Machine Learning for Time Series Data in Python

How is your ML model handling time series data?

Black Raven (James Ng)

24 Oct 2020 · 40 min read

This is a memo to share what I have learnt in Machine Learning for Time Series Data (using Python), capturing the learning objectives as well as my personal notes. The course is taught by Chris Holdgraf from DataCamp, and it includes 4 chapters:

Chapter 1. Time Series and Machine Learning Primer

Chapter 2. Time Series as Inputs to a Model

Chapter 3. Predicting Time Series Data

Chapter 4. Validating and Inspecting Time Series Models

Time series data is ubiquitous. Whether it be stock market fluctuations, sensor data recording climate change, or activity in the brain, any signal that changes over time can be described as a time series. Machine learning has emerged as a powerful method for leveraging complexity in data in order to generate predictions and insights into the problem one is trying to solve.

Photo by Hossam M. Omar on Unsplash

This course is an intersection between these two worlds of machine learning and time series data, and covers feature engineering, spectograms, and other advanced techniques in order to classify heartbeat sounds and predict stock prices.

Chapter 1. Time Series and Machine Learning Primer

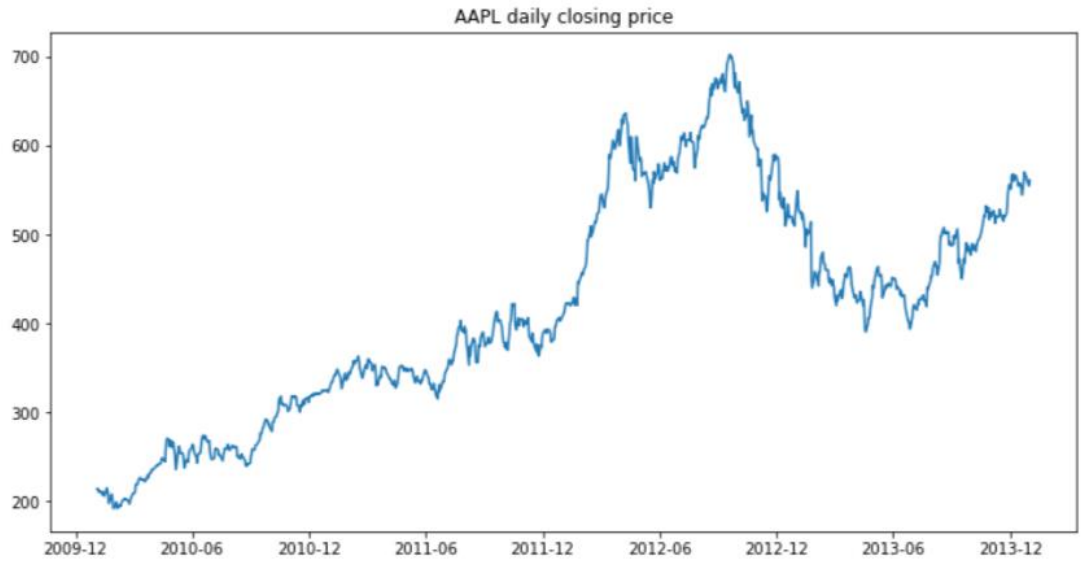
This chapter is an introduction to the basics of machine learning, time series data, and the intersection between the two.

Timeseries kinds and applications

Time Series = data that changes over time, data point with corresponding time point, example:

- atmospheric CO_2 over time
- waveform of voice (speaking)
- fluctuation of stock's value over the year
- demographic information about a city

```
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 machine learning pipeline:

- Feature extraction
- Model fitting
- Prediction and validation

Identifying a time series

Which of the following data sets is **not** considered time series data?

- ○ Test grades for the last fall and spring semesters of high-school students.
- ○ A student's attendance record each week of the semester.
- ○ The school's national annual ranking since 2000.
- ◉ A list of the average length of each class at the school.

Answer: A list of the average length of each class at the school. You don't have timestamps for each data point, so it is not a time series.

Plotting a time series (I)

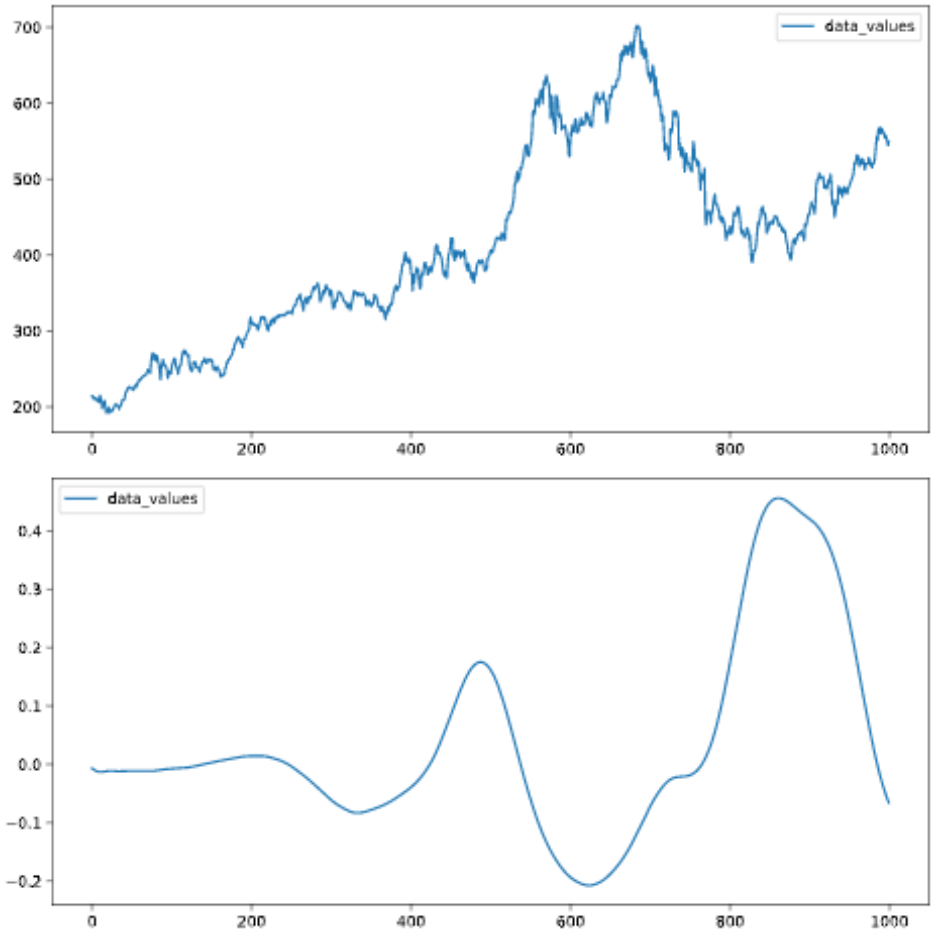
In this exercise, you'll practice plotting the values of two time series without the time component.

Two DataFrames, `data` and `data2` are available in your workspace.

Unless otherwise noted, assume that all required packages are loaded with their common aliases throughout this course.

Note: This course assumes some familiarity with time series data, as well as how to use them in data analytics pipelines. For an introduction to time series, we recommend the **Introduction to Time Series Analysis in Python** and **Visualizing Time Series Data with Python** courses.

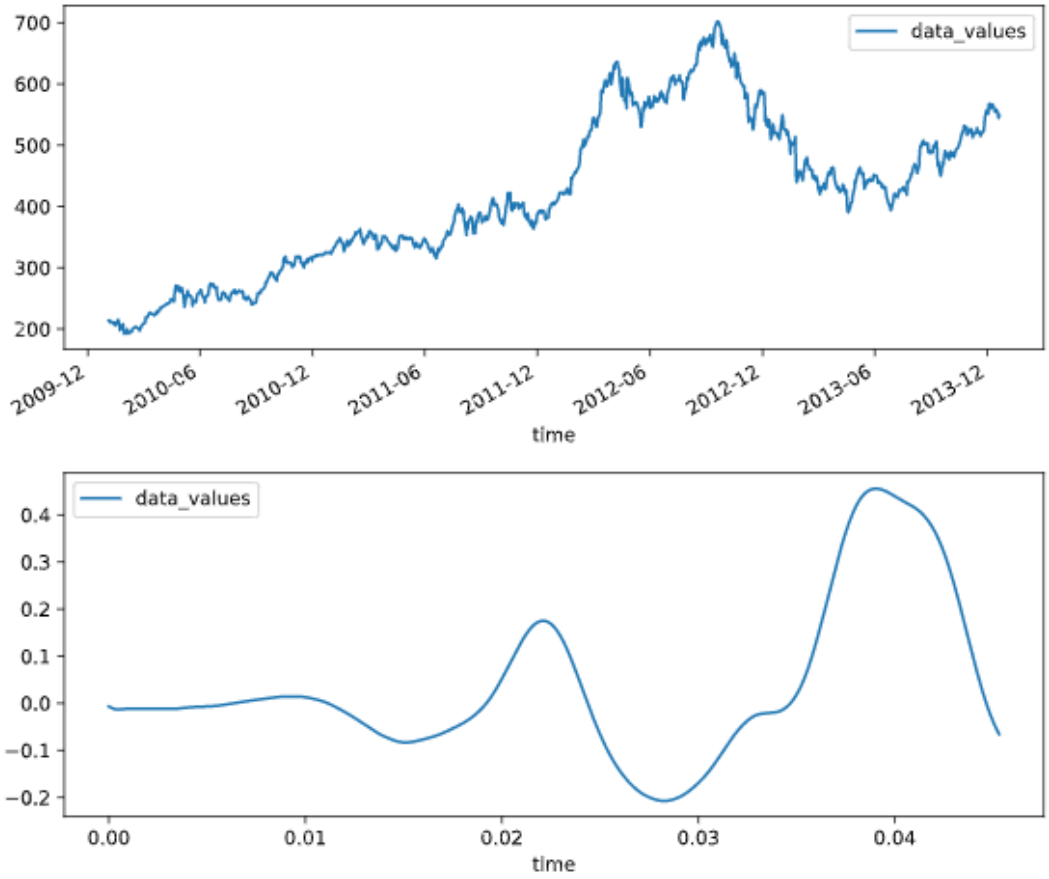
```
# Print the first 5 rows of data
print(data.head())

# Print the first 5 rows of data2
print(data2.head())

# Plot the time series in each dataset
fig, axs = plt.subplots(2, 1, figsize=(5, 10))
data.iloc[:1000].plot(y='data_values', ax=axs[0])
data2.iloc[:1000].plot(y='data_values', ax=axs[1])
plt.show()
```



Plotting a time series (II)

You'll now plot both the datasets again, but with the included time stamps for each (stored in the column called `"time"`). Let's see if this gives you some more context for understanding each time series data.

```
# Plot the time series in each dataset
fig, axs = plt.subplots(2, 1, figsize=(5, 10))
data.iloc[:1000].plot(x='time', y='data_values', ax=axs[0])
data2.iloc[:1000].plot(x='time', y='data_values', ax=axs[1])
plt.show()
```



As you can now see, each time series has a very different sampling frequency (the amount of time between samples).

The first is daily stock market data, and the second is an audio waveform.

Machine learning basics

Always visualise your data:
df.shape
df.head()
df.plot(..., ax=ax)

Preparing data for scikit-learn:
Scikit-learn expects a particular structure of data: (samples, features), at least 2-dimensional

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)
```

Investigating the model:
model.coef_

Predicting with a fit model:

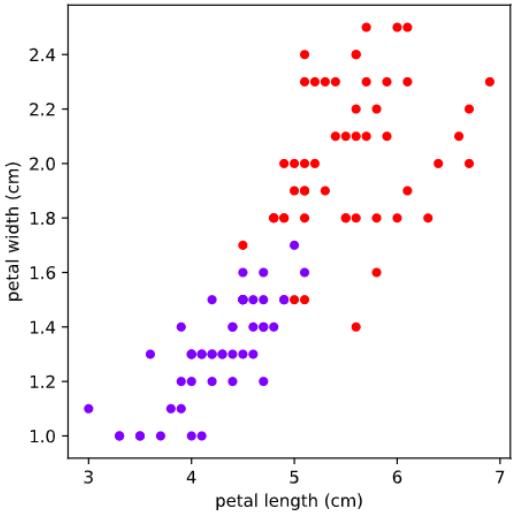
```
# Generate predictions
predictions = model.predict(X_test)
```

Fitting a simple model: classification

In this exercise, you'll use the iris dataset (representing petal characteristics of a number of flowers) to practice using the scikit-learn API to fit a classification model. You can see a sample plot of the data to the right.

Note: This course assumes some familiarity with Machine Learning and `scikit-learn`. For an introduction to scikit-learn, we recommend the **Supervised Learning with Scikit-Learn** and **Preprocessing for Machine Learning in Python** courses.

```
# Print the first 5 rows for inspection
print(data.head())

from sklearn.svm import LinearSVC

# Construct data for the model
X = data[['petal length (cm)', 'petal width (cm)']]
y = data[['target']]

# Fit the model
model = LinearSVC()
model.fit(X, y)
```



You've successfully fit a classifier to predict flower type!

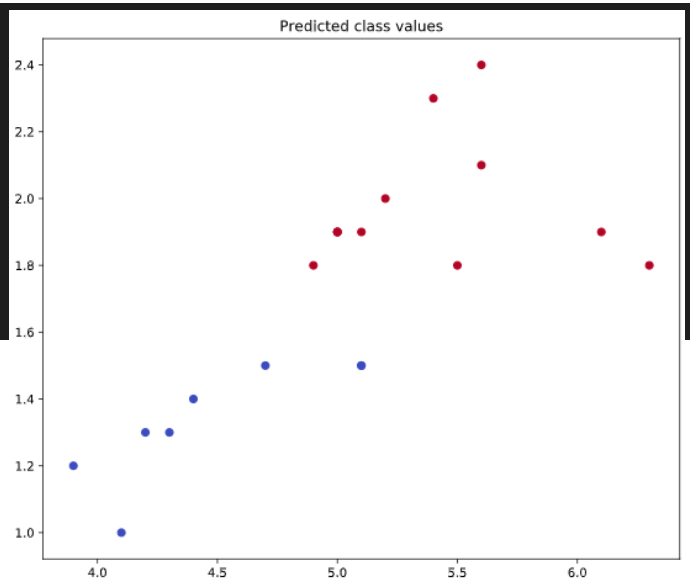
Predicting using a classification model

Now that you have fit your classifier, let's use it to predict the type of flower (or class) for some newly-collected flowers.

Information about petal width and length for several new flowers is stored in the variable `targets`. Using the classifier you fit, you'll predict the type of each flower.

```
# Create input array
X_predict = targets[['petal length (cm)', 'petal width (cm)']]

# Predict with the model
```

```
predictions = model.predict(X_predict)
print(predictions)

# Visualize predictions and actual values
plt.scatter(X_predict['petal length (cm)'], X_predict['petal width (cm)'],
            c=predictions, cmap=plt.cm.coolwarm)
plt.title("Predicted class values")
plt.show()
```



Note that the output of your predictions are all integers,

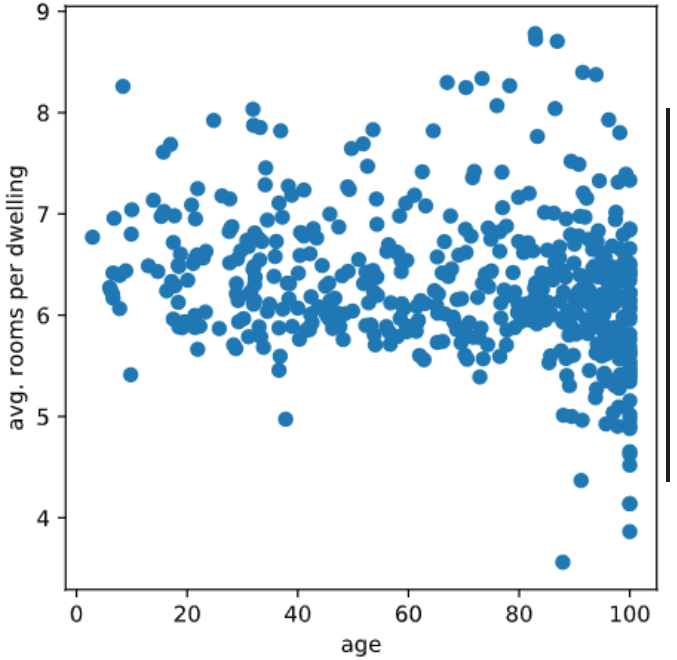
representing that datapoint's predicted class.

Fitting a simple model: regression

In this exercise, you'll practice fitting a regression model using data from the Boston housing market. A DataFrame called `boston` is available in your workspace. It contains many variables of data (stored as columns). Can you find a relationship between the following two variables?

