, alpha in enumerate(alphas):
    y_predicted = Ridge(alpha=alpha).fit(X_train, y_train).predict(X_test)
    ax.plot(y_predicted, c=cmap(ii / len(alphas)))
ax.legend(['True values', 'Model 1', 'Model 2', 'Model 3'])
ax.set(xlabel="Time")
```

Visualize predictions with scikit-learn



Scoring regression models

- Two most common methods:
  - Correlation (r)
  - Coefficient of Determination (R^2)

Coefficient of Determination (R^2)

- The value of R^2 is bounded on the top by 1, and can be infinitely low
- Values closer to 1 mean the model does a better job of predicting outputs

$$1 - \frac{error(model)}{variance(testdata)}$$

R^2 in scikit-learn

```
from sklearn.metrics import r2_score
print(r2_score(y_predicted, y_test))
```

```
0.08
```

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Cleaning and improving your data

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

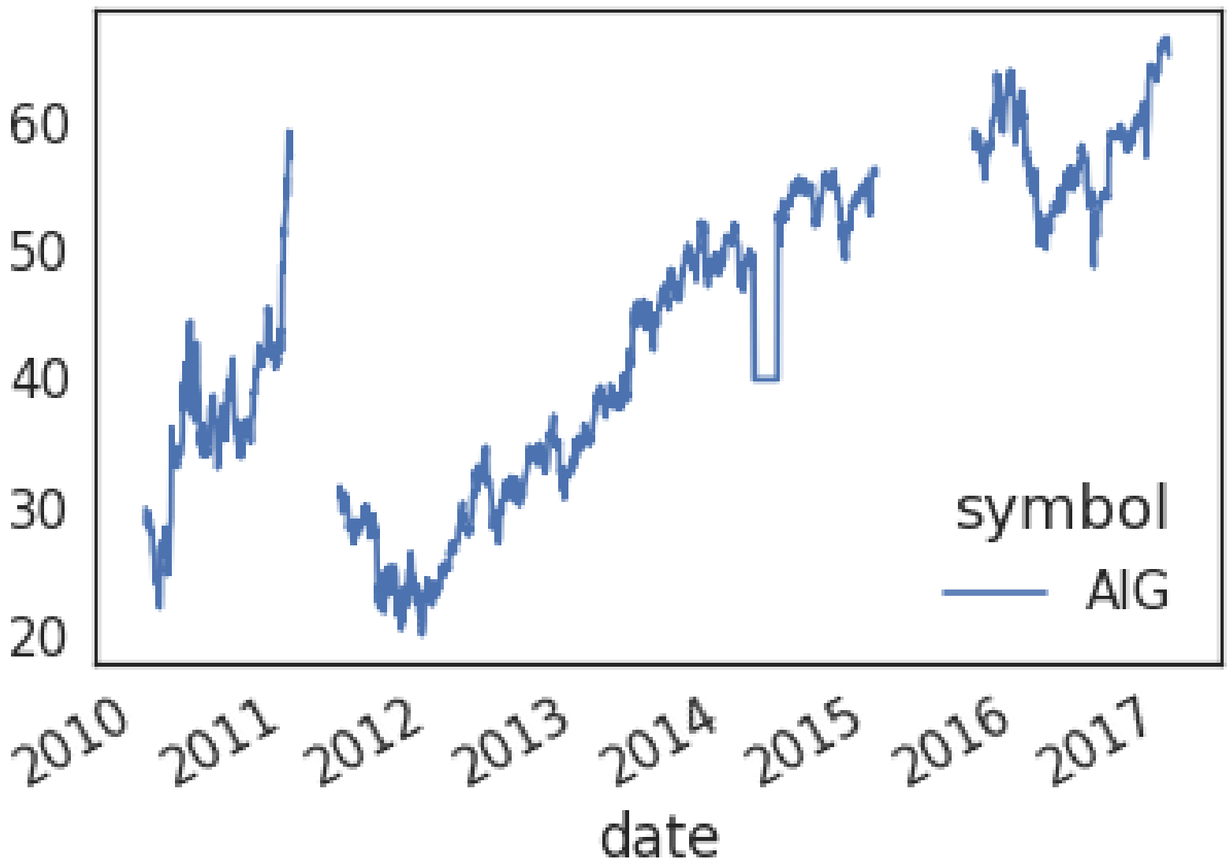
Fellow, Berkeley Institute for Data Science

datacamp

Data is messy

- Real-world data is often messy
- The two most common problems are *missing data* and *outliers*
- This often happens because of human error, machine sensor malfunction, database failures, etc
- Visualizing your raw data makes it easier to spot these problems

What messy data looks like



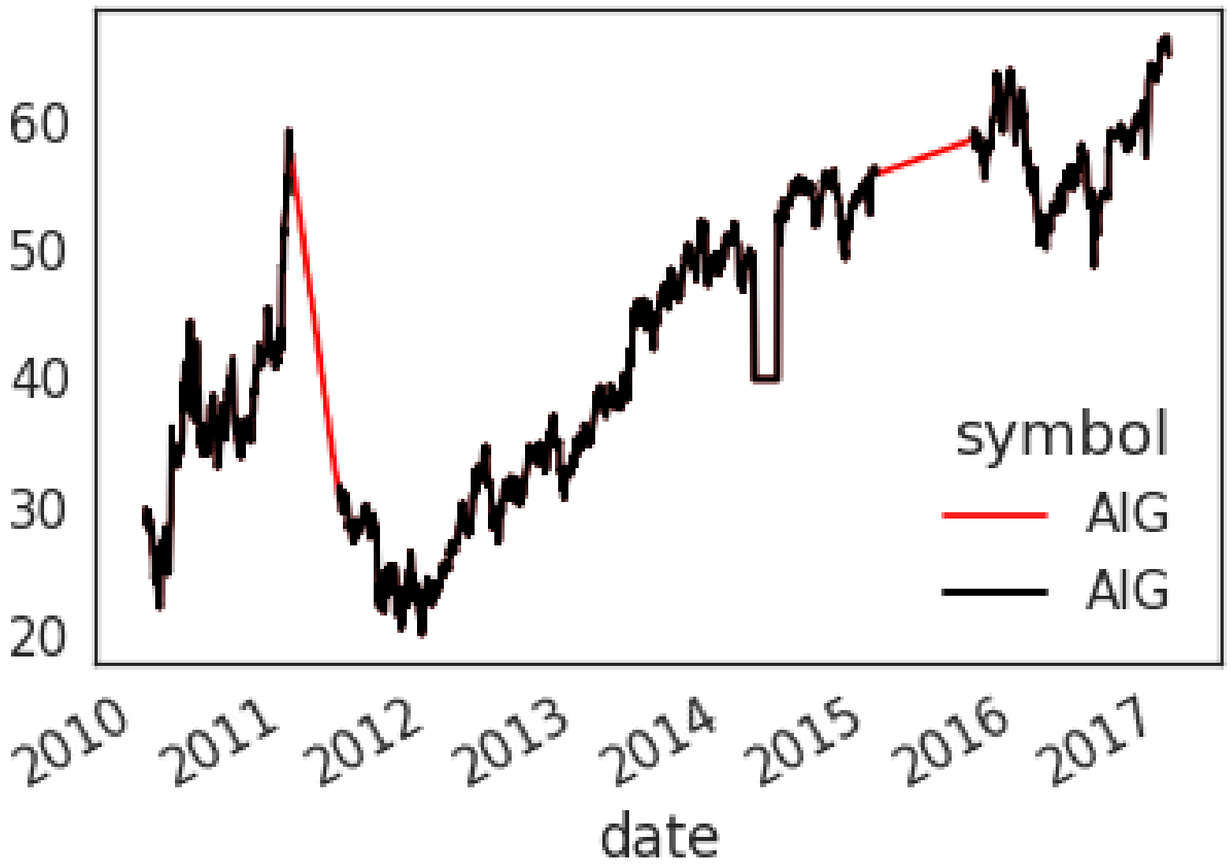
Interpolation: using time to fill in missing data

- A common way to deal with missing data is to *interpolate* missing values
- With timeseries data, you can use time to assist in interpolation.
- In this case, **interpolation** means using using the *known* values on either side of a gap in the data to make assumptions about what's missing.

Interpolation in Pandas