- `"AGE"`: proportion of owner-occupied units built prior to 1940
- `"RM"` : average number of rooms per dwelling

```
from sklearn import linear_model

# Prepare input and output DataFrames
X = boston[['AGE']]
y = boston[['RM']]

# Fit the model
model = linear_model.LinearRegression()
model.fit(X, y)
```



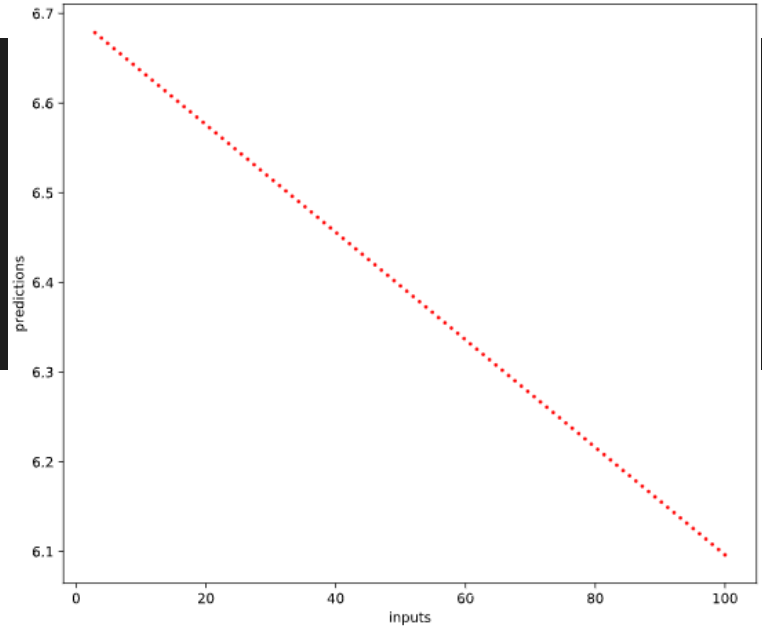
In regression, the output of your model is a continuous array of numbers, not class identity.

Predicting using a regression model

Now that you've fit a model with the Boston housing data, lets see what predictions it generates on some new data. You can investigate the underlying relationship that the model has found between inputs and outputs by feeding in a range of numbers as inputs and seeing what the model predicts for each input.

A 1-D array `new_inputs` consisting of 100 "new" values for `"AGE"` (proportion of owner-occupied units built prior to 1940) is available in your workspace along with the `model` you fit in the previous exercise.

```
# Generate predictions with the model using those inputs
predictions = model.predict(new_inputs.reshape(-1, 1))

# Visualize the inputs and predicted values
plt.scatter(new_inputs, predictions, color='r', s=3)
plt.xlabel('inputs')
plt.ylabel('predictions')
plt.show()
```



Here the red line shows the relationship that your model found.

As the proportion of pre-1940s houses gets larger,

the average number of rooms gets slightly lower.

Machine learning and time series data

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.

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*-1th 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

```
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
```

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

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

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

df['date']
```

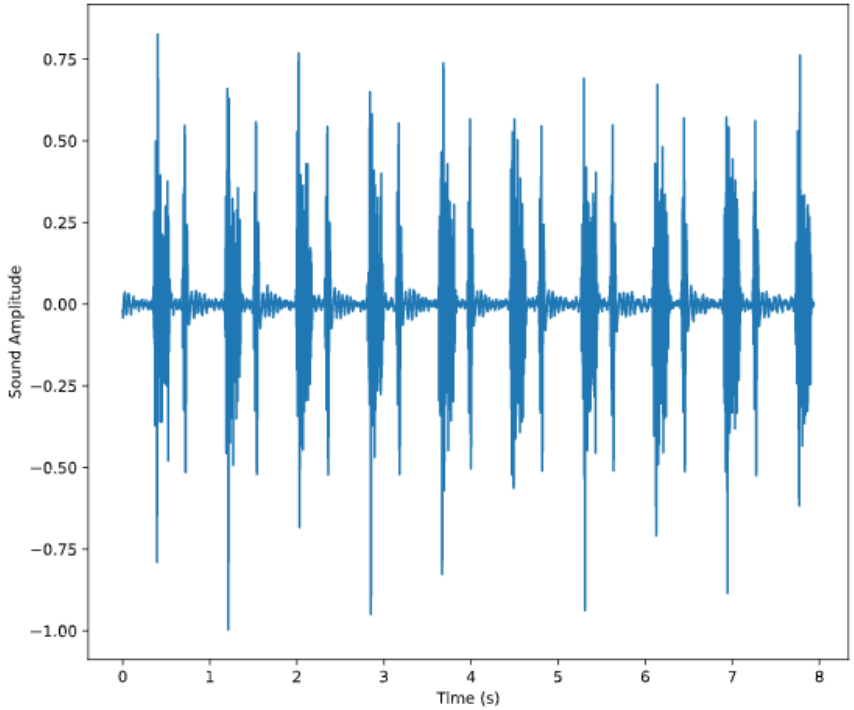
```
0   2017-01-01
1   2017-01-02
2   2017-01-03
Name: date, dtype: datetime64[ns]
```

Inspecting the classification data

In these final exercises of this chapter, you'll explore the two datasets you'll use in this course.

The first is a collection of heartbeat sounds. Hearts normally have a predictable sound pattern as they beat, but some disorders can cause the heart to beat *abnormally*. This dataset contains a *training* set with labels for each type of heartbeat, and a *testing* set with no labels. You'll use the *testing* set to validate your models.
As you have labeled data, this dataset is ideal for *classification*. In fact, it was originally offered as a part of a **public Kaggle competition**.

```
import librosa as lr
from glob import glob

# List all the wav files in the folder
audio_files = glob(data_dir + '/*.wav')

# Read in the first audio file, create the time array
audio, sfreq = lr.load(audio_files[0])
time = np.arange(0, len(audio)) / sfreq

# Plot audio over time
fig, ax = plt.subplots()
ax.plot(time, audio)
ax.set(xlabel='Time (s)', ylabel='Sound Amplitude')
plt.show()
```



There are several seconds of heartbeat sounds in here, though note that most of this time is silence. A common procedure in machine learning is to separate the datapoints with lots of stuff happening from the ones that don't.

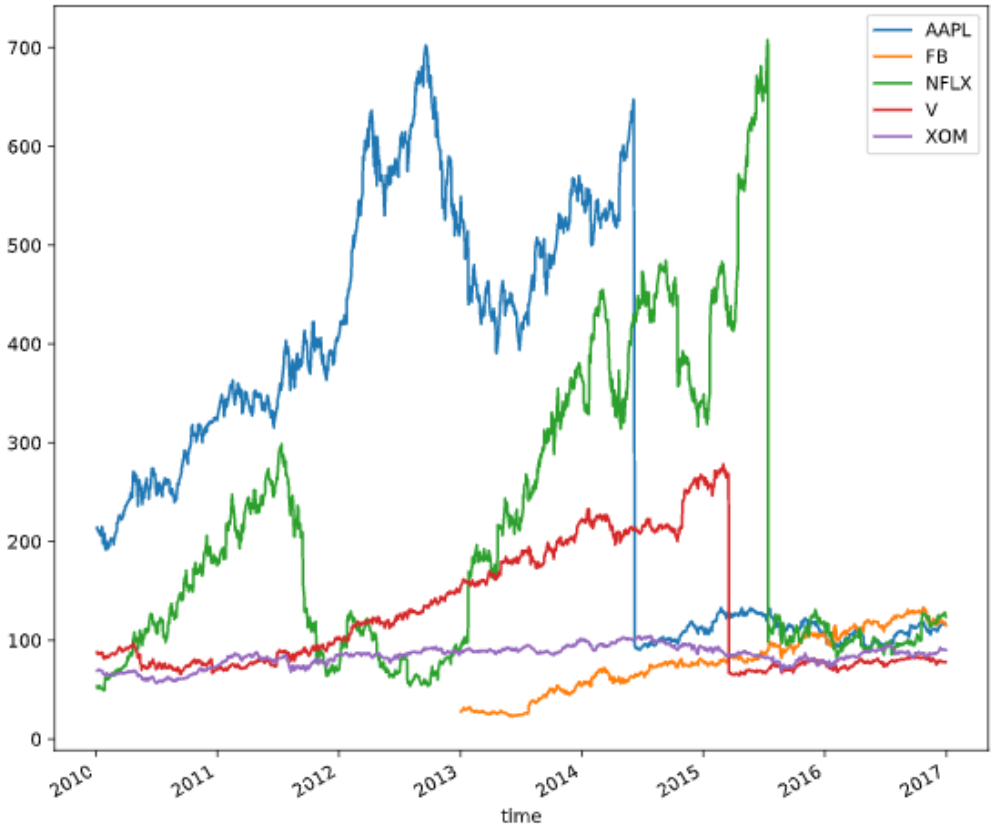
Inspecting the regression data

The next dataset contains information about company market value over several years of time. This is one of the most popular kind of time series data used for regression. If you can model the value of a company as it changes over time, you can make predictions about where that company will be in the future. This dataset was also originally provided as part of a **public Kaggle competition**.

In this exercise, you'll plot the time series for a number of companies to get an understanding of how they are (or aren't) related to one another.

```
# Read in the data
data = pd.read_csv('prices.csv', index_col=0)
```

```
# Convert the index of the DataFrame to datetime
data.index = pd.to_datetime(data.index)
print(data.head())

# Loop through each column, plot its values over time
fig, ax = plt.subplots()
for column in data.columns:
    data[column].plot(ax=ax, label=column)
ax.legend()
plt.show()
```



Note that each company's value is sometimes correlated with others, and sometimes not. Also note there are a lot of 'jumps' in there - what effect do you think these jumps would have on a predictive model?

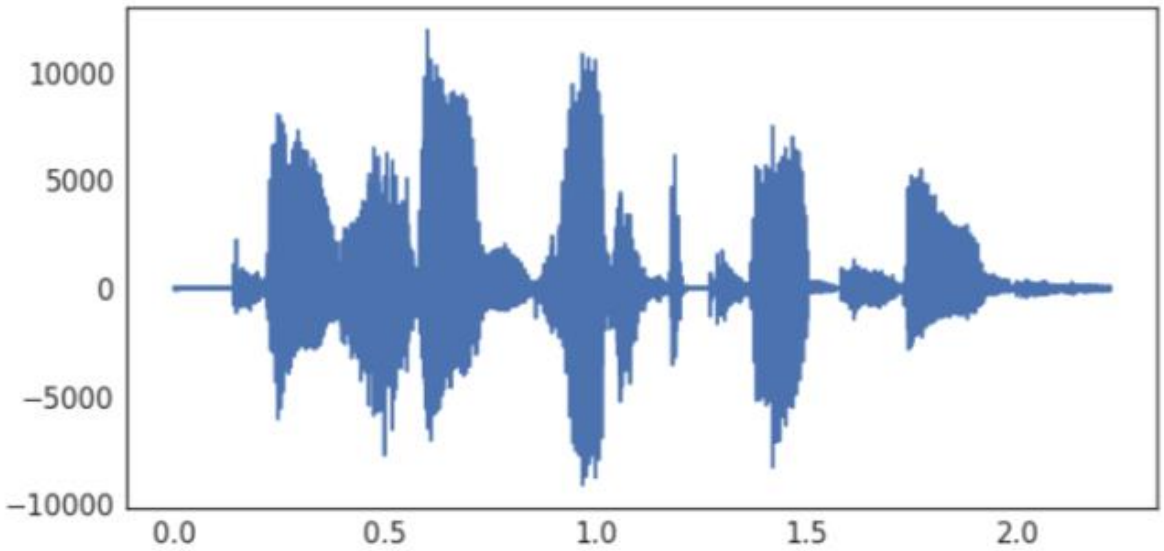
Chapter 2. Time Series as Inputs to a Model

The easiest way to incorporate time series into your machine learning pipeline is to use them as features in a model. This chapter covers common features that are extracted from time series in order to do machine learning.

Classifying a time series

Visualising audio waveform

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



Using raw timeseries data is too noisy for classification, as we need to calculate features. An easy start is to summarise the audio data, ie, Minimum, Maximum, Average.


```
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,)
```

axis=-1 will collapse across the last dimension, which is time. The result is an array of numbers, one array per timeseries. Therefore, the 2-D array (samples x time) has been collapsed into a 1-D array (samples) for each feature of interest. We can then combine these (samples x features) as inputs to a model.

For classification problems, we also need a label for each timeseries that allows us to build a classifier.

In order to prepare your data for scikit-learn, remember to ensure that it has the correct shape, which is samples by features. Use the column stack function, which let us stack 1-D arrays by turning them into the columns of a 2-D array. Labels array is reshaped from 1-D to 2-D.

```
# 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)
```

After fitting the model, we can then score the classifier (there are several ways) with scikit-learn.

```
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)
```

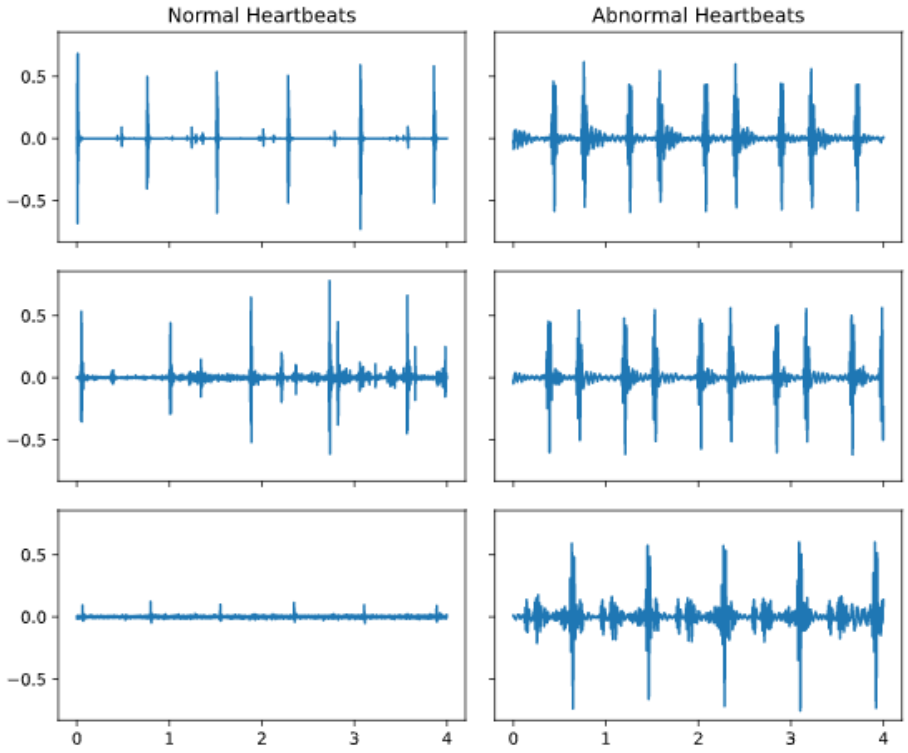
Many repetitions of sounds

In this exercise, you'll start with perhaps the simplest classification technique: averaging across dimensions of a dataset and visually inspecting the result.

You'll use the heartbeat data described in the last chapter. Some recordings are *normal* heartbeat activity, while others are *abnormal* activity. Let's see if you can spot the difference.

Two DataFrames, `normal` and `abnormal`, each with the shape of `(n_times_points, n_audio_files)` containing the audio for several heartbeats are available in your workspace. Also, the sampling frequency is loaded into a variable called `sfreq`. A convenience plotting function `show_plot_and_make_titles()` is also available in your workspace.

```
fig, axs = plt.subplots(3, 2, figsize=(15, 7), sharex=True, sharey=True)

# Calculate the time array
time = np.arange(normal.shape[0]) / sfreq

# Stack the normal/abnormal audio so you can loop and plot
stacked_audio = np.hstack([normal, abnormal]).T

# Loop through each audio file / ax object and plot
# .T.ravel() transposes the array, then unravels it into a 1-D vector for looping
for iaudio, ax in zip(stacked_audio, axs.T.ravel()):
    ax.plot(time, iaudio)
show_plot_and_make_titles()
```



As you can see there is a lot of variability in the raw data, let's see if you can average out some of that noise to notice a difference.

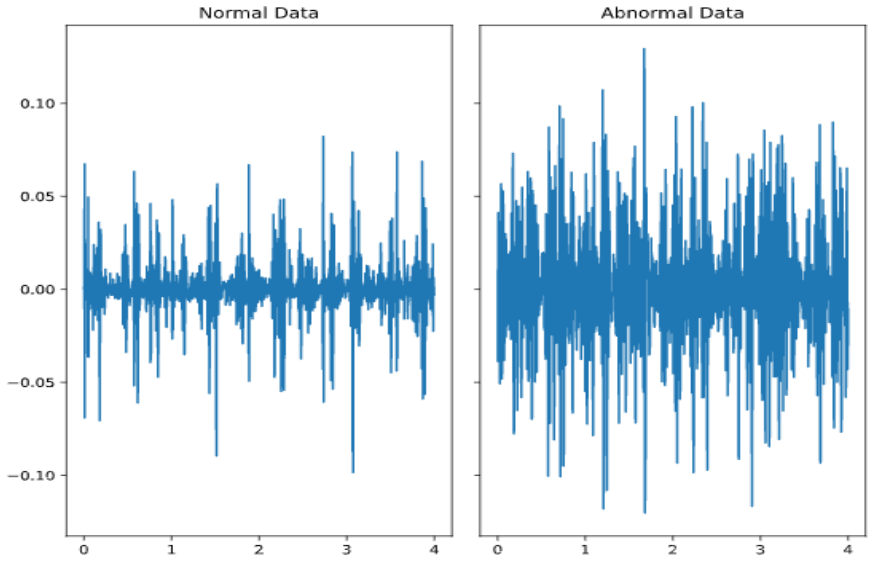
Invariance in time

While you should always start by visualizing your raw data, this is often uninformative when it comes to discriminating between two classes of data points. Data is usually noisy or exhibits complex patterns that aren't discoverable by the naked eye.

Another common technique to find simple differences between two sets of data is to *average* across multiple instances of the same class. This *may* remove noise and reveal underlying patterns (or, it may not).

In this exercise, you'll average across many instances of each class of heartbeat sound.
The two DataFrames (`normal` and `abnormal`) and the time array (`time`) from the previous exercise are available in your workspace.

```
# Average across the audio files of each DataFrame
mean_normal = np.mean(normal, axis=1)
mean_abnormal = np.mean(abnormal, axis=1)

# Plot each average over time
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 3), sharey=True)
ax1.plot(time, mean_normal)
ax1.set(title="Normal Data")
ax2.plot(time, mean_abnormal)
ax2.set(title="Abnormal Data")
plt.show()
```



Do you see a noticeable difference between the two? Maybe, but it's quite noisy. Let's see how you can dig into the data a bit further.

Build a classification model

While eye-balling differences is a useful way to gain an intuition for the data, let's see if you can operationalize things with a model. In this exercise, you will use each repetition as a datapoint, and each moment in time as a feature to fit a classifier that attempts to predict abnormal vs. normal heartbeats using *only the raw data*.

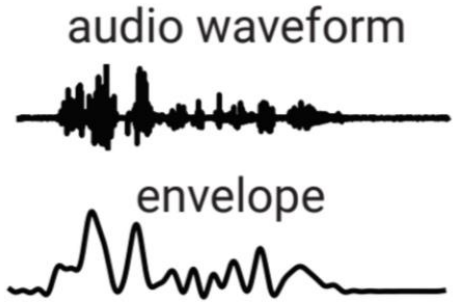
We've split the two DataFrames (`normal` and `abnormal`) into `X_train`, `X_test`, `y_train`, and `y_test`.

```
from sklearn.svm import LinearSVC

# Initialize and fit the model
model = LinearSVC()
model.fit(X_train, y_train)

# Generate predictions and score them manually
predictions = model.predict(X_test)
print(sum(predictions == y_test.squeeze()) / len(y_test))
```

```
<script.py> output:
    0.555555555556
```

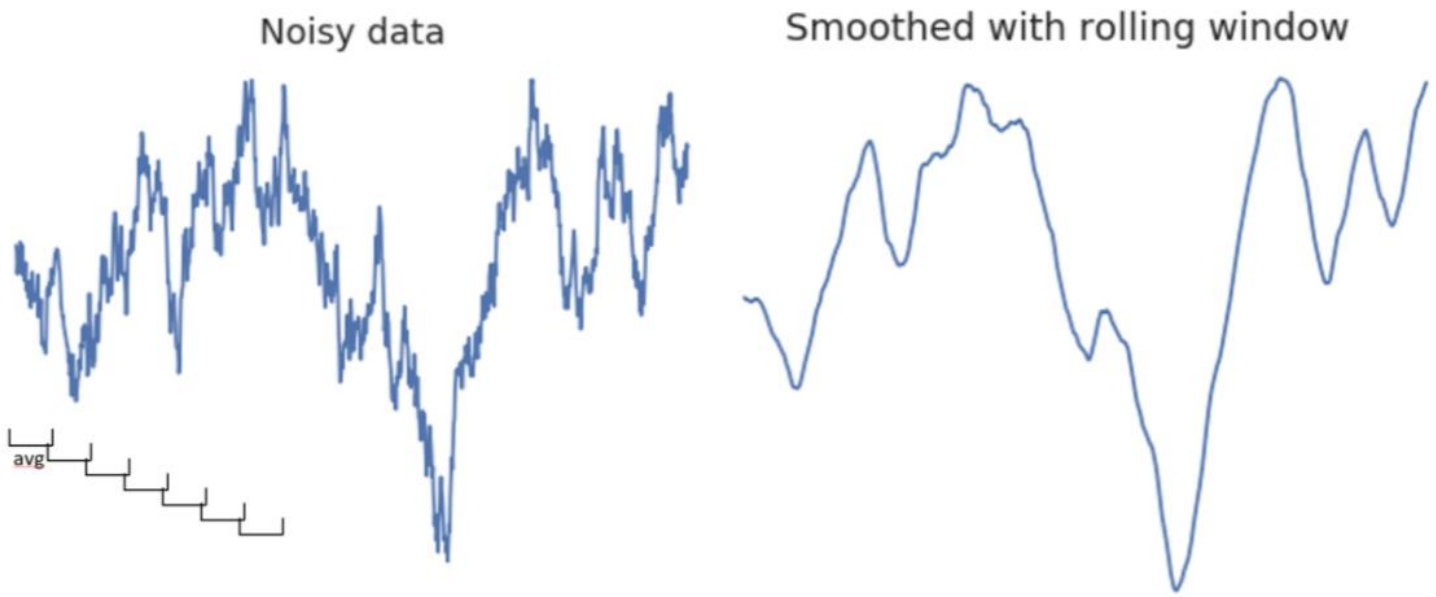
Note that your predictions didn't do so well. That's because the features you're using as inputs to the model (raw data) aren't very good at differentiating classes. Next, you'll explore how to calculate some more complex features that may improve the results.

Improving features for classification



Smooth the data to calculate the **auditory envelope** (total amount of audio energy at each moment of time)

Smoothing the timeseries = averaging locally only (instead of averaging over all time)
This removes short-term noise, while retaining the general pattern.

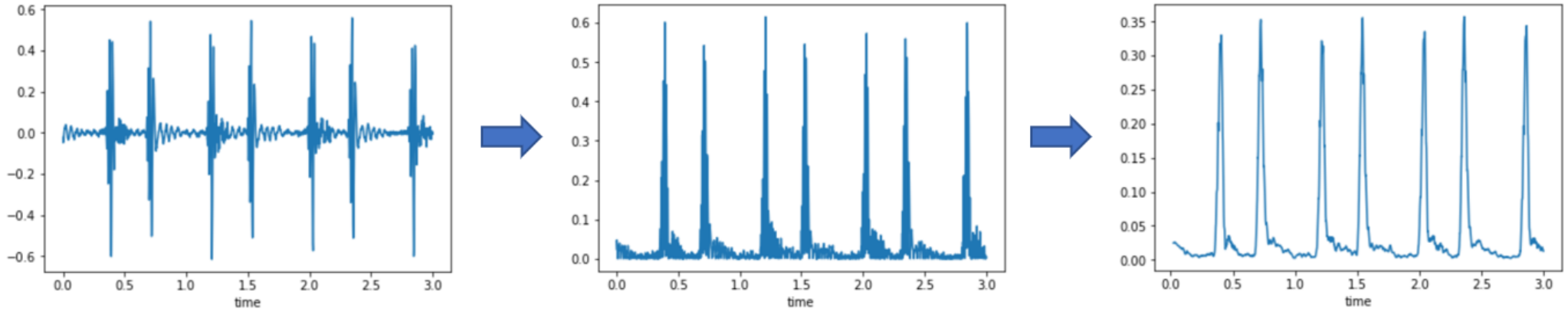



```
# Audio is a Pandas DataFrame
print(audio.shape)
# (n_times, n_audio_files)
```

```
(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()
```

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



Raw audio signal | Absolute value of each timepoint | Smooth the rectified signal

The result is a smooth representation of how the audio energy changes over 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])
y = labels.reshape([-1, 1])
```

```
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]
```

Computing the tempogram (tempo of a sound over time) using **librosa**, and use it to calculate features.

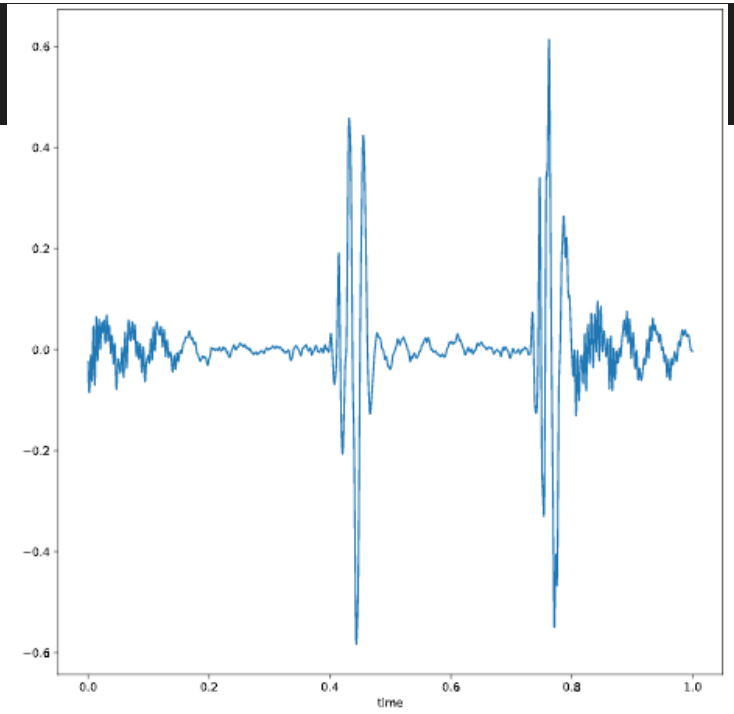
```
# Import librosa and calculate the tempo of a 1-D sound array
import librosa as lr
audio_tempo = lr.beat.tempo(audio, sr=sfreq,
                            hop_length=2**6, aggregate=None)
```

Calculating the envelope of sound

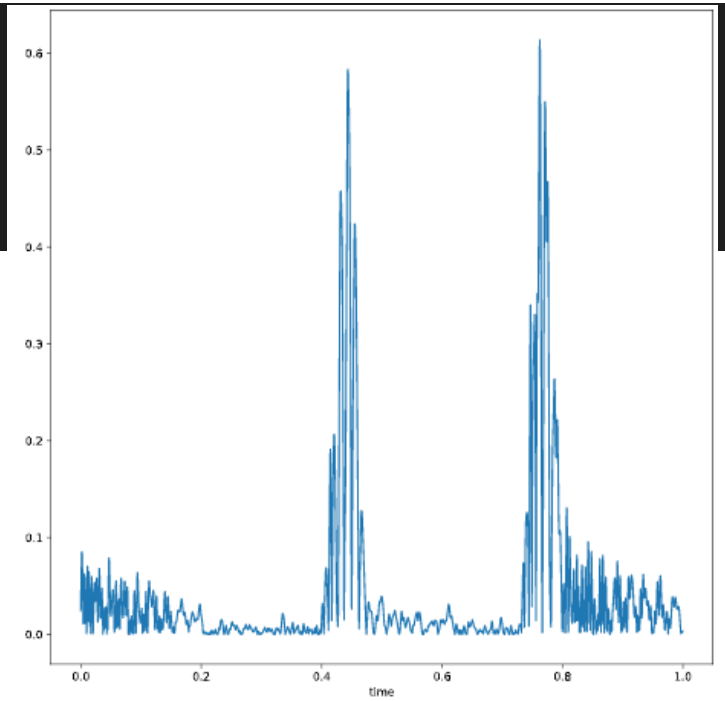
One of the ways you can improve the features available to your model is to remove some of the noise present in the data. In audio data, a common way to do this is to *smooth* the data and then *rectify* it so that the total amount of sound energy over time is more distinguishable. You'll do this in the current exercise.

A heartbeat file is available in the variable `audio`.

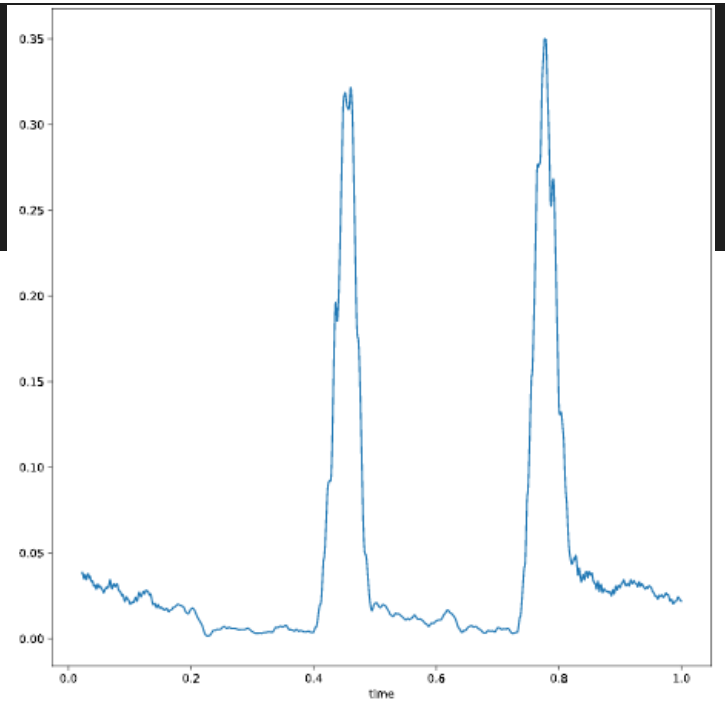
```
# Plot the raw data first
audio.plot(figsize=(10, 5))
plt.show()
```



```python
# Rectify the audio signal
audio_rectified = audio.apply(np.abs)

# Plot the result
audio_rectified.plot(figsize=(10, 5))
plt.show()
```




```python
# Smooth by applying a rolling mean
audio_rectified_smooth = audio_rectified.rolling(50).mean()

# Plot the result
audio_rectified_smooth.plot(figsize=(10, 5))
plt.show()
```



By calculating the envelope of each sound and smoothing it,

you've eliminated much of the noise and have a cleaner signal

to tell you when a heartbeat is happening.

Calculating features from the envelope

Now that you've removed some of the noisier fluctuations in the audio, let's see if this improves your ability to classify.

`audio_rectified_smooth` from the previous exercise is available in your workspace.

```
# Calculate stats
means = np.mean(audio_rectified_smooth, axis=0)
stds = np.std(audio_rectified_smooth, axis=0)
maxs = np.max(audio_rectified_smooth, axis=0)

# Create the X and y arrays
X = np.column_stack([means, stds, maxs])
y = labels.reshape([-1, 1])

# Fit the model and score on testing data
from sklearn.model_selection import cross_val_score
percent_score = cross_val_score(model, X, y, cv=5)
print(np.mean(percent_score))
```

```
<script.py> output:
    0.716666666667
```

This model is both simpler (only 3 features) and more understandable (features are simple summary statistics of the data).

Derivative features: The tempogram

One benefit of cleaning up your data is that it lets you compute more sophisticated features. For example, the envelope calculation you performed is a common technique in computing **tempo** and **rhythm** features. In this exercise, you'll use `librosa` to compute some tempo and rhythm features for heartbeat data, and fit a model once more.