```
# Return a boolean that notes where missing values are
missing = prices.isna()

# Interpolate linearly within missing windows
prices_interp = prices.interpolate('linear')

# Plot the interpolated data in red and the data w/ missing values in black
ax = prices_interp.plot(c='r')
prices.plot(c='k', ax=ax, lw=2)
```

Visualizing the interpolated data



Using a rolling window to transform data

- Another common use of rolling windows is to transform the data
- We've already done this once, in order to *smooth* the data
- However, we can also use this to do more complex transformations

Transforming data to standardize variance

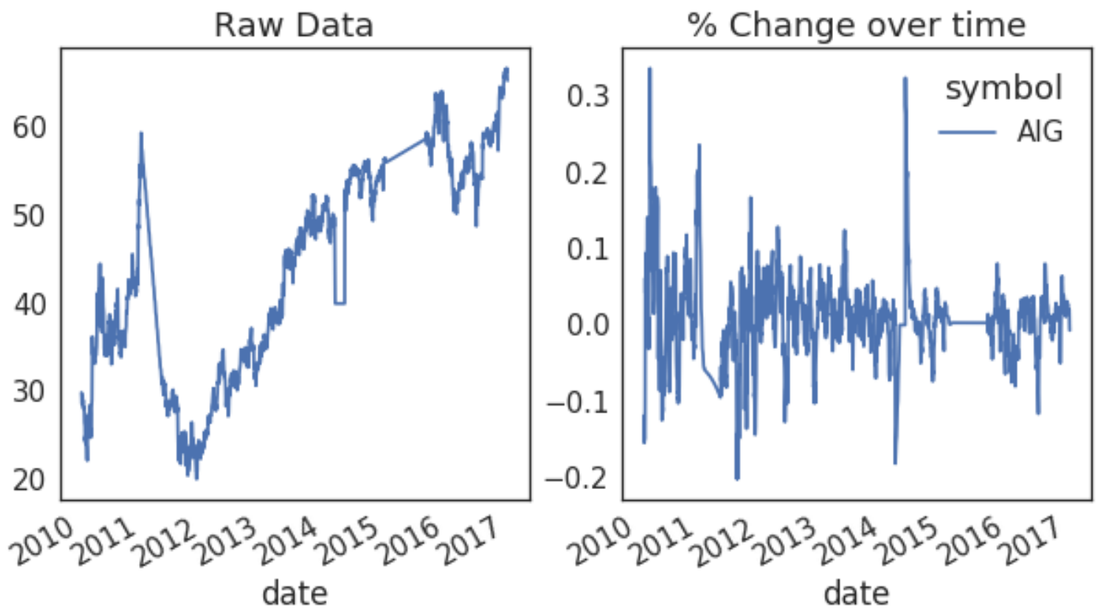
- A common transformation to apply to data is to standardize its mean and variance over time. There are many ways to do this.
- Here, we'll show how to convert your dataset so that each point represents the *% change over a previous window*.
- This makes timepoints more comparable to one another if the absolute values of data change a lot

Transforming to percent change with Pandas

```
def percent_change(values):
    """Calculates the % change between the last value
    and the mean of previous values"""
    # Separate the last value and all previous values into variables
    previous_values = values[:-1]
    last_value = values[-1]

    # Calculate the % difference between the last value
    # and the mean of earlier values
    percent_change = (last_value - np.mean(previous_values)) \
    / np.mean(previous_values)
    return percent_change
```

Applying this to our data

```
# Plot the raw data
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
ax = prices.plot(ax=axs[0])

# Calculate % change and plot
ax = prices.rolling(window=20).aggregate(percent_change).plot(ax=axs[1])
ax.legend_.set_visible(False)
```



Finding outliers in your data

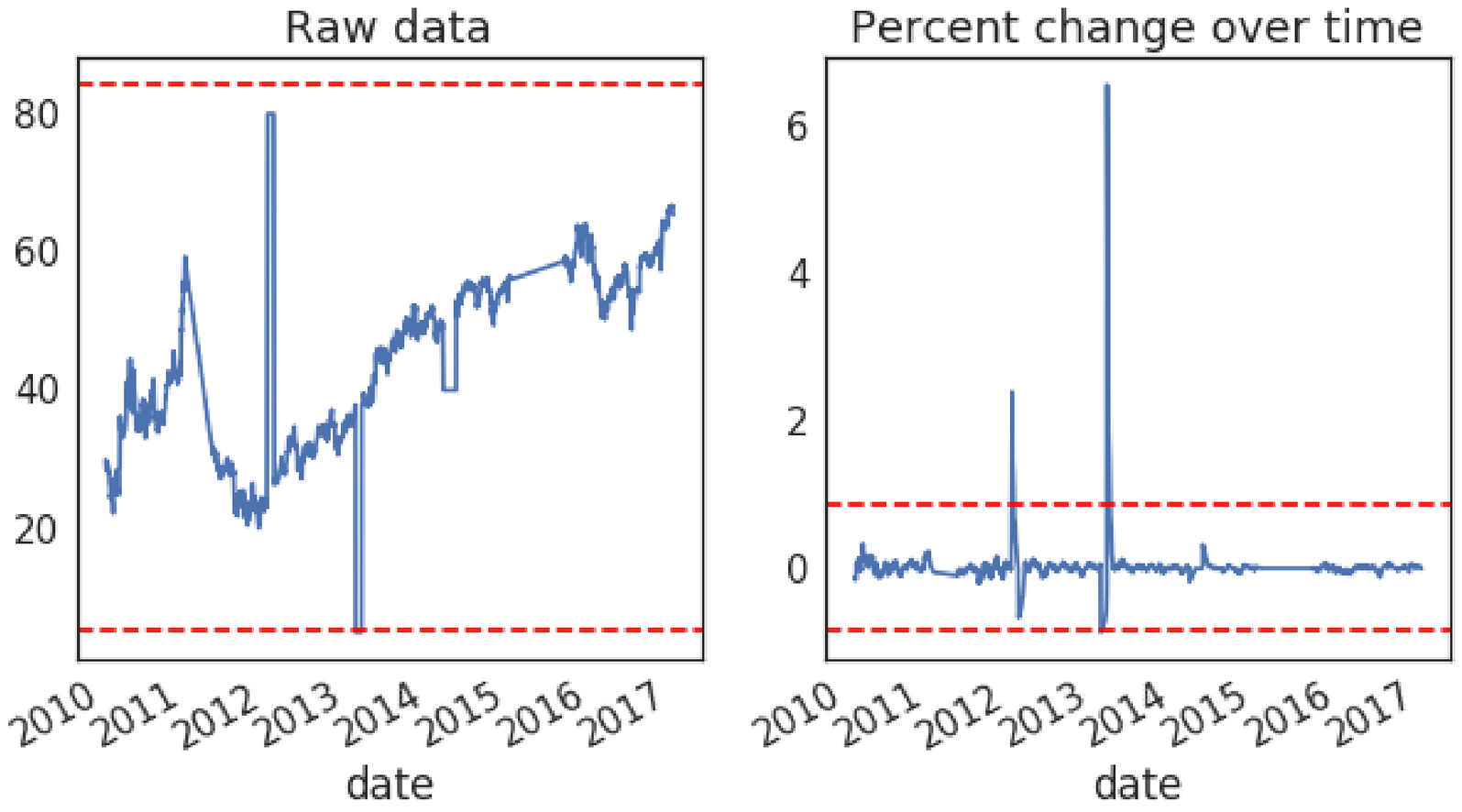
- Outliers are datapoints that are significantly statistically different from the dataset.
- They can have negative effects on the predictive power of your model, biasing it away from its "true" value
- One solution is to *remove* or *replace* outliers with a more representative value

> **Be very careful** about doing this - often it is difficult to determine what is a legitimately extreme value vs an abberation

Plotting a threshold on our data

```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
for data, ax in zip([prices, prices_perc_change], axs):
    # Calculate the mean / standard deviation for the data
    this_mean = data.mean()
    this_std = data.std()

    # Plot the data, with a window that is 3 standard deviations
    # around the mean
    data.plot(ax=ax)
    ax.axhline(this_mean + this_std * 3, ls='--', c='r')
    ax.axhline(this_mean - this_std * 3, ls='--', c='r')
```

Visualizing outlier thresholds

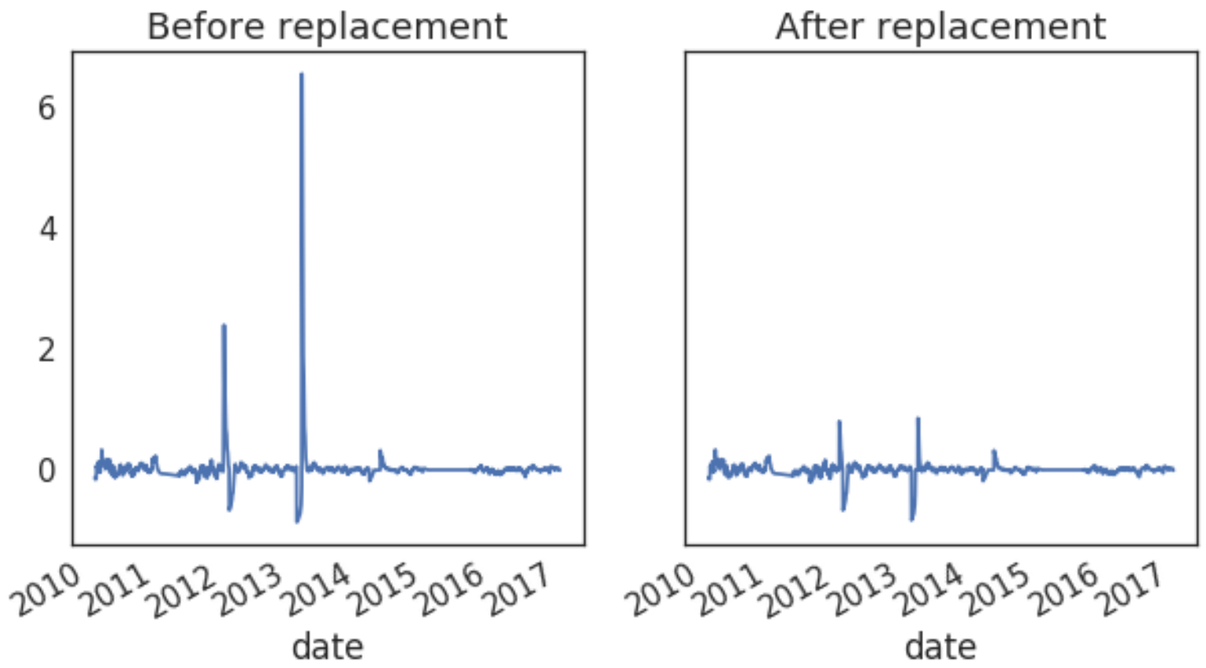


Replacing outliers using the threshold

```
# Center the data so the mean is 0
prices_outlier_centered = prices_outlier_perc - prices_outlier_perc.mean()

# Calculate standard deviation
std = prices_outlier_perc.std()

# Use the absolute value of each datapoint
# to make it easier to find outliers
outliers = np.abs(prices_outlier_centered) > (std * 3)

# Replace outliers with the median value
# We'll use np.nanmean since there may be nans around the outliers
prices_outlier_fixed = prices_outlier_centered.copy()
prices_outlier_fixed[outliers] = np.nanmedian(prices_outlier_fixed)
```

Visualize the results

```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
prices_outlier_centered.plot(ax=axs[0])
prices_outlier_fixed.plot(ax=axs[1])
```



Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

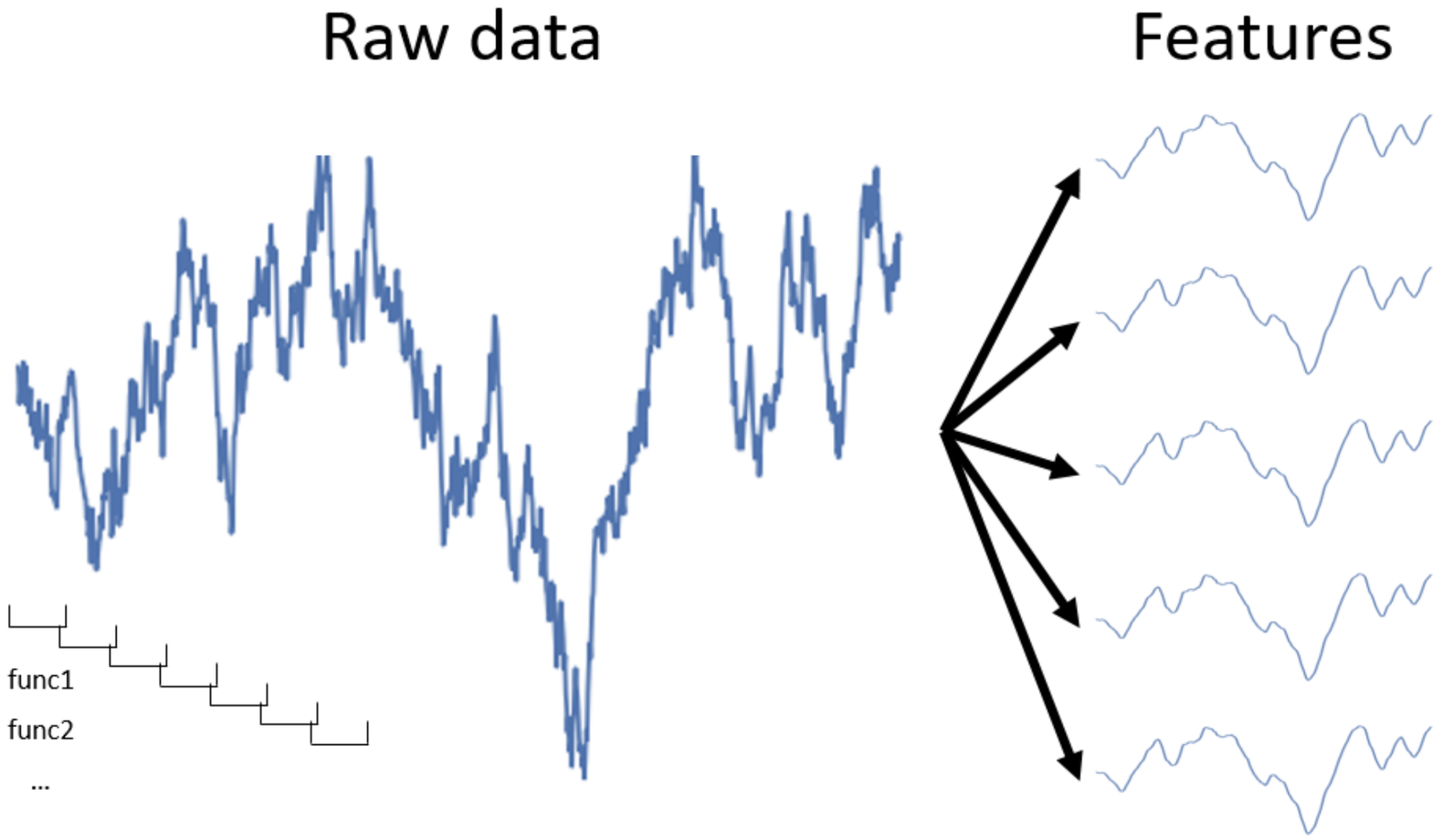
Creating features over time

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

Extracting features with windows



Using .aggregate for feature extraction

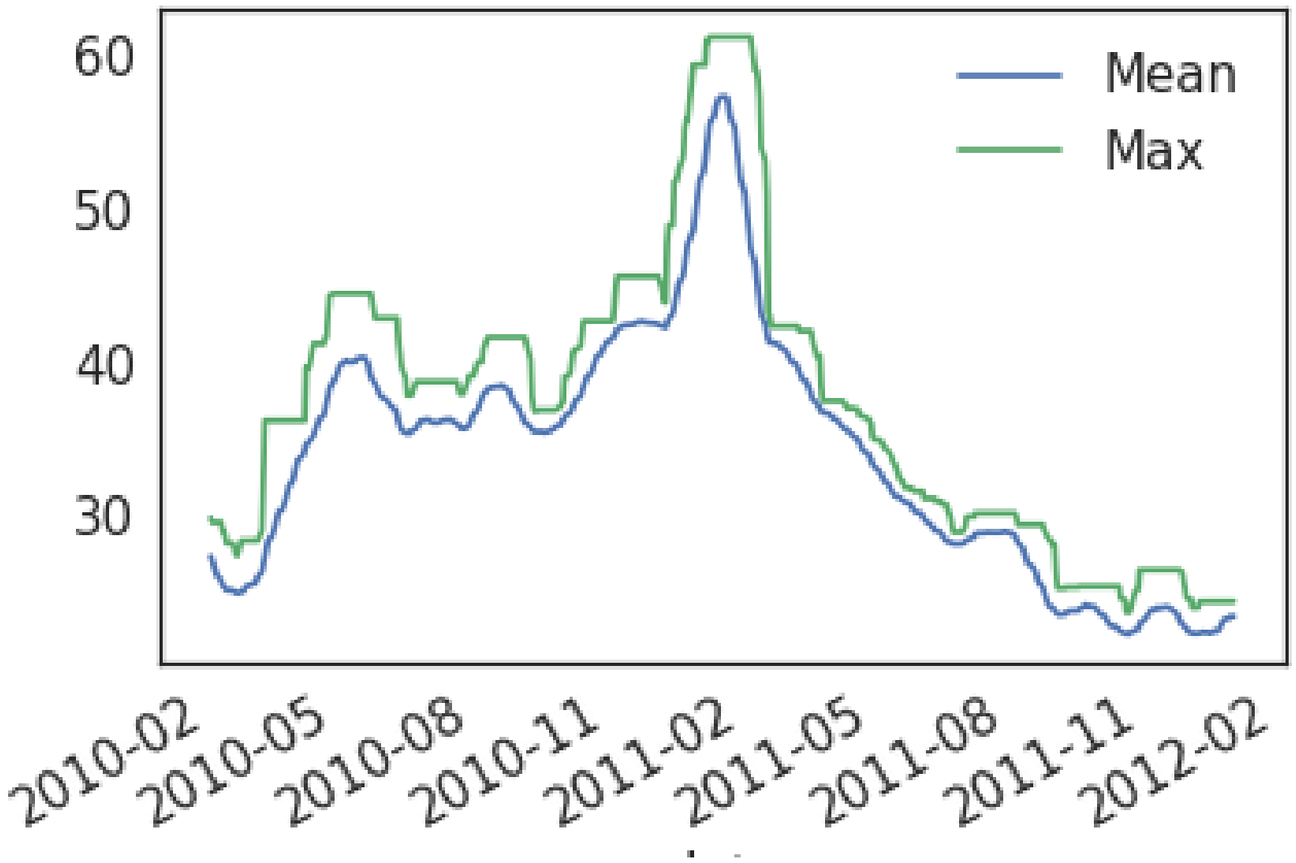
```
# Visualize the raw data
print(prices.head(3))
```