Note that `librosa` functions tend to only operate on **numpy arrays** instead of DataFrames, so we'll access our Pandas data as a Numpy array with the `.values` attribute.

```
# Calculate the tempo of the sounds
tempos = []
for col, i_audio in audio.items():
    tempos.append(lr.beat.tempo(i_audio.values, sr=sfreq, hop_length=2**6, aggregate=None))

# Convert the list to an array so you can manipulate it more easily
tempos = np.array(tempos)

# Calculate statistics of each tempo
tempos_mean = tempos.mean(axis=-1)
tempos_std = tempos.std(axis=-1)
tempos_max = tempos.max(axis=-1)
```

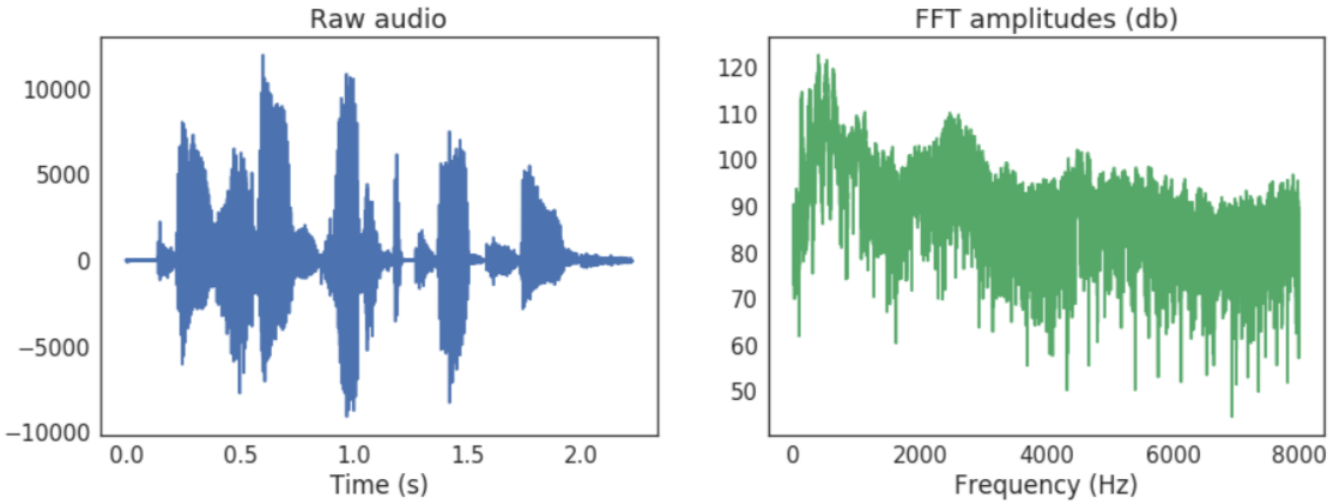
```
# Create the X and y arrays
X = np.column_stack([means, stds, maxs, tempos_mean, tempos_std, tempos_max])
y = labels.reshape([-1, 1])

# Fit the model and score on testing data
percent_score = cross_val_score(model, X, y, cv=5)
print(np.mean(percent_score))
```

```
<script.py> output:
    0.533333333333
```

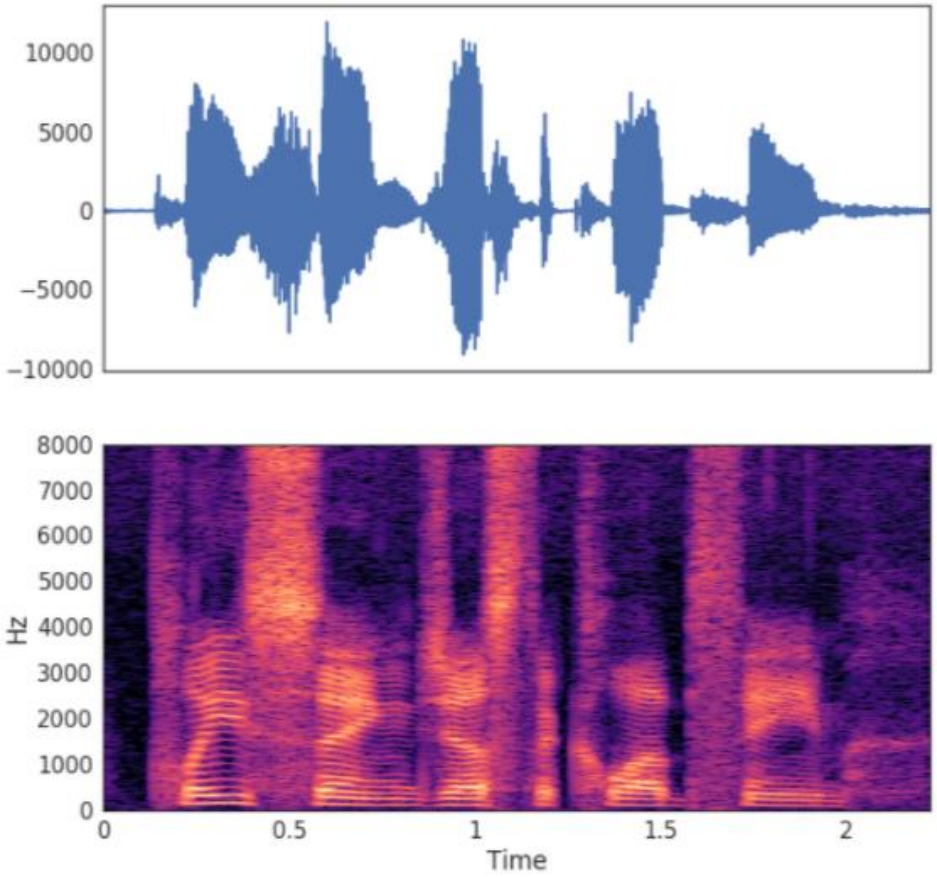
Note that your predictive power may not have gone up (because this dataset is quite small), but you now have a more rich feature representation of audio that your model can use!

The spectrogram

Timeseries data is a combination of 2 components: fast-changing things and slow-changing things. To describe their relative presence, do a Fourier Transformation (FFT). This converts a single timeseries into an array that describes the timeseries as a combination of oscillations.

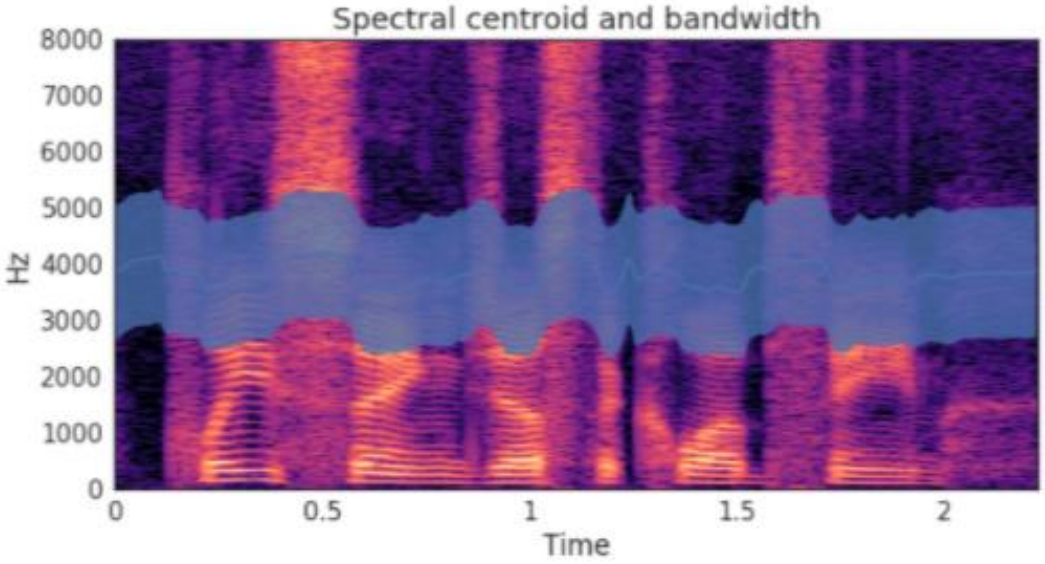


A spectrogram is a collection of windowed FFT over time, called a Short-Time Fourier Transform (**STFT**).



```
# 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)
```

```
# 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])
```

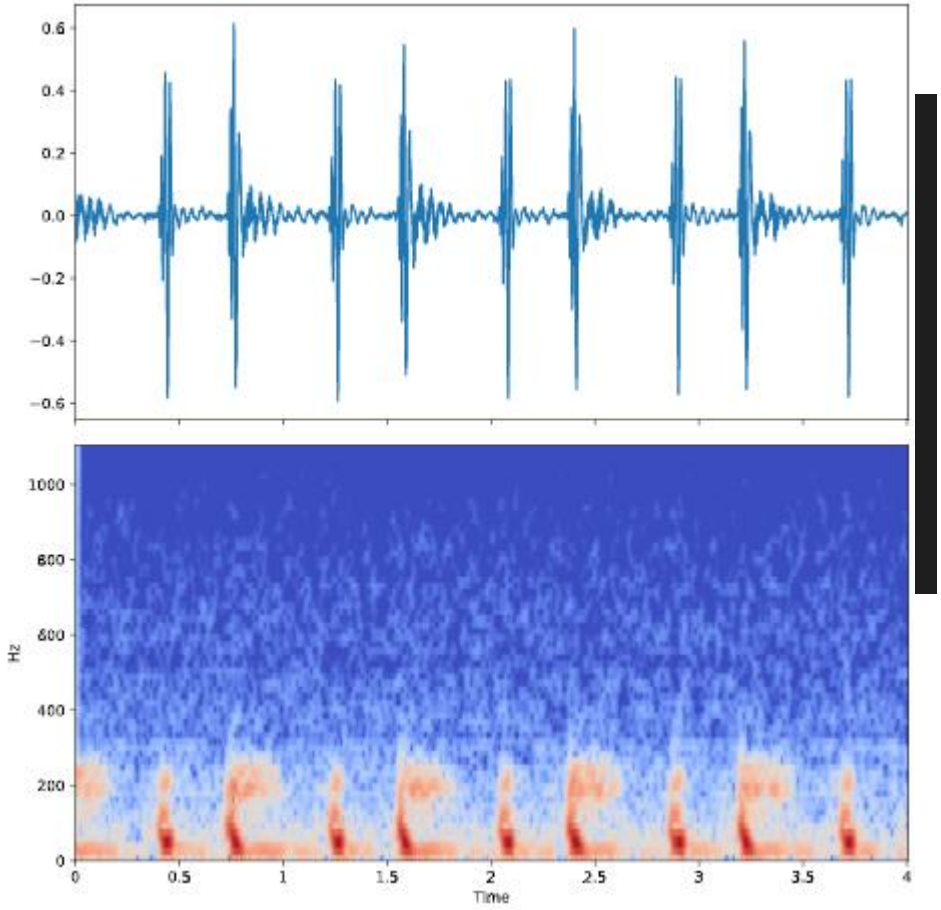
Spectrograms of heartbeat audio

Spectral engineering is one of the most common techniques in machine learning for time series data. The first step in this process is to calculate a **spectrogram** of sound. This describes what spectral content (e.g., low and high pitches) are present in the sound over time. In this exercise, you'll calculate a spectrogram of a heartbeat audio file.

We've loaded a single heartbeat sound in the variable `audio`.

```
# Import the stft function
from librosa.core import stft

# Prepare the STFT
HOP_LENGTH = 2**4
spec = stft(audio, hop_length=HOP_LENGTH, n_fft=2**7)
```

```
from librosa.core import amplitude_to_db
from librosa.display import specshow

# Convert into decibels
spec_db = amplitude_to_db(spec)

# Compare the raw audio to the spectrogram of the audio
fig, axs = plt.subplots(2, 1, figsize=(10, 10), sharex=True)
axs[0].plot(time, audio)
specshow(spec_db, sr=sfreq, x_axis='time', y_axis='hz',
         hop_length=HOP_LENGTH)
plt.show()
```



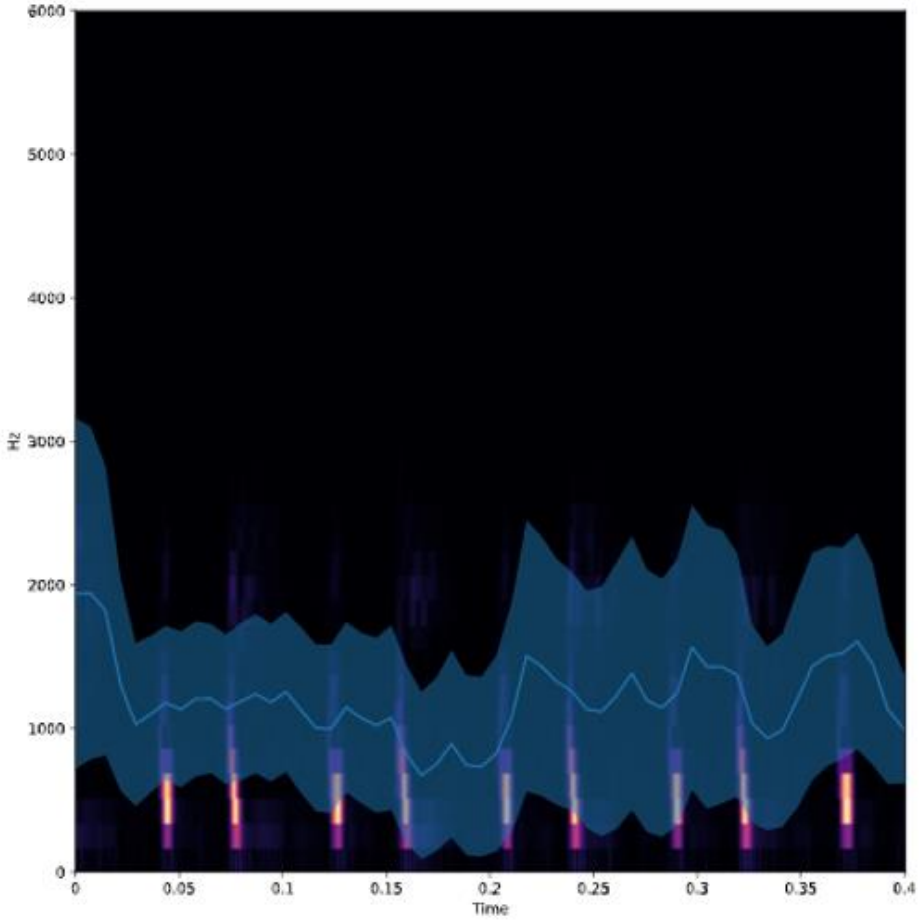
Do you notice that the heartbeats come in pairs, as seen by the vertical lines in the spectrogram?

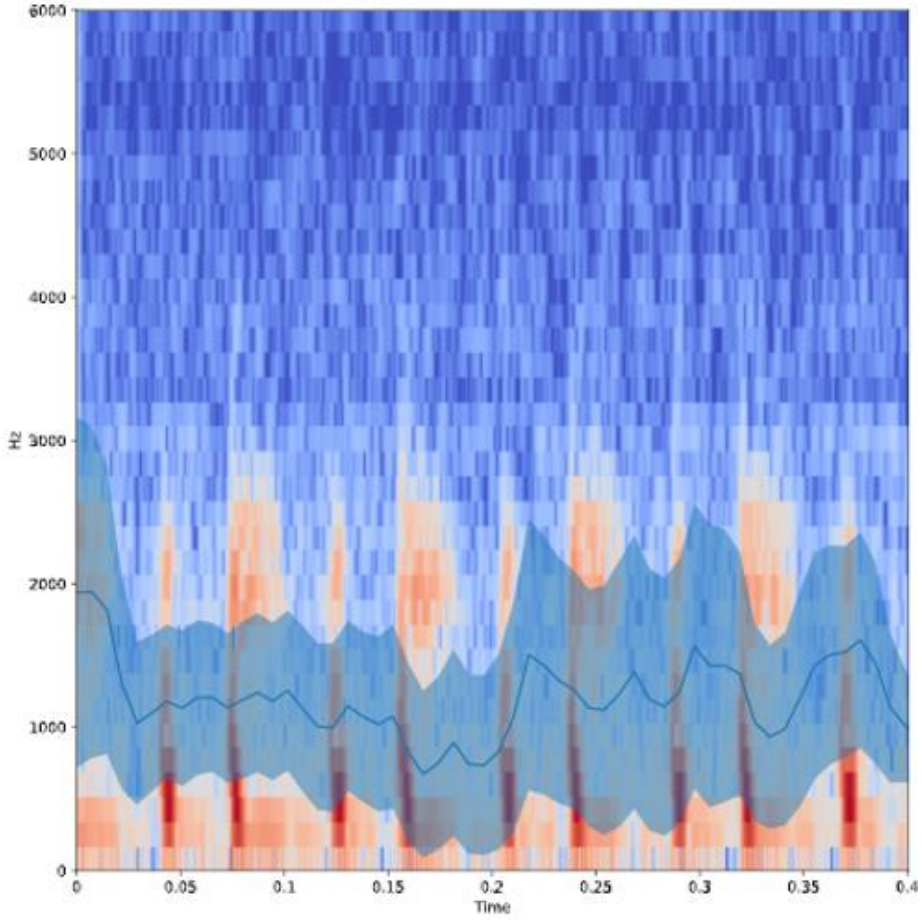
Engineering spectral features

As you can probably tell, there is a lot more information in a spectrogram compared to a raw audio file. By computing the spectral features, you have a much better idea of what's going on. As such, there are all kinds of spectral features that you can compute using the spectrogram as a base. In this exercise, you'll look at a few of these features. The spectogram `spec` from the previous exercise is available in your workspace.

```
import librosa as lr

# 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]
```

```
from librosa.core import amplitude_to_db
from librosa.display import specshow

# Convert spectrogram to decibels for visualization
spec_db = amplitude_to_db(spec)

# Display these features on top of the spectrogram
fig, ax = plt.subplots(figsize=(10, 5))
ax = specshow(spec_db, 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=.5)
ax.set(ylim=[None, 6000])
plt.show()
```





As you can see, the spectral centroid and bandwidth characterize the spectral content in each sound over time. They give us a summary of the spectral content that we can use in a classifier.