```
symbol             AIG        ABT
date
2010-01-04  29.889999  54.459951
2010-01-05  29.330000  54.019953
2010-01-06  29.139999  54.319953
```

```
# Calculate a rolling window, then extract two features
feats = prices.rolling(20).aggregate([np.std, np.max]).dropna()
print(feats.head(3))
```

```
                AIG                  ABT
                std       amax       std       amax
date
2010-02-01  2.051966  29.889999  0.868830  56.239949
2010-02-02  2.101032  29.629999  0.869197  56.239949
2010-02-03  2.157249  29.629999  0.852509  56.239949
```

Check the properties of your features!



Using partial() in Python

```
# If we just take the mean, it returns a single value
a = np.array([[0, 1, 2], [0, 1, 2], [0, 1, 2]])
print(np.mean(a))
```

```
1.0
```

```
# We can use the partial function to initialize np.mean
# with an axis parameter
from functools import partial
mean_over_first_axis = partial(np.mean, axis=0)

print(mean_over_first_axis(a))
```

```
[0. 1. 2.]
```

Percentiles summarize your data

- Percentiles are a useful way to get more fine-grained summaries of your data (as opposed to using `np.mean`)
- For a given dataset, the Nth percentile is the value where N% of the data is below that datapoint, and 100-N% of the data is above that datapoint.

```
print(np.percentile(np.linspace(0, 200), q=20))
```

```
40.0
```

Combining np.percentile() with partial functions to calculate a range of percentiles

```
data = np.linspace(0, 100)

# Create a list of functions using a list comprehension
percentile_funcs = [partial(np.percentile, q=ii) for ii in [20, 40, 60]]

# Calculate the output of each function in the same way
percentiles = [i_func(data) for i_func in percentile_funcs]
print(percentiles)
```

```
[20.0, 40.00000000000001, 60.0]
```

```
# Calculate multiple percentiles of a rolling window
data.rolling(20).aggregate(percentiles)
```

Calculating "date-based" features

- Thus far we've focused on calculating "statistical" features - these are features that correspond statistical properties of the data, like "mean", "standard deviation", etc
- However, don't forget that timeseries data often has more "human" features associated with it, like days of the week, holidays, etc.
- These features are often useful when dealing with timeseries data that spans multiple years (such as stock value over time)

datetime features using Pandas

```
# Ensure our index is datetime
prices.index = pd.to_datetime(prices.index)

# Extract datetime features
day_of_week_num = prices.index.weekday
print(day_of_week_num[:10])
```

```
Index([0 1 2 3 4 0 1 2 3 4], dtype='object')
```

```
day_of_week = prices.index.weekday_name
print(day_of_week[:10])
```

```
Index(['Monday' 'Tuesday' 'Wednesday' 'Thursday' 'Friday' 'Monday' 'Tuesday'
 'Wednesday' 'Thursday' 'Friday'], dtype='object')
```

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Time-delayed features and auto-regressive models

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

datacamp

The past is useful

- Timeseries data almost always have information that is *shared between timepoints*
- Information in the past can help predict what happens in the future
- Often the features best-suited to predict a timeseries are previous values of the same timeseries.

A note on smoothness and auto-correlation

- A common question to ask of a timeseries: how smooth is the data.
- AKA, how correlated is a timepoint with its neighboring timepoints (called **autocorrelation**).
- The amount of auto-correlation in data will impact your models.

Creating time-lagged features

- Let's see how we could build a model that uses values in the past as input features.
- We can use this to assess how auto-correlated our signal is (and lots of other stuff too)

Time-shifting data with Pandas

```
print(df)
```

```
     df
0   0.0
1   1.0
2   2.0
3   3.0
4   4.0
```

```
# Shift a DataFrame/Series by 3 index values towards the past
print(df.shift(3))
```

```
     df
0   NaN
1   NaN
2   NaN
3   0.0
4   1.0
```

Creating a time-shifted DataFrame

```
# data is a pandas Series containing time series data
data = pd.Series(...)

# Shifts
shifts = [0, 1, 2, 3, 4, 5, 6, 7]

# Create a dictionary of time-shifted data
many_shifts = {'lag_{}'.format(ii): data.shift(ii) for ii in shifts}

# Convert them into a dataframe
many_shifts = pd.DataFrame(many_shifts)
```

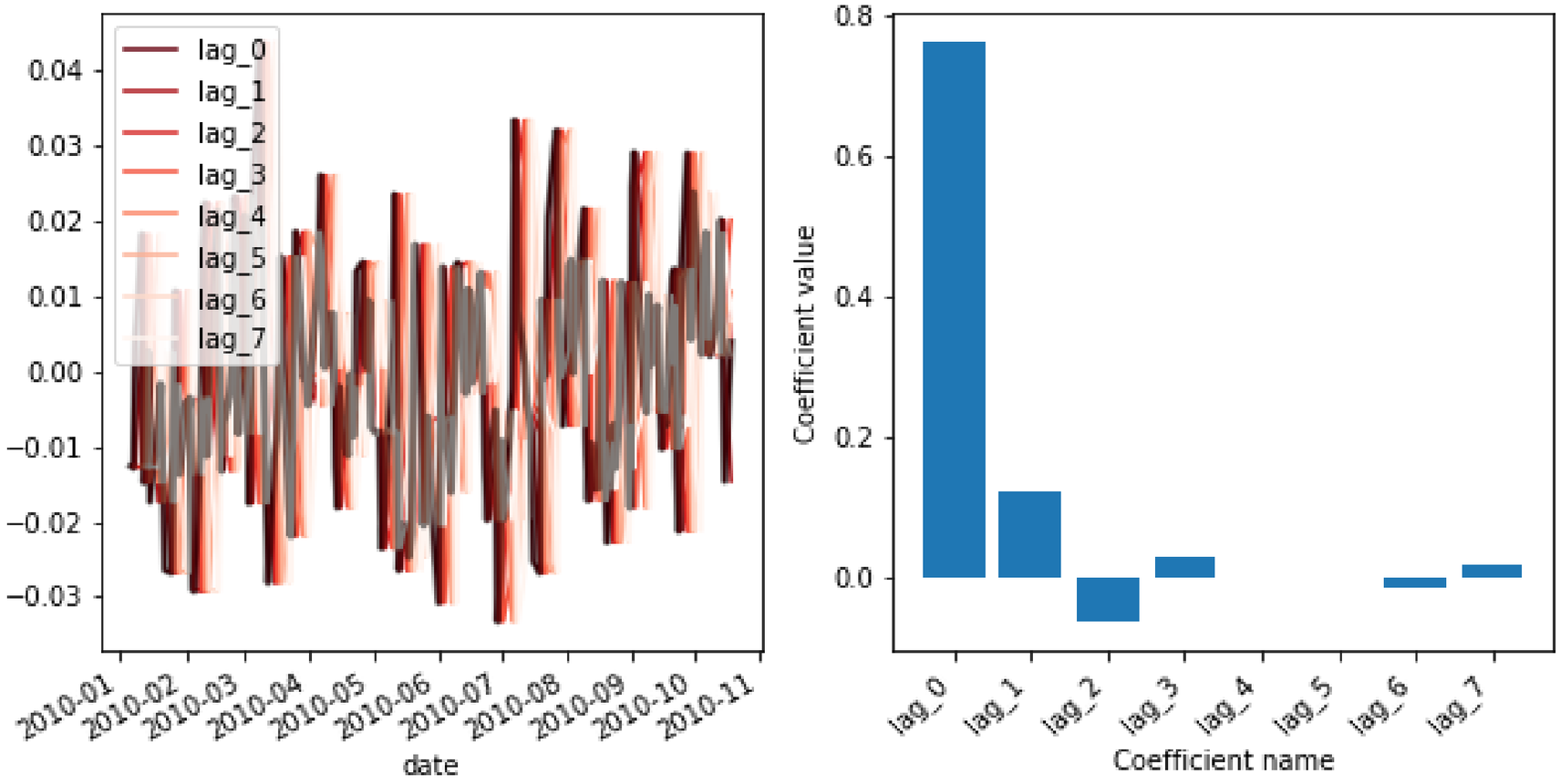
Fitting a model with time-shifted features

```
# Fit the model using these input features
model = Ridge()
model.fit(many_shifts, data)
```

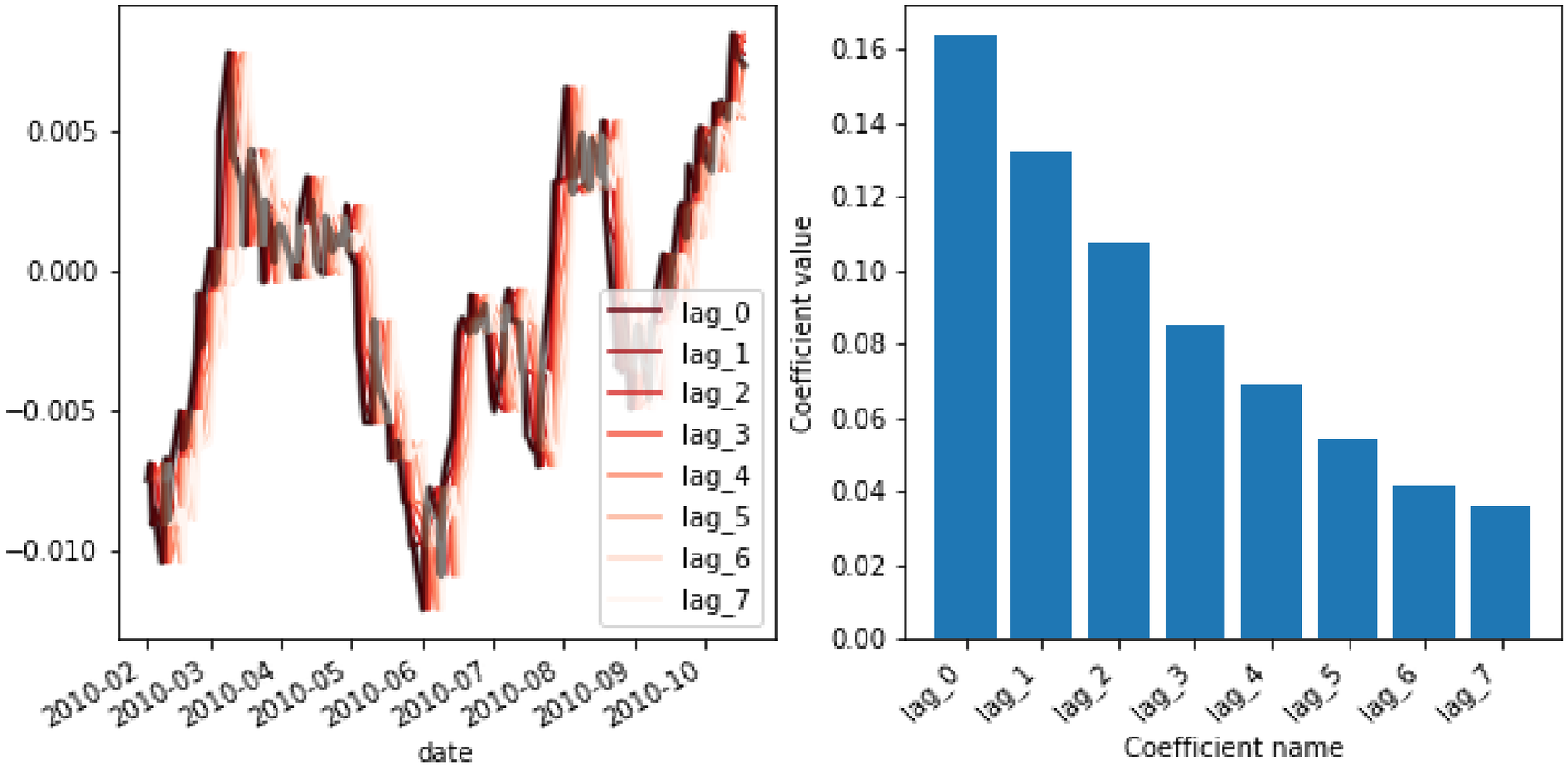
Interpreting the auto-regressive model coefficients

```
# Visualize the fit model coefficients
fig, ax = plt.subplots()
ax.bar(many_shifts.columns, model.coef_)
ax.set(xlabel='Coefficient name', ylabel='Coefficient value')

# Set formatting so it looks nice
plt.setp(ax.get_xticklabels(), rotation=45, horizontalalignment='right')
```

Visualizing coefficients for a rough signal



Visualizing coefficients for a smooth signal



Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Cross-validating timeseries data

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

datacamp

Cross validation with scikit-learn