Combining many features in a classifier

You've spent this lesson engineering many features from the audio data - some contain information about how the audio changes in time, others contain information about the spectral content that is present.

The beauty of machine learning is that it can handle all of these features at the same time. If there is different information present in each feature, it should improve the classifier's ability to distinguish the types of audio. Note that this often requires more advanced techniques such as regularization, which we'll cover in the next chapter.

For the final exercise in the chapter, we've loaded many of the features that you calculated before. Combine all of them into an array that can be fed into the classifier, and see how it does.

```
# Loop through each spectrogram
bandwidths = []
centroids = []

for spec in spectrograms:
    # Calculate the mean spectral bandwidth
    this_mean_bandwidth = np.mean(lr.feature.spectral_bandwidth(S=spec))
    # Calculate the mean spectral centroid
    this_mean_centroid = np.mean(lr.feature.spectral_centroid(S=spec))
    # Collect the values
    bandwidths.append(this_mean_bandwidth)
    centroids.append(this_mean_centroid)
```

```
# Create X and y arrays
X = np.column_stack([means, stds, maxs, tempo_mean, tempo_max, tempo_std, bandwidths, centroids])
y = labels.reshape([-1, 1])

# Fit the model and score on testing data
percent_score = cross_val_score(model, X, y, cv=5)
print(np.mean(percent_score))
```

```
<script.py> output:
    0.483333333333
```

You calculated many different features of the audio, and combined each of them under the assumption that they provide independent information that can be used in classification. You may have noticed that the accuracy of your models varied a lot when using different set of features. This chapter was focused on creating new "features" from raw data and not obtaining the best accuracy. To improve the accuracy, you want to find the right features that provide relevant information and also build models on *much* larger data.

Chapter 3. Predicting Time Series Data

If you want to predict patterns from data over time, there are special considerations to take in how you choose and construct your model. This chapter covers how to gain insights into the data before fitting your model, as well as best-practices in using predictive modeling for time series data.

Predicting data over time

Timeseries regression predicts continuous outputs; classification predicts categorical outputs.

CLASSIFICATION

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

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

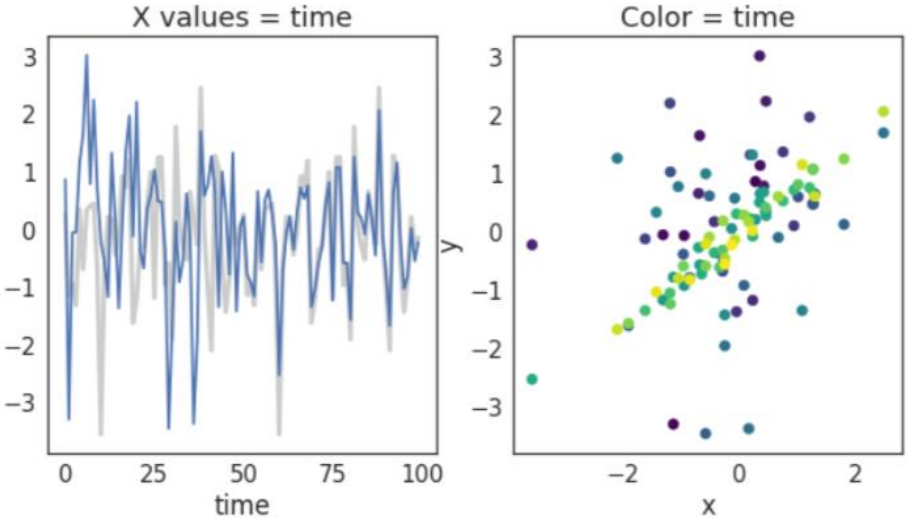
REGRESSION

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

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

Timeseries have patterns that change over time. Two timeseries that seem correlated at one moment may not remain so over time. Visualising timeseries data by directly comparing 2 segments of time.

```
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')
```

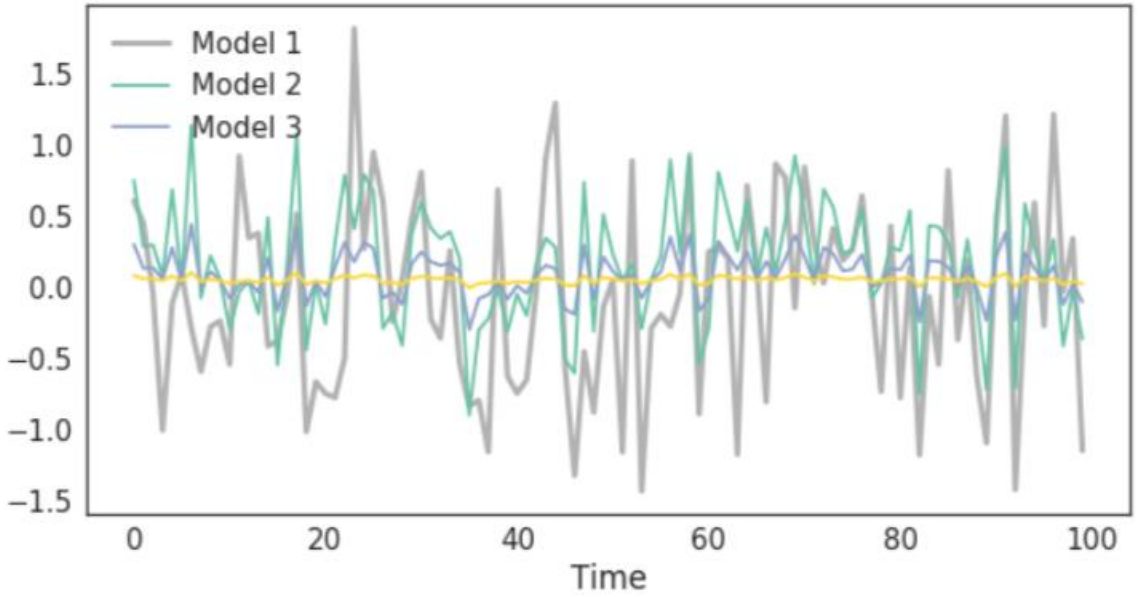


It seems like these 2 timeseries are uncorrelated at first, but then move in sync with one another. Confirm this by looking at the brighter colors on the right chart: brighter daatapoints fall on a line, meaning that for those moments in time, the 2 variables had a linear relationship.

Visualising predictions with scikit-learn:

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

```
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")
```



Coefficient of Determintation (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

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

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

```
0.08
```

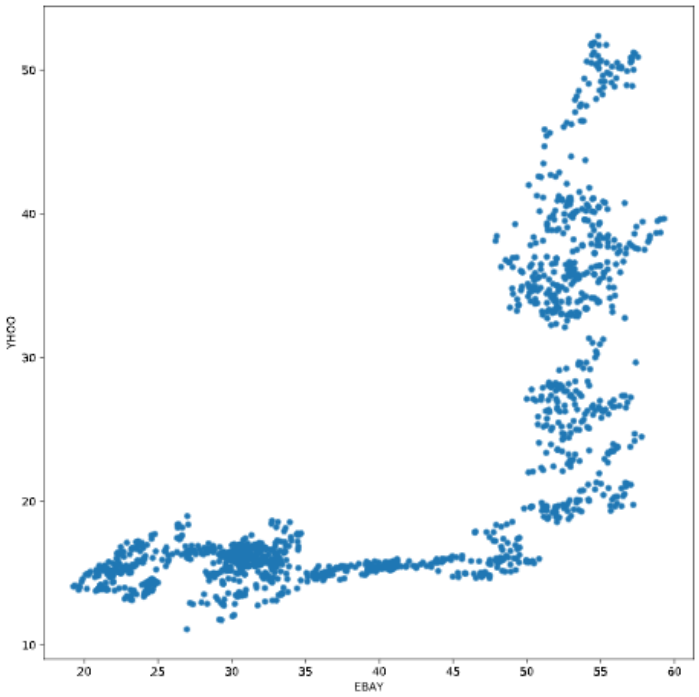
Introducing the dataset

You would deal with stock market prices that fluctuate over time. In this exercise you've got historical prices from two tech companies (**Ebay** and **Yahoo**) in the DataFrame `prices`. You'll visualize the raw data for the two companies, then generate a scatter plot showing how the values for each company compare with one another. Finally, you'll add in a "time" dimension to your scatter plot so you can see how this relationship changes over time.

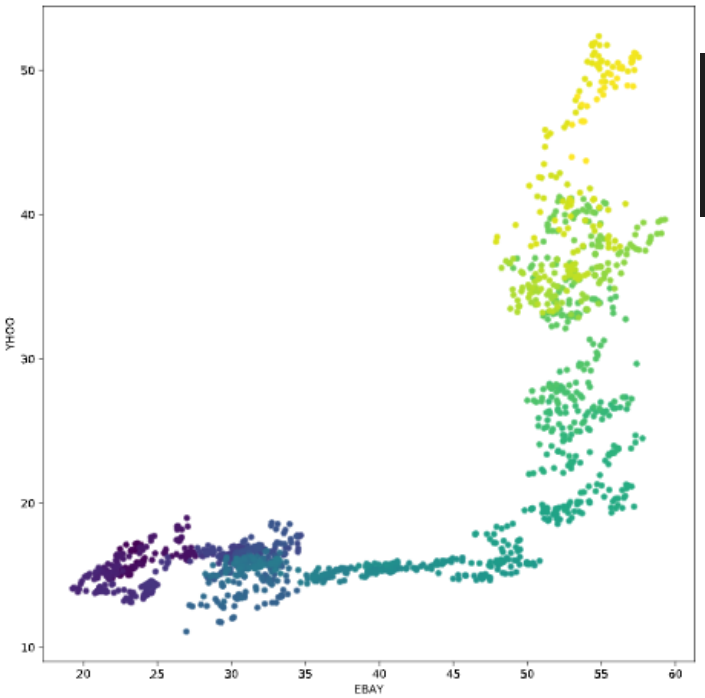
The data has been loaded into a DataFrame called `prices`.

```
# Plot the raw values over time
prices.plot()
plt.show()
```


```
# Scatterplot with one company per axis
prices.plot.scatter('EBAY', 'YHOO')
plt.show()
```



```
# Scatterplot with color relating to time
prices.plot.scatter('EBAY', 'YHOO', c=prices.index,
                    cmap=plt.cm.viridis, colorbar=False)
plt.show()
```



As you can see, these two time series seem somewhat related to each other, though its a complex relationship that changes over time.

Fitting a simple regression model

Now we'll look at a larger number of companies. Recall that we have historical price values for many companies. Let's use data from several companies to predict the value of a test company. You'll attempt to predict the value of the **Apple** stock price using the values of NVidia, Ebay, and Yahoo. Each of these is stored as a column in the `all_prices` DataFrame. Below is a mapping from company name to column name:

```
ebay: "EBAY"
nvidia: "NVDA"
yahoo: "YHOO"
apple: "AAPL"
```

We'll use these columns to define the input/output arrays in our model.

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# Use stock symbols to extract training data
```

```
X = all_prices[['EBAY', 'NVDA', 'YHOO']]
y = all_prices[['AAPL']]

# Fit and score the model with cross-validation
scores = cross_val_score(Ridge(), X, y, cv=3)
print(scores)
```

```
<script.py> output:
    [-6.09050633 -0.3179172  -3.72957284]
```

As you can see, fitting a model with raw data doesn't give great results.

Visualizing predicted values

When dealing with time series data, it's useful to visualize model predictions on top of the "actual" values that are used to test the model.

In this exercise, after splitting the data (stored in the variables `X` and `y`) into training and test sets, you'll build a model and then visualize the model's predictions on top of the testing data in order to estimate the model's performance.

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Split our data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    train_size=.8, shuffle=False, random_state=1)

# Fit our model and generate predictions
model = Ridge()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
score = r2_score(y_test, predictions)
print(score)
```

```
<script.py> output:
    -5.70939901949
```

```
# Visualize our predictions along with the "true" values, and print the score
fig, ax = plt.subplots(figsize=(15, 5))
ax.plot(y_test, color='k', lw=3)
ax.plot(predictions, color='r', lw=2)
plt.show()
```

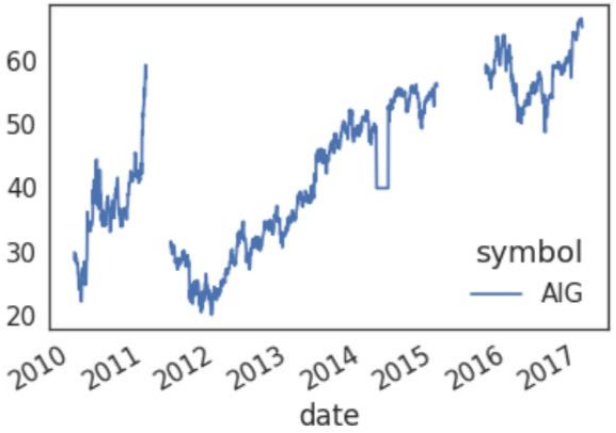
Now you have an explanation for your poor score.

The predictions clearly deviate from the true time series values.

Advanced time series prediction

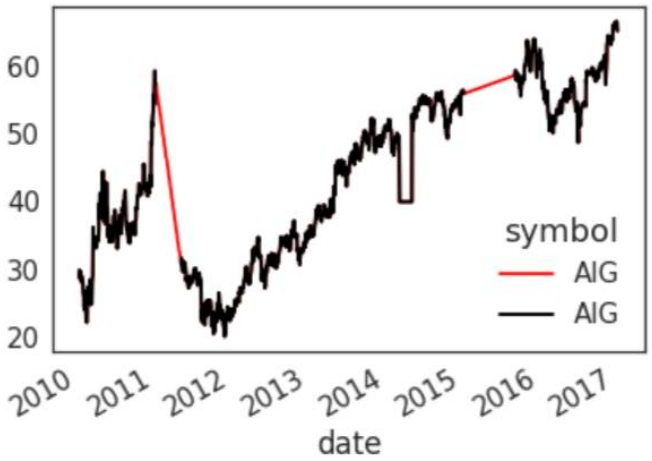
Real-world data is often messy, with most commonly missing data and outliers. This is usually because of human error, machine sensor malfunction, database failures, etc. Visualising raw data makes easier to spot these problems. Example:

- Missing data
- Outliers



Use **interpolation** to fill in missing data, ie, using the known values on either side of a gap in the data to make assumptions about what is missing

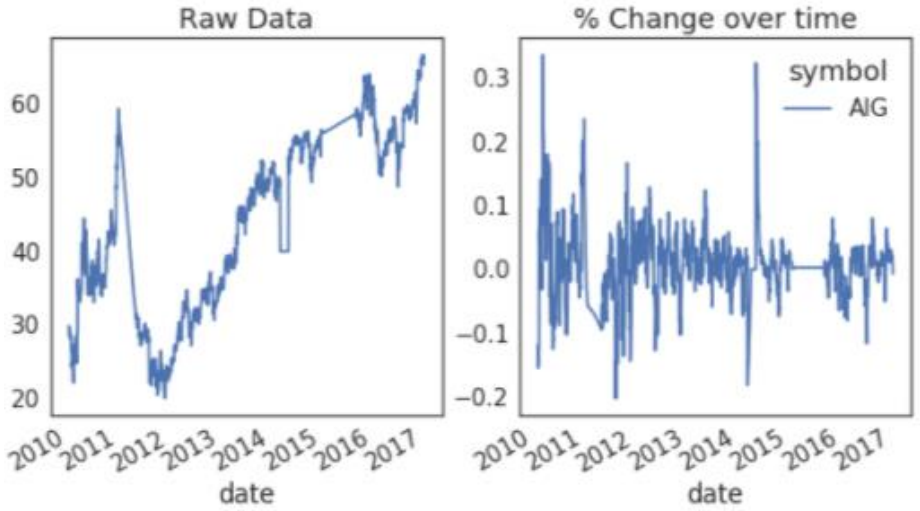
```
# Return a boolean that notes where missing values are
missing = prices.isna()

# 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)
```



Use a rolling window to transform data, to standardise its mean and variance over time.

```
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
```

```
# 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)
```



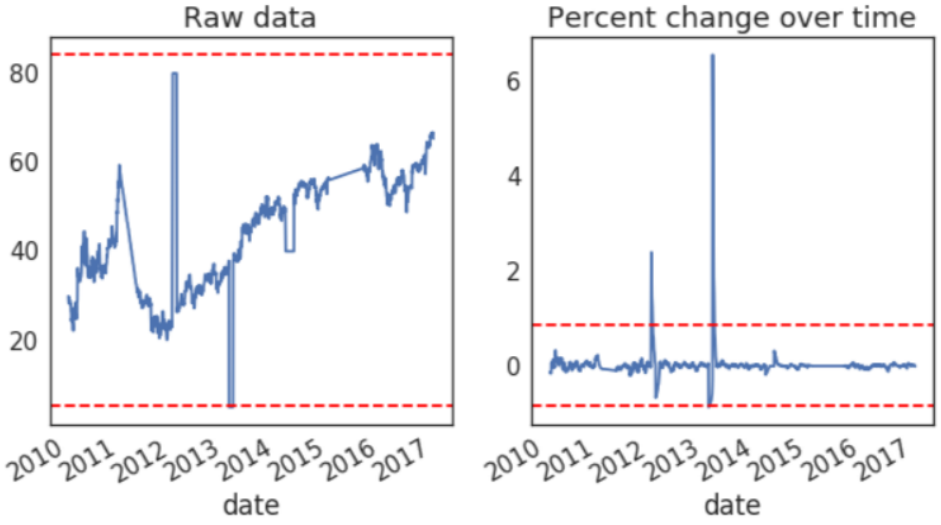
On the right, the data is now roughly centered at zero, and periods of high and low changes are easier to spot.

We use this transformation to detect outliers.

Identify and handle outliers

```
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')
```

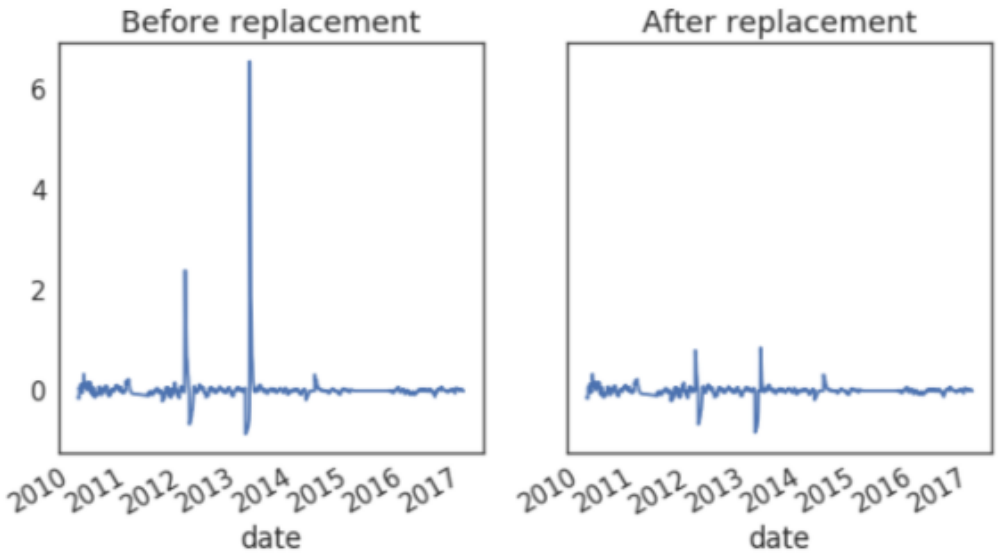
Outliers are datapoints that are significantly statistically different from the dataset, ie, more than 3 standard deviation from the mean. They can have negative effects on the predictive power of your model, biasing it away from its 'true' value. One solution is to remove or replace outliers with a more representative value.



Datapoints deemed an outlier depend on the transformation of the data. On the right, we see a few outlier datapoints that were not outliers in the raw data.

```
# 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)
```

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



Visualizing messy data

Let's take a look at a new dataset - this one is a bit less-clean than what you've seen before.
As always, you'll first start by visualizing the raw data. Take a close look and try to find datapoints that could be problematic for fitting models.
The data has been loaded into a DataFrame called `prices`.

```
# Visualize the dataset
prices.plot(legend=False)
plt.tight_layout()
plt.show()

# Count the missing values of each time series
missing_values = prices.isna().sum()
print(missing_values)
```

In the plot, you can see there are clearly missing chunks of time in your data.

There also seem to be a few 'jumps' in the data.

Imputing missing values

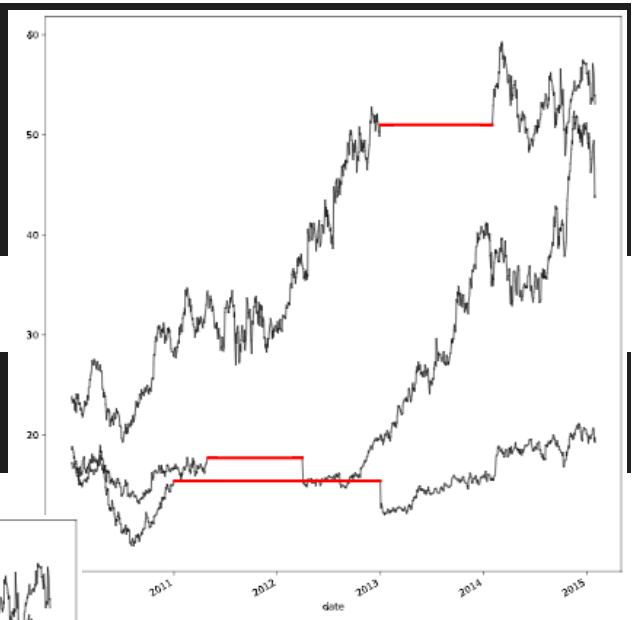
When you have missing data points, how can you fill them in?

In this exercise, you'll practice using different interpolation methods to fill in some missing values, visualizing the result each time. But first, you will create the function (`interpolate_and_plot()`) you'll use to interpolate missing data points and plot them.

A single time series has been loaded into a DataFrame called `prices`.

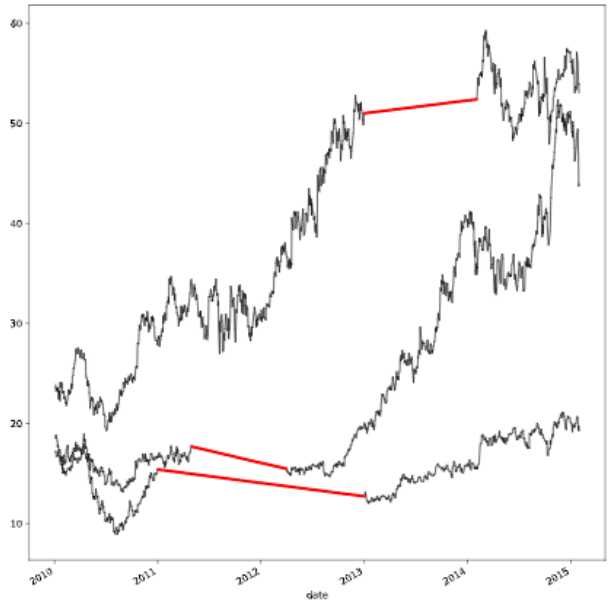
```
# Create a function we'll use to interpolate and plot
def interpolate_and_plot(prices, interpolation):

    # Create a boolean mask for missing values
    missing_values = prices.isna()

    # Interpolate the missing values
    prices_interp = prices.interpolate(interpolation)

    # Plot the results, highlighting the interpolated values in black
```

```
    fig, ax = plt.subplots(figsize=(10, 5))
    prices_interp.plot(color='k', alpha=.6, ax=ax, legend=False)

    # Now plot the interpolated values on top in red
    prices_interp[missing_values].plot(ax=ax, color='r', lw=3, legend=False)
    plt.show()
```

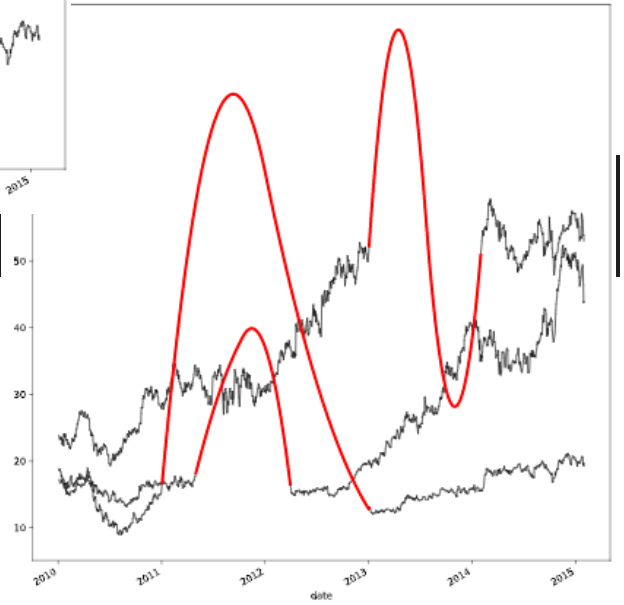



```
# Interpolate using the latest non-missing value
interpolation_type = 'zero'
interpolate_and_plot(prices, interpolation_type)
```

```
# Interpolate linearly
interpolation_type = 'linear'
interpolate_and_plot(prices, interpolation_type)
```




```
# Interpolate with a quadratic function
interpolation_type = 'quadratic'
interpolate_and_plot(prices, interpolation_type)
```



When you interpolate, the pre-existing data is used to infer the values of missing data.

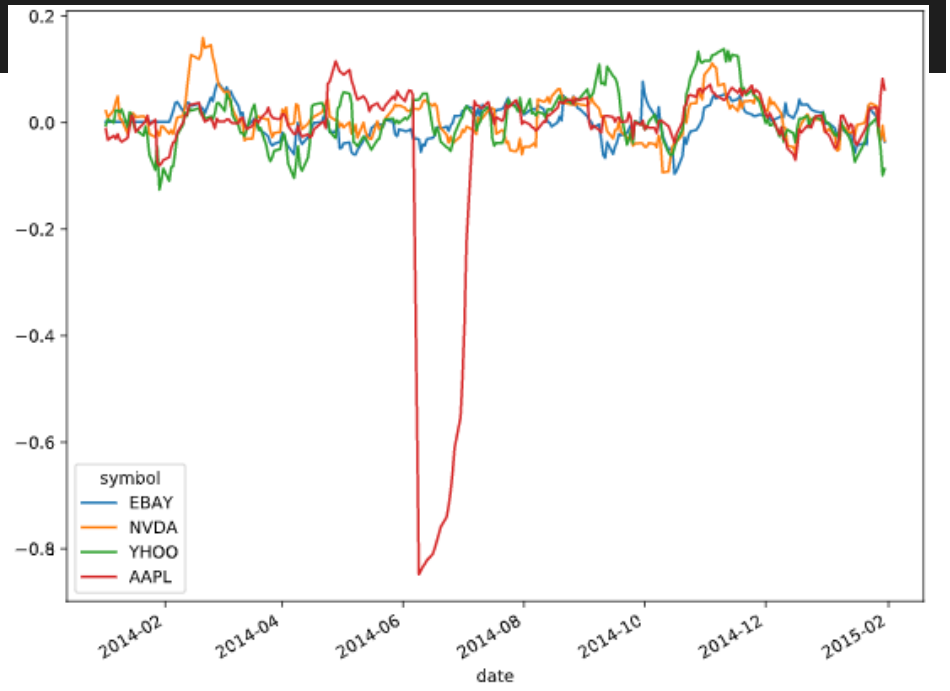
As you can see, the method you use for this has a big effect on the outcome.

Transforming raw data

In the last chapter, you calculated the rolling mean. In this exercise, you will define a function that calculates the percent change of the latest data point from the mean of a window of previous data points. This function will help you calculate the percent change over a rolling window.

This is a more stable kind of time series that is often useful in machine learning.

```
# Your custom function
def percent_change(series):
    # Collect all *but* the last value of this window, then the final value
    previous_values = series[:-1]
    last_value = series[-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

# Apply your custom function and plot
prices_perc = prices.rolling(20).apply(percent_change)
prices_perc.loc["2014":"2015"].plot()
plt.show()
```



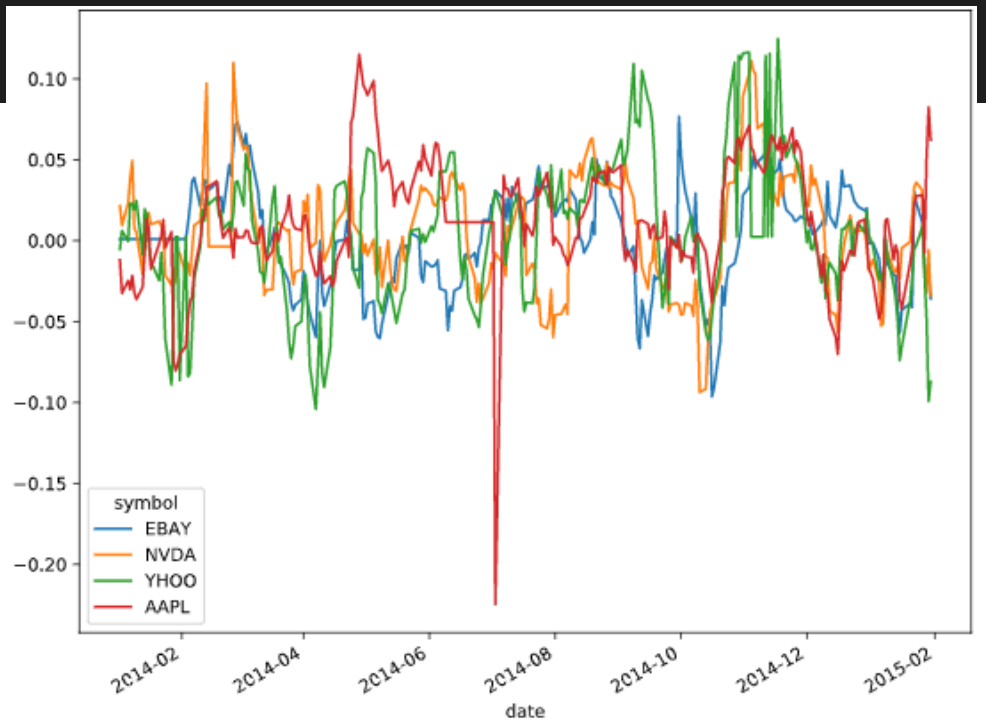
You've converted the data so it's easier to compare one time point to another.

This is a cleaner representation of the data.

Handling outliers

In this exercise, you'll handle outliers - data points that are so different from the rest of your data, that you treat them *differently* from other "normal-looking" data points. You'll use the output from the previous exercise (percent change over time) to detect the outliers. First you will write a function that replaces outlier data points with the median value from the entire time series.

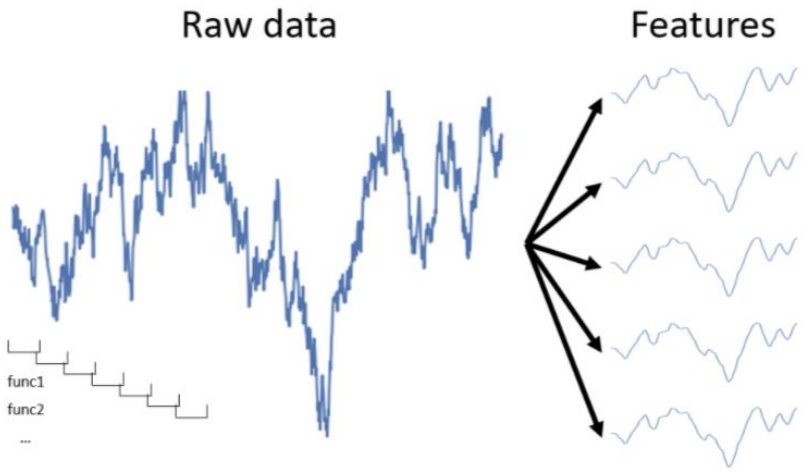
```
def replace_outliers(series):
    # Calculate the absolute difference of each timepoint from the series mean
    absolute_differences_from_mean = np.abs(series - np.mean(series))

    # Calculate a mask for the differences that are > 3 standard deviations from the mean
    this_mask = absolute_differences_from_mean > (np.std(series) * 3)

    # Replace these values with the median accross the data
    series[this_mask] = np.nanmedian(series)
    return series

# Apply your preprocessing function to the timeseries and plot the results
prices_perc = prices_perc.apply(replace_outliers)
prices_perc.loc["2014":"2015"].plot()
plt.show()
```



Since you've converted the data to % change over time, it was easier to spot and correct the outliers.

Creating features over time

Extracting features using **rolling windows**:



Extracting features using **.aggregate**:

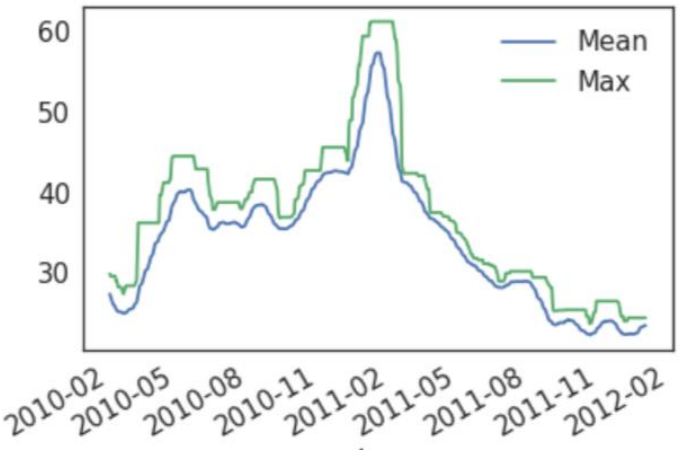
```
# Visualize the raw data
print(prices.head(3))
```

```
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
```

Always plot the features you have extracted over time,
as this can give you a clue on how they behave
and help you spot noisy data and outliers.



Here we see that the max value is much jumpier than the mean.

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.]
```

Using **np.percentile()** to summarise your data:

```
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)
```

Using **datetime** features in 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')
```

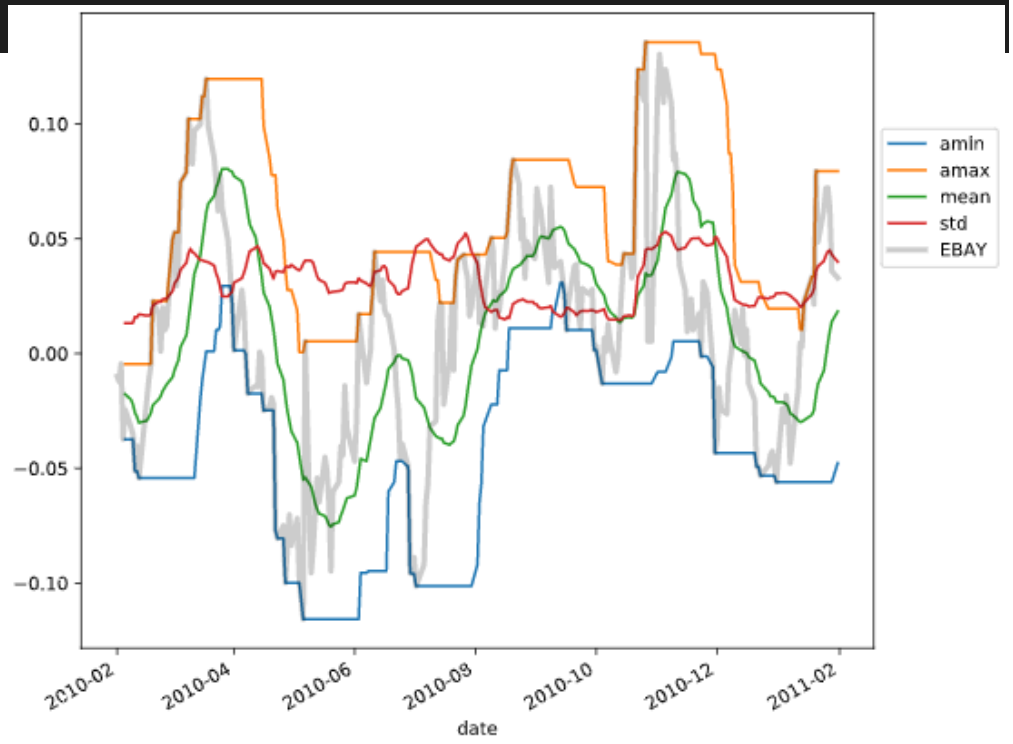
```
day_of_week = prices.index.weekday_name
print(day_of_week[:10])
```

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

Engineering multiple rolling features at once

Now that you've practiced some simple feature engineering, let's move on to something more complex. You'll calculate a collection of features for your time series data and visualize what they look like over time. This process resembles how many other time series models operate.

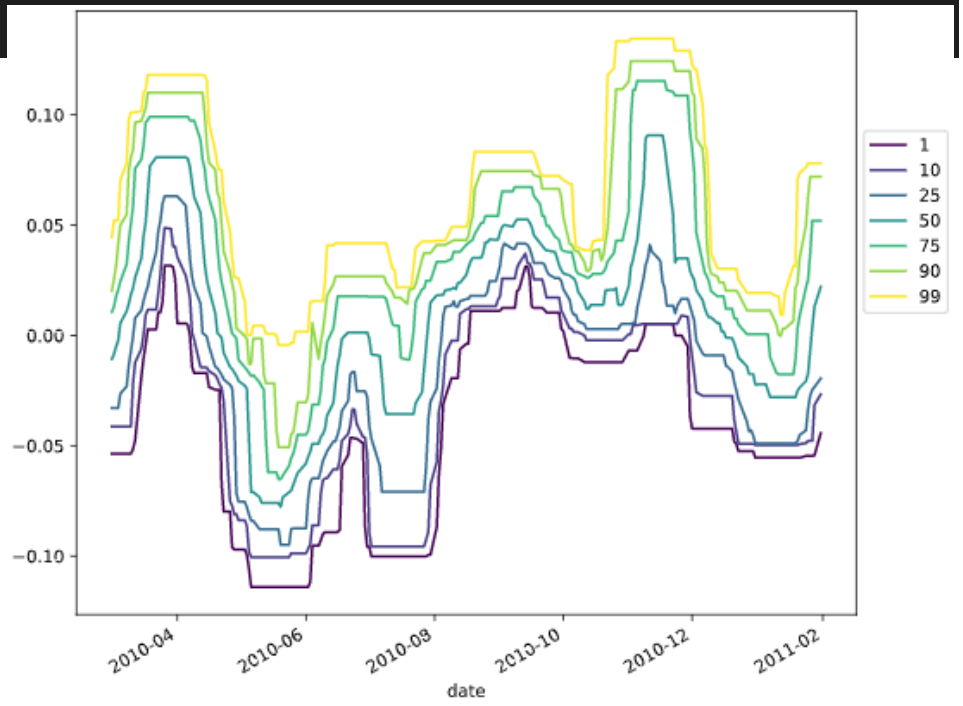
```
# Define a rolling window with Pandas, excluding the right-most datapoint of the window
prices_perc_rolling = prices_perc.rolling(20, min_periods=5, closed='right')

# Define the features you'll calculate for each window
features_to_calculate = [np.min, np.max, np.mean, np.std]

# Calculate these features for your rolling window object
features = prices_perc_rolling.aggregate(features_to_calculate)

# Plot the results
ax = features.loc[:"2011-01"].plot()
prices_perc.loc[:"2011-01"].plot(ax=ax, color='k', alpha=.2, lw=3)
ax.legend(loc=(1.01, .6))
plt.show()
```



Percentiles and partial functions

In this exercise, you'll practice how to pre-choose arguments of a function so that you can pre-configure how it runs. You'll use this to calculate several percentiles of your data using the same `percentile()` function in `numpy`.

```
# Import partial from functools
from functools import partial
percentiles = [1, 10, 25, 50, 75, 90, 99]

# Use a list comprehension to create a partial function for each quantile
percentile_functions = [partial(np.percentile, q=percentile) for percentile in percentiles]

# Calculate each of these quantiles on the data using a rolling window
prices_perc_rolling = prices_perc.rolling(20, min_periods=5, closed='right')
features_percentiles = prices_perc_rolling.aggregate(percentile_functions)

# Plot a subset of the result
ax = features_percentiles.loc[:"2011-01"].plot(cmap=plt.cm.viridis)
ax.legend(percentiles, loc=(1.01, .5))
plt.show()
```



Using "date" information

It's easy to think of timestamps as pure numbers, but don't forget they generally correspond to things that happen in the real world. That means there's often extra information encoded in the data such as "is it a weekday?" or "is it a holiday?". This information is often useful in predicting timeseries data.

In this exercise, you'll extract these *date/time* based features. A single time series has been loaded in a variable called `prices`.

```
# Extract date features from the data, add them as columns
prices_perc['day_of_week'] = prices_perc.index.dayofweek
prices_perc['week_of_year'] = prices_perc.index.weekofyear
prices_perc['month_of_year'] = prices_perc.index.month

# Print prices_perc
print(prices_perc)
```

```
<script.py> output:
                EBAY  day_of_week  week_of_year  month_of_year
    date
    2014-01-02  0.017938            3             1              1
    2014-01-03  0.002268            4             1              1
    2014-01-06 -0.027365            0             2              1
    2014-01-07 -0.006665            1             2              1
    2014-01-08 -0.017206            2             2              1
    2014-01-09 -0.023270            3             2              1
    2014-01-10 -0.022257            4             2              1
    2015-12-21 -0.037511            0            52             12
    2015-12-22 -0.024807            1            52             12
    2015-12-23 -0.026665            2            52             12
    2015-12-24 -0.028684            3            52             12
    2015-12-28 -0.026797            0            53             12
    2015-12-29 -0.013726            1            53             12
    2015-12-30 -0.017296            2            53             12
    2015-12-31 -0.024640            3            53             12

    [504 rows x 4 columns]
```

Chapter 4. Validating and Inspecting Time Series Models

Once you've got a model for predicting time series data, you need to decide if it's a good or a bad model. This chapter coves the basics of generating predictions with models in order to validate them against "test" data.

Creating features from the past

Timeseries data has a linear flow (matching the progression of time), patterns will persist over a span of datapoints, ie, information shared between timepoints. As a result, we can use info from the past to predict values in the future.

The smoothness of timeseries data reflects how much correlation there is between one time point and those that come before and after it. The extent to which previous timepoints are predictive of subsequent tiempoints is often described as “autocorrelation”, and can have a big impact on the performance of your model.

Time-shifting data with Pandas

```
print(df)
```

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

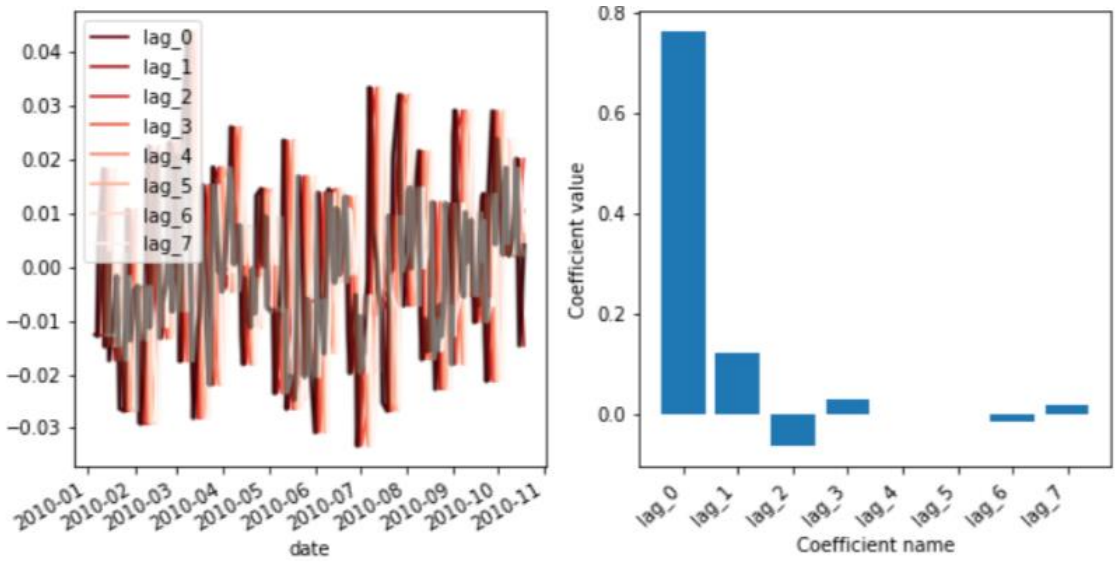
```
# Shift a DataFrame/Series by 3 index values towards the past
print(df.shift(3))
```

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

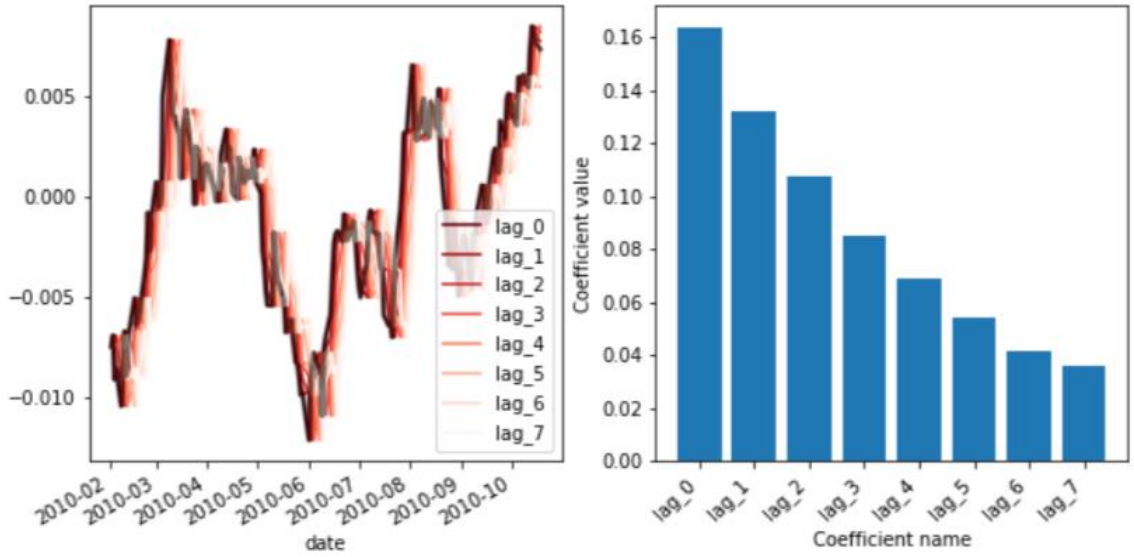
```python
# data is a pandas Series containing time series data
data = pd.Series(...)

# 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)
```

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

```python
# 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')
```



Example of non-smooth signal



Example of smooth signal, coefficients for time lags drop off to zero smoothly.

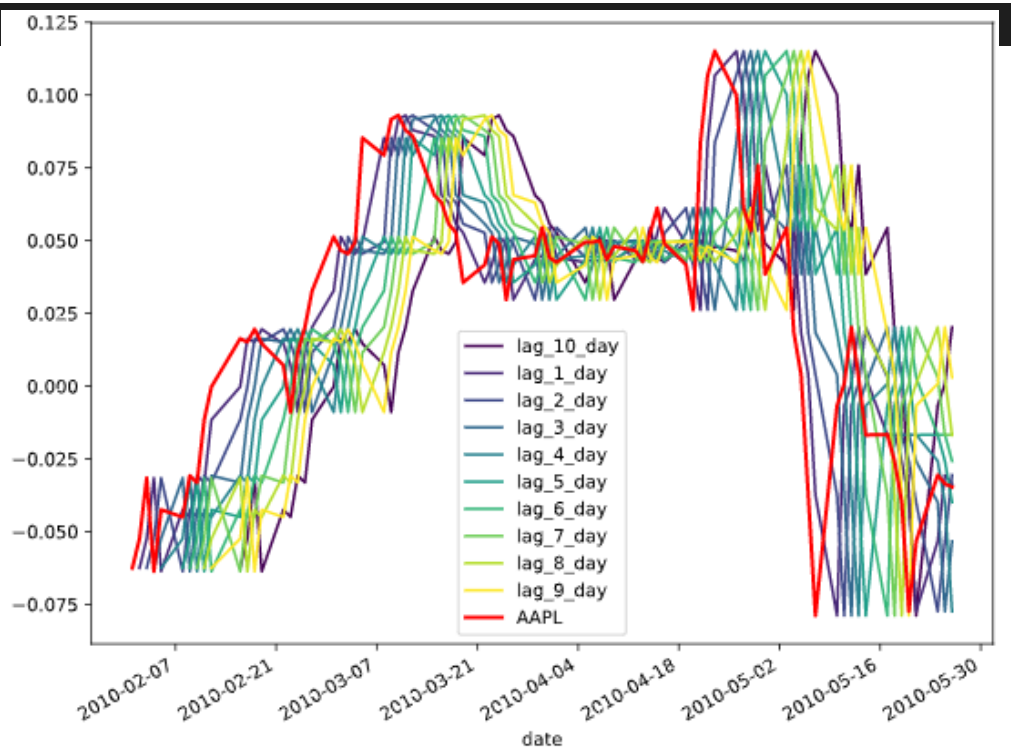
Creating time-shifted features

In machine learning for time series, it's common to use information about previous time points to predict a subsequent time point.