```
# Iterating over the "split" method yields train/test indices
for tr, tt in cv.split(X, y):
    model.fit(X[tr], y[tr])
    model.score(X[tt], y[tt])
```

Cross validation types: KFold

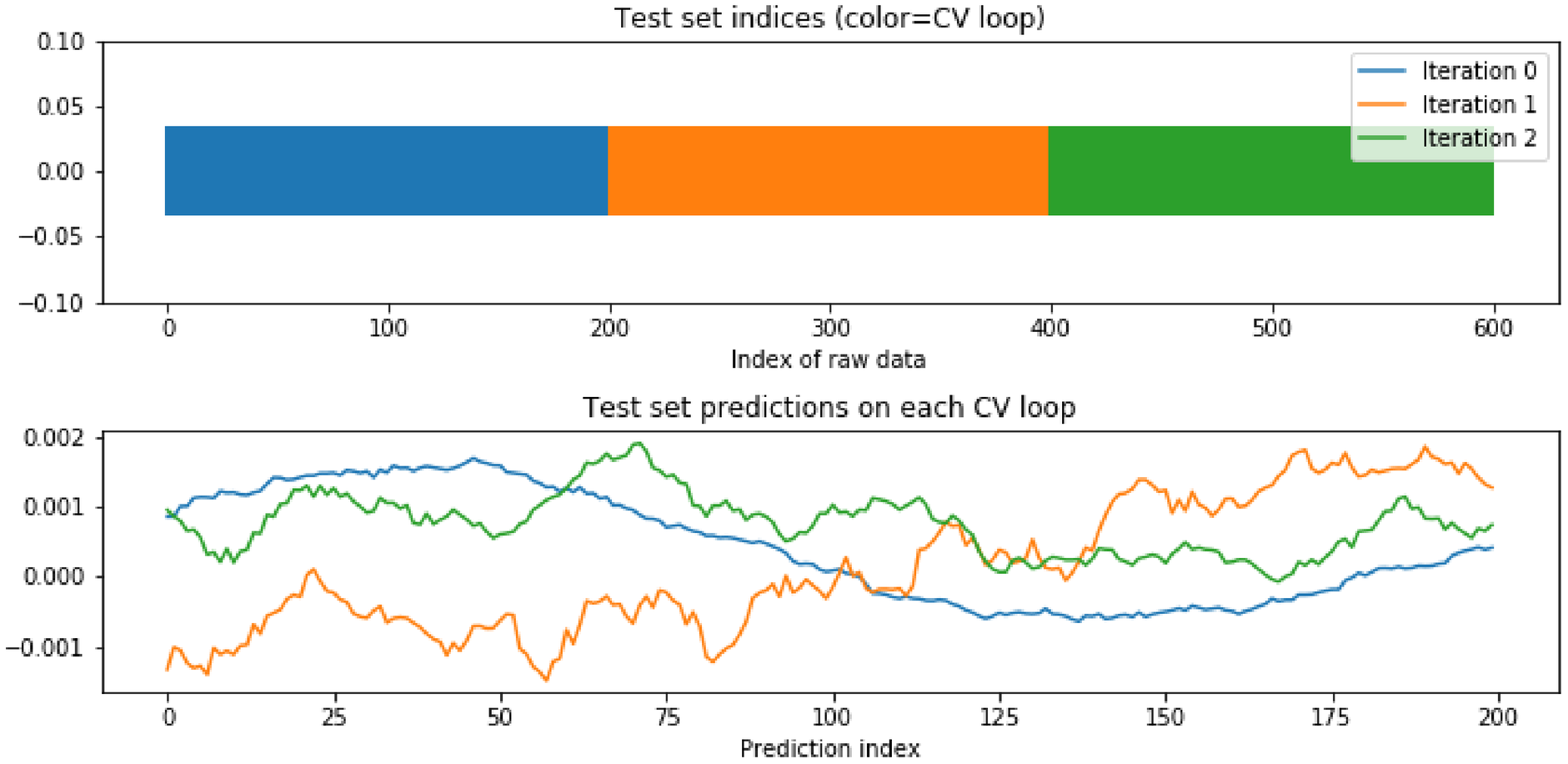
- `KFold` cross-validation splits your data into multiple "folds" of equal size
- It is one of the most common cross-validation routines

```
from sklearn.model_selection import KFold
cv = KFold(n_splits=5)
for tr, tt in cv.split(X, y):
    ...
```

Visualizing model predictions

```
fig, axs = plt.subplots(2, 1)

# Plot the indices chosen for validation on each loop
axs[0].scatter(tt, [0] * len(tt), marker='_', s=2, lw=40)
axs[0].set(ylim=[-.1, .1], title='Test set indices (color=CV loop)',
           xlabel='Index of raw data')

# Plot the model predictions on each iteration
axs[1].plot(model.predict(X[tt]))
axs[1].set(title='Test set predictions on each CV loop',
           xlabel='Prediction index')
```

Visualizing KFold CV behavior



A note on shuffling your data

- Many CV iterators let you shuffle data as a part of the cross-validation process.
- This only works if the data is i.i.d., which timeseries usually is **not**.
- You should *not* shuffle your data when making predictions with timeseries.

```
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=3)
for tr, tt in cv.split(X, y):
    ...
```

Visualizing shuffled CV behavior

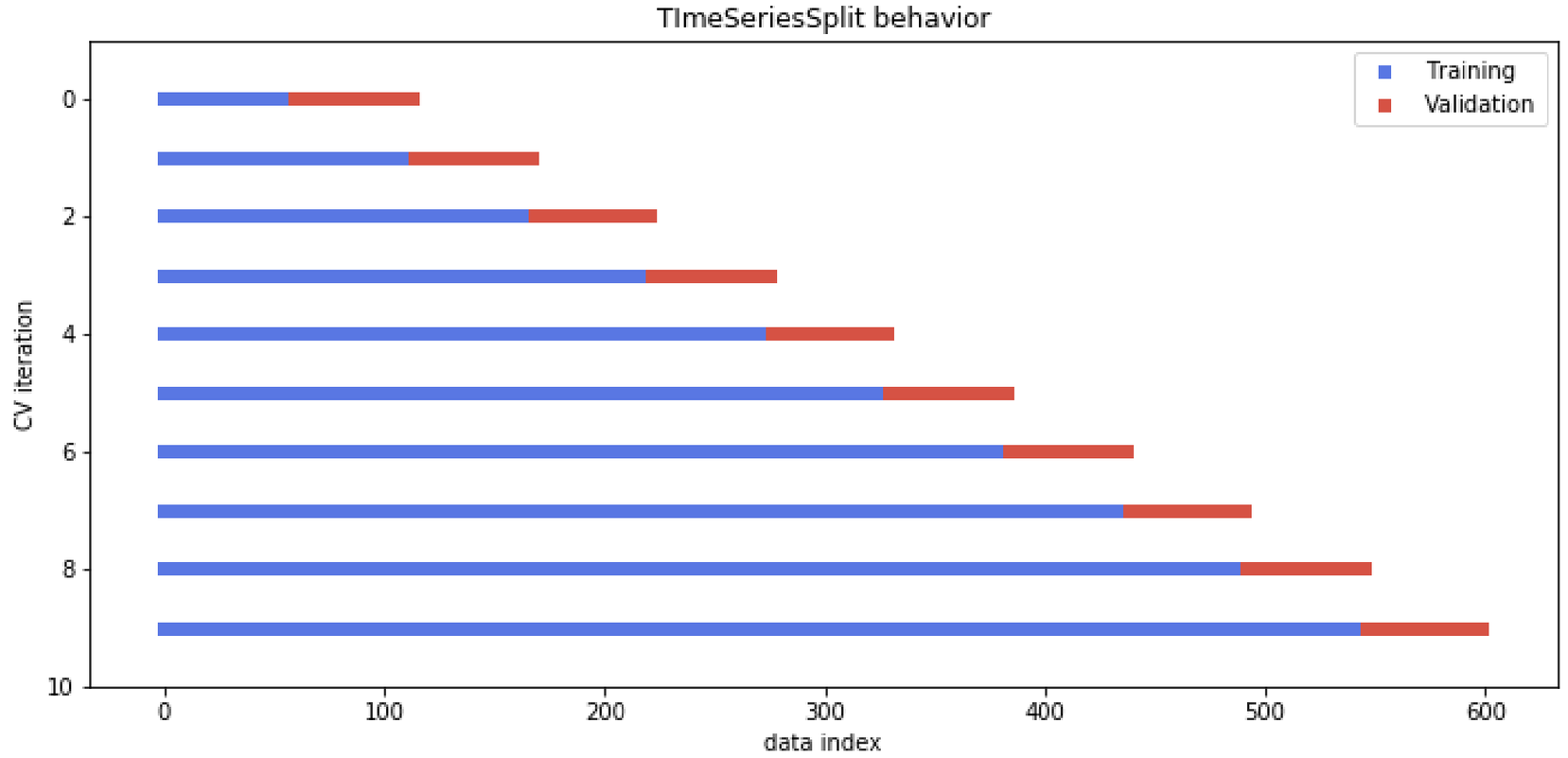
Using the time series CV iterator

- Thus far, we've broken the linear passage of time in the cross validation
- However, you generally **should not** use datapoints in the future to predict data in the past
- One approach: Always use training data from the **past** to predict the **future**

Visualizing time series cross validation iterators

```
# Import and initialize the cross-validation iterator
from sklearn.model_selection import TimeSeriesSplit
cv = TimeSeriesSplit(n_splits=10)

fig, ax = plt.subplots(figsize=(10, 5))
for ii, (tr, tt) in enumerate(cv.split(X, y)):
    # Plot training and test indices
    l1 = ax.scatter(tr, [ii] * len(tr), c=[plt.cm.coolwarm(.1)],
                    marker='_', lw=6)
    l2 = ax.scatter(tt, [ii] * len(tt), c=[plt.cm.coolwarm(.9)],
                    marker='_', lw=6)
    ax.set(ylim=[10, -1], title='TimeSeriesSplit behavior',
           xlabel='data index', ylabel='CV iteration')
    ax.legend([l1, l2], ['Training', 'Validation'])
```

Visualizing the TimeSeriesSplit cross validation iterator



Custom scoring functions in scikit-learn

```
def myfunction(estimator, X, y):
    y_pred = estimator.predict(X)
    my_custom_score = my_custom_function(y_pred, y)
    return my_custom_score
```

A custom correlation function for scikit-learn

```
def my_pearsonr(est, X, y):
    # Generate predictions and convert to a vector
    y_pred = est.predict(X).squeeze()

    # Use the numpy "corrcoef" function to calculate a correlation matrix
    my_corrcoef_matrix = np.corrcoef(y_pred, y.squeeze())

    # Return a single correlation value from the matrix
    my_corrcoef = my_corrcoef[1, 0]
    return my_corrcoef
```

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Stationarity and stability

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

datacamp

Stationarity

- Stationary time series do not change their statistical properties over time
- E.g., mean, standard deviation, trends
- Most time series are non-stationary to some extent


A stationary(ish) time series
2011-08
2011-09
2011-10
2011-11
2011-12
2012-01
A non-stationary time series
2010-11
2011-01
2011-03
2011-05
2011-07
2011-09
2011-11
2012-01
date

Model stability

- Non-stationary data results in variability in our model
- The statistical properties the model finds may change with the data
- In addition, we will be less certain about the correct values of model parameters
- How can we quantify this?

Cross validation to quantify parameter stability

- One approach: use cross-validation
- Calculate model parameters on each iteration
- Assess parameter stability across all CV splits

Bootstrapping the mean

- Bootstrapping is a common way to assess variability
- The bootstrap:
  1. Take a random sample of data **with replacement**
  2. Calculate the mean of the sample
  3. Repeat this process many times (1000s)
  4. Calculate the percentiles of the result (usually 2.5, 97.5)

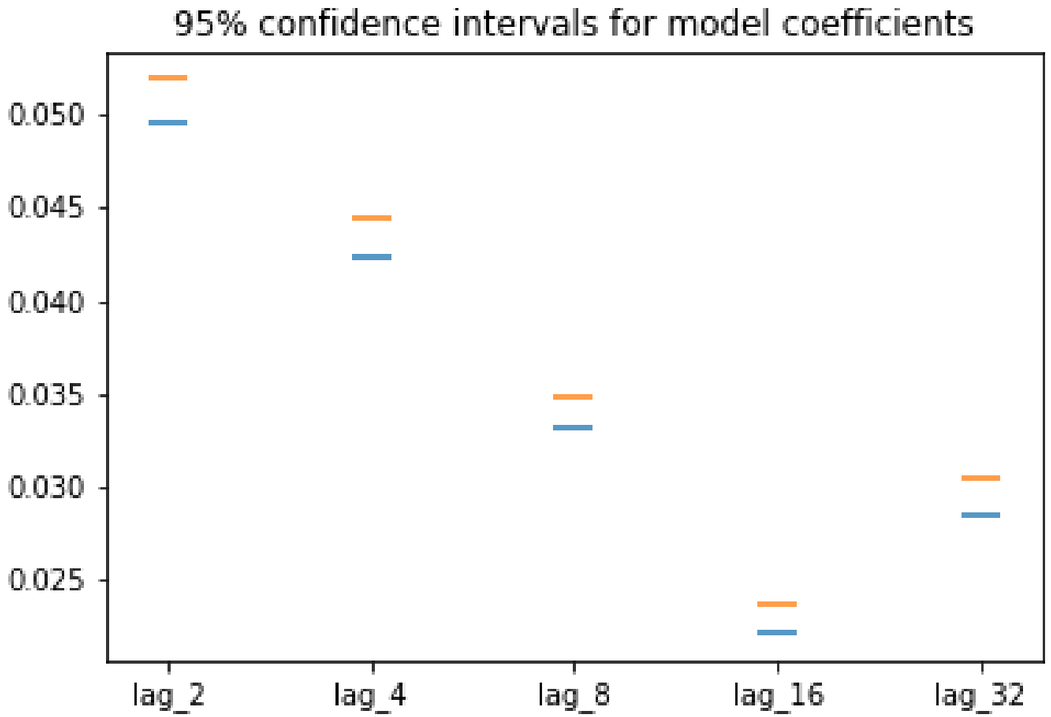
The result is a *95% confidence interval* of the mean of each coefficient.

Bootstrapping the mean

```
from sklearn.utils import resample

# cv_coefficients has shape (n_cv_folds, n_coefficients)
n_boots = 100
bootstrap_means = np.zeros(n_boots, n_coefficients)
for ii in range(n_boots):
    # Generate random indices for our data with replacement,
    # then take the sample mean
    random_sample = resample(cv_coefficients)
    bootstrap_means[ii] = random_sample.mean(axis=0)

# Compute the percentiles of choice for the bootstrapped means
percentiles = np.percentile(bootstrap_means, (2.5, 97.5), axis=0)
```

Plotting the bootstrapped coefficients

```
fig, ax = plt.subplots()
ax.scatter(many_shifts.columns, percentiles[0], marker='_', s=200)
ax.scatter(many_shifts.columns, percentiles[1], marker='_', s=200)
```



Assessing model performance stability

- If using the TimeSeriesSplit, can *plot* the model's score over time.
- This is useful in finding certain regions of time that hurt the score
- Also useful to find non-stationary signals

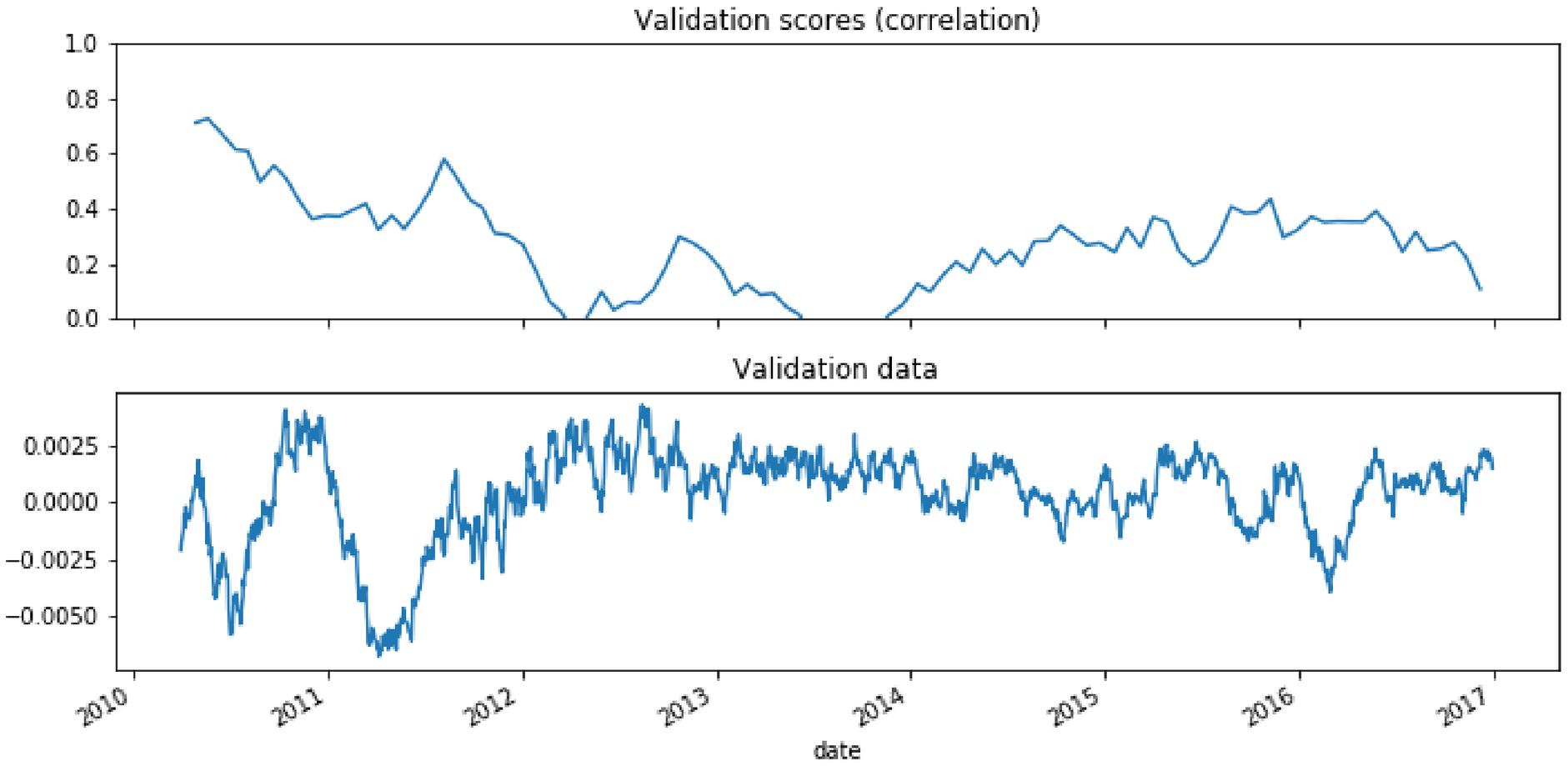
Model performance over time