In this exercise, you'll "shift" your raw data and visualize the results. You'll use the *percent change* time series that you calculated in the previous chapter, this time with a *very short* window. A short window is important because, in a real-world scenario, you want to predict the day-to-day fluctuations of a time series, not its change over a longer window of time.

```
# These are the "time lags"
shifts = np.arange(1, 11).astype(int)

# Use a dictionary comprehension to create name: value pairs, one pair per shift
shifted_data = {"lag_{}_day".format(day_shift): prices_perc.shift(day_shift) for day_shift in shifts}

# Convert into a DataFrame for subsequent use
prices_perc_shifted = pd.DataFrame(shifted_data)

# Plot the first 100 samples of each
ax = prices_perc_shifted.iloc[:100].plot(cmap=plt.cm.viridis)
prices_perc.iloc[:100].plot(color='r', lw=2)
ax.legend(loc='best')
```

```
plt.show()
```



Special case: Auto-regressive models

Now that you've created time-shifted versions of a single time series, you can fit an *auto-regressive* model. This is a regression model where the input features are time-shifted versions of the output time series data. You are using previous values of a timeseries to predict current values of the same timeseries (thus, it is auto-regressive).

By investigating the coefficients of this model, you can explore any repetitive patterns that exist in a timeseries, and get an idea for how far in the past a data point is predictive of the future.

```
# Replace missing values with the median for each column
X = prices_perc_shifted.fillna(np.nanmedian(prices_perc_shifted))
y = prices_perc.fillna(np.nanmedian(prices_perc))

# Fit the model
model = Ridge()
model.fit(X, y)
```

You've filled in the missing values with the median so that it behaves well with scikit-learn.

Visualize regression coefficients

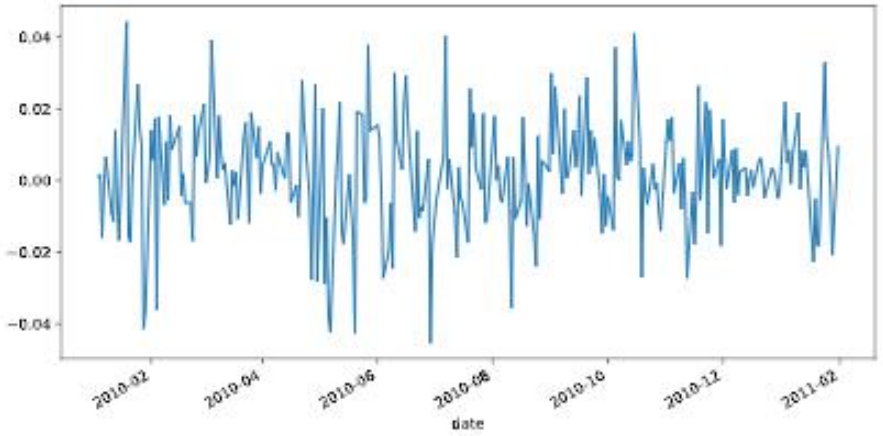
Now that you've fit the model, let's visualize its coefficients. This is an important part of machine learning because it gives you an idea for how the different features of a model affect the outcome.

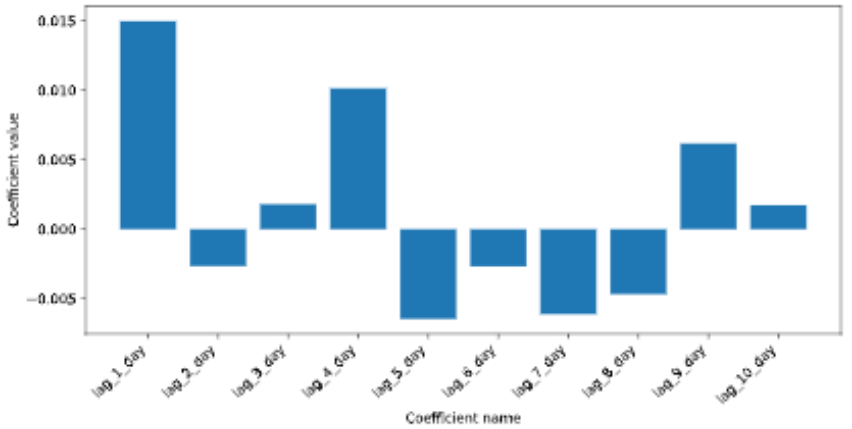
The shifted time series DataFrame (`prices_perc_shifted`) and the regression model (`model`) are available in your workspace.

In this exercise, you will create a function that, given a set of coefficients and feature names, visualizes the coefficient values.

```
def visualize_coefficients(coefs, names, ax):
    # Make a bar plot for the coefficients, including their names on the x-axis
    ax.bar(names, coefs)
    ax.set(xlabel='Coefficient name', ylabel='Coefficient value')

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

```
# Visualize the output data up to "2011-01"
fig, axs = plt.subplots(2, 1, figsize=(10, 5))
y.loc[:'2011-01'].plot(ax=axs[0])

# Run the function to visualize model's coefficients
visualize_coefficients(model.coef_, prices_perc_shifted.columns, ax=axs[1])
plt.show()
```





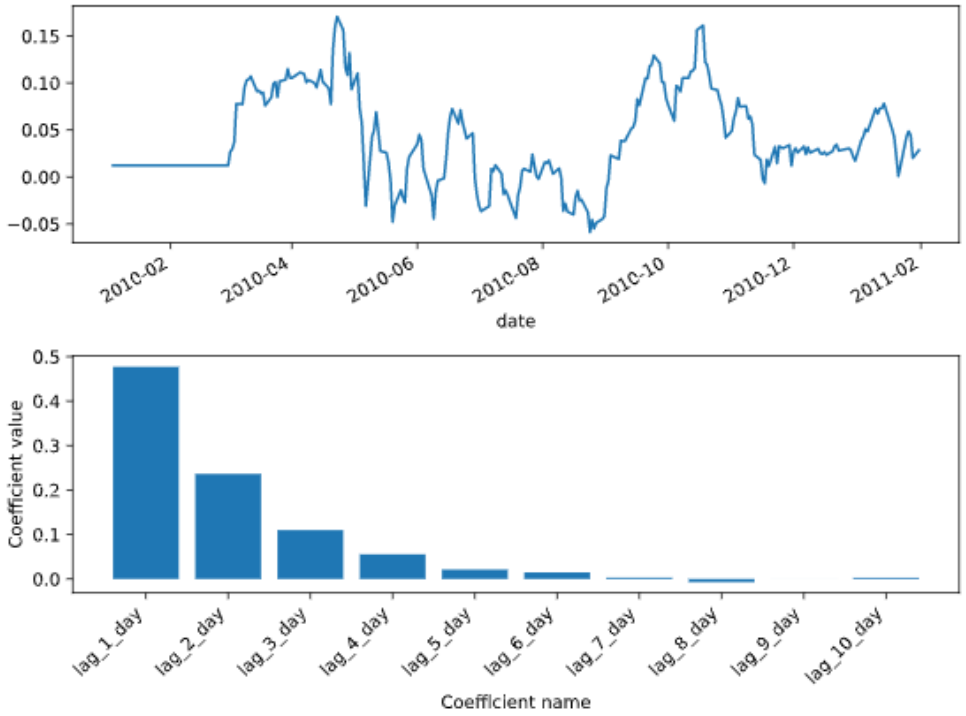
When you use time-lagged features on the raw data, you see that the highest coefficient by far is the first one. This means that the N-1th time point is useful in predicting the Nth timepoint, but no other points are useful.

Auto-regression with a smoother time series

Now, let's re-run the same procedure using a smoother signal. You'll use the same *percent change* algorithm as before, but this time use a much larger window (40 instead of 20). As the window grows, the difference between neighboring timepoints gets smaller, resulting in a *smoother* signal. What do you think this will do to the auto-regressive model?

`prices_perc_shifted` and `model` (updated to use a window of 40) are available in your workspace.

```
# Visualize the output data up to "2011-01"
fig, axs = plt.subplots(2, 1, figsize=(10, 5))
y.loc[:'2011-01'].plot(ax=axs[0])

# Run the function to visualize model's coefficients
visualize_coefficients(model.coef_, prices_perc_shifted.columns, ax=axs[1])
plt.show()
```



As you can see here, by transforming your data with a larger window, you've also changed the relationship between each timepoint and the ones that come just before it. This model's coefficients gradually go down to zero, which means that the signal itself is smoother over time. Be careful when you see something like this, as it means your data is not i.i.d.

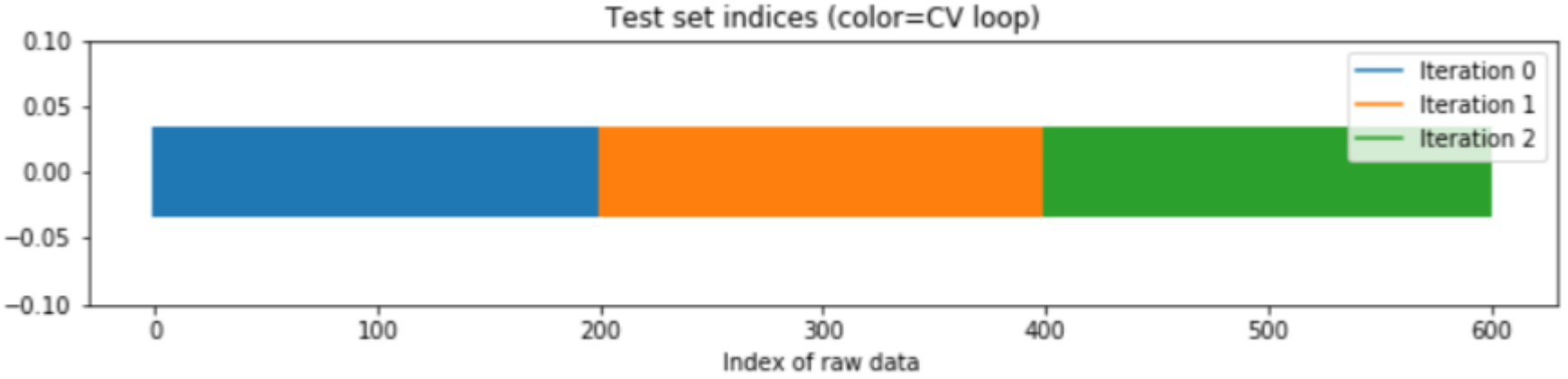
Cross-validating time series data

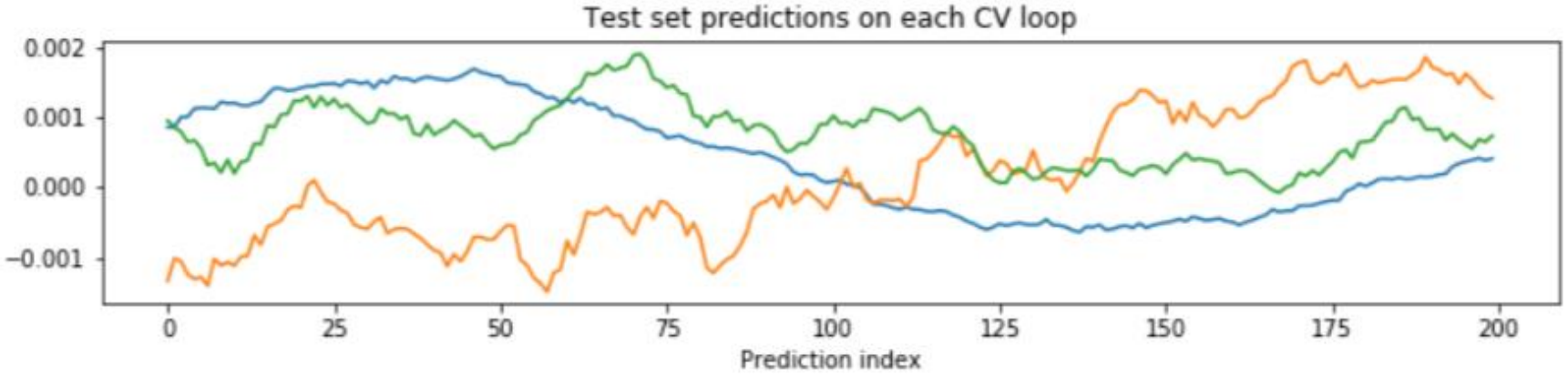
```
# Iterating over the "split" method yields train/test indices
for tr, tt in cv.split(X, y):
    model.fit(X[tr], y[tr])
    model.score(X[tt], y[tt])
```

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

```
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')
```

Visualising KFold CV bahaviour:

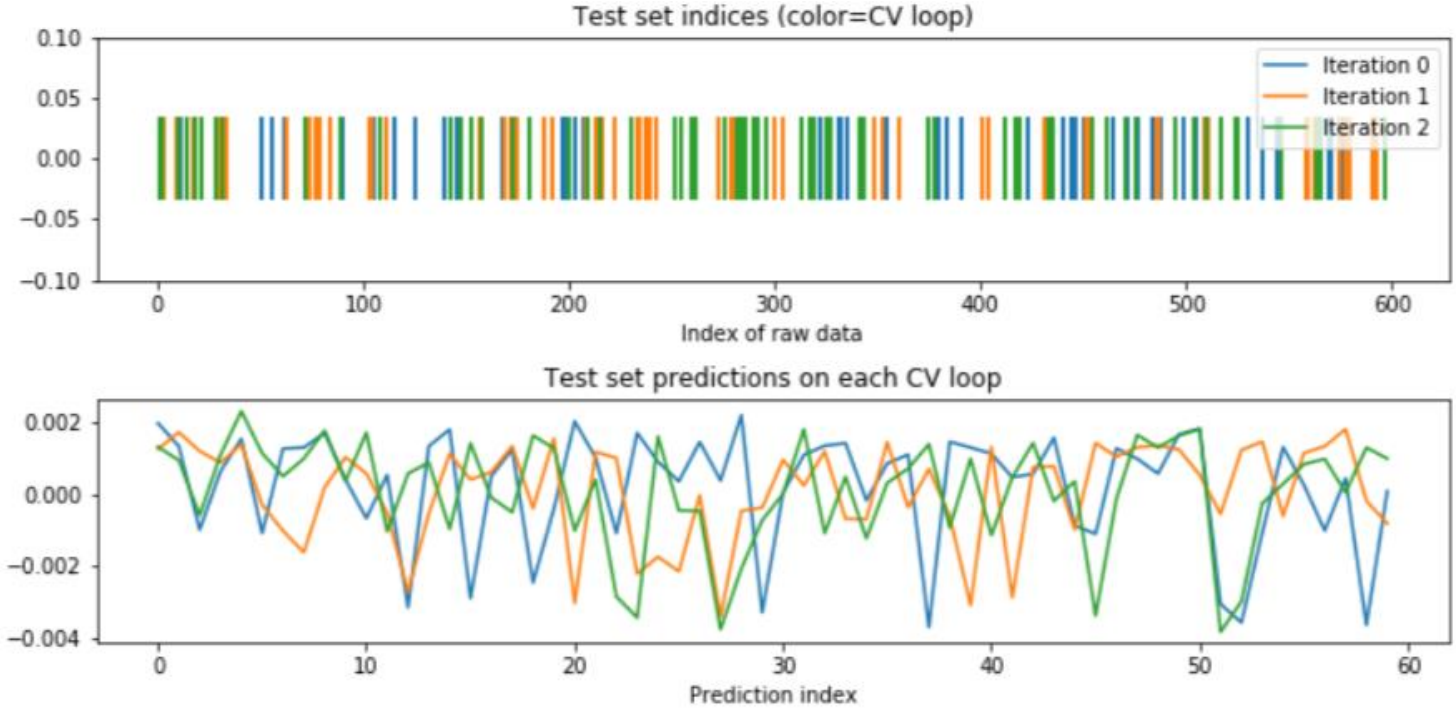




Timeseries data is not i.i.d., therefore do NOT shuffle data when making predictions. For example, we use ShuffleSplit cross-validation iterator, which randomly permutates the data labels in each iteration.