```
def my_corrcoef(est, X, y):
    """Return the correlation coefficient
    between model predictions and a validation set."""
    return np.corrcoef(y, est.predict(X))[1, 0]

# Grab the date of the first index of each validation set
first_indices = [data.index[tt[0]] for tr, tt in cv.split(X, y)]

# Calculate the CV scores and convert to a Pandas Series
cv_scores = cross_val_score(model, X, y, cv=cv, scoring=my_corrcoef)
cv_scores = pd.Series(cv_scores, index=first_indices)
```

Visualizing model scores as a timeseries

```
fig, axs = plt.subplots(2, 1, figsize=(10, 5), sharex=True)

# Calculate a rolling mean of scores over time
cv_scores_mean = cv_scores.rolling(10, min_periods=1).mean()
cv_scores.plot(ax=axs[0])
axs[0].set(title='Validation scores (correlation)', ylim=[0, 1])

# Plot the raw data
data.plot(ax=axs[1])
axs[1].set(title='Validation data')
```

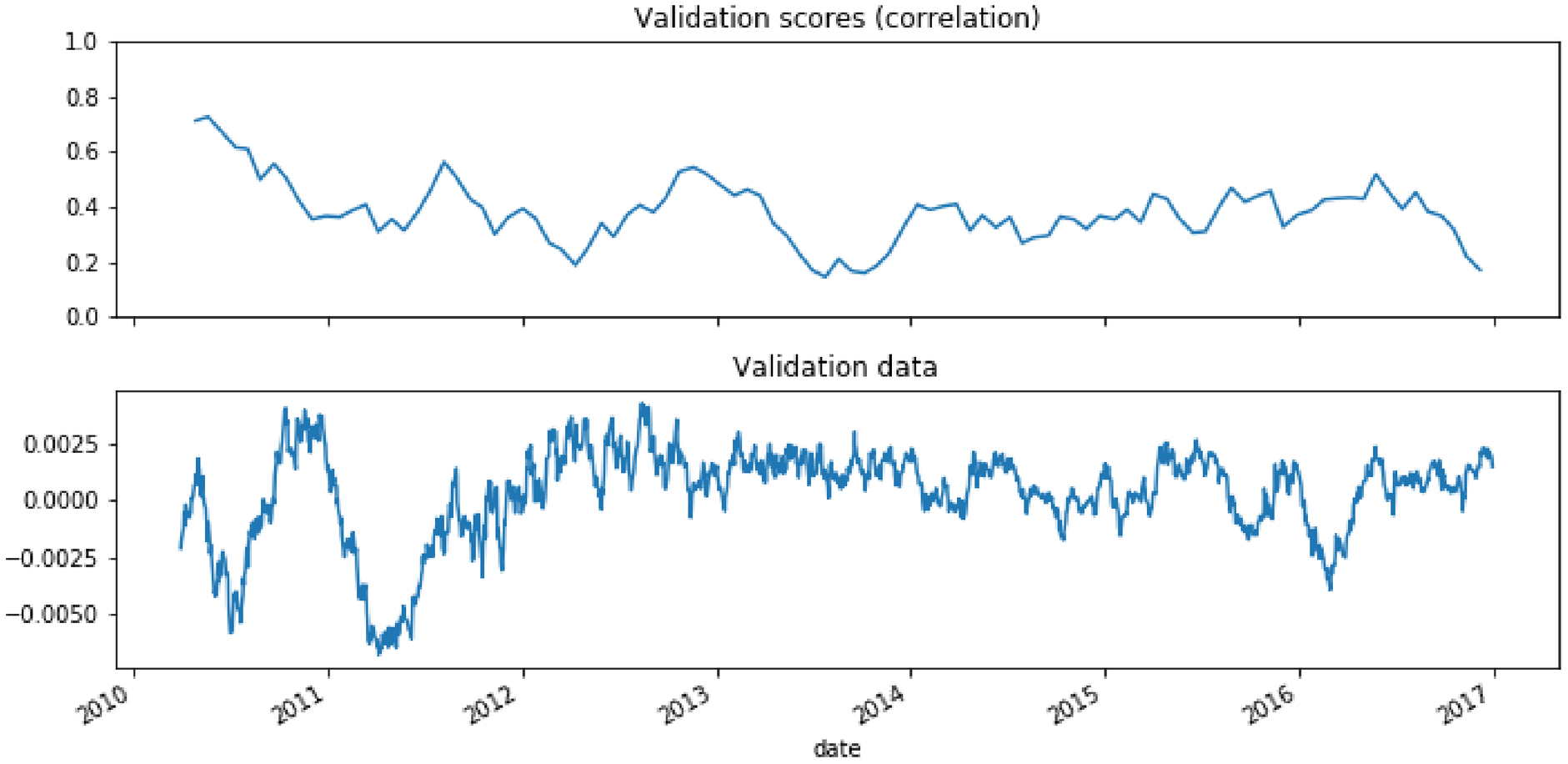
Visualizing model scores



Fixed windows with time series cross-validation

```
# Only keep the last 100 datapoints in the training data
window = 100

# Initialize the CV with this window size
cv = TimeSeriesSplit(n_splits=10, max_train_size=window)
```

Non-stationary signals



Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Wrapping-up

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf

Fellow, Berkeley Institute for Data Science

Timeseries and machine learning

- The many applications of time series + machine learning
- Always visualize your data first
- The scikit-learn API standardizes this process

Feature extraction and classification

- Summary statistics for time series classification
- Combining multiple features into a single input matrix
- Feature extraction for time series data

Model fitting and improving data quality

- Time series features for regression
- Generating predictions over time
- Cleaning and improving time series data

Validating and assessing our model performance

- Cross-validation with time series data (don't shuffle the data!)
- Time series stationarity
- Assessing model coefficient and score stability

Advanced concepts in time series

- Advanced window functions
- Signal processing and filtering details
- Spectral analysis

Advanced machine learning

- Advanced time series feature extraction (e.g., `tsfresh`)
- More complex model architectures for regression and classification
- Production-ready pipelines for time series analysis

Ways to practice

- There are a lot of opportunities to practice your skills with time series data.
- Kaggle has a number of time series predictions challenges
- Quantopian is also useful for learning and using predictive models others have built.

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON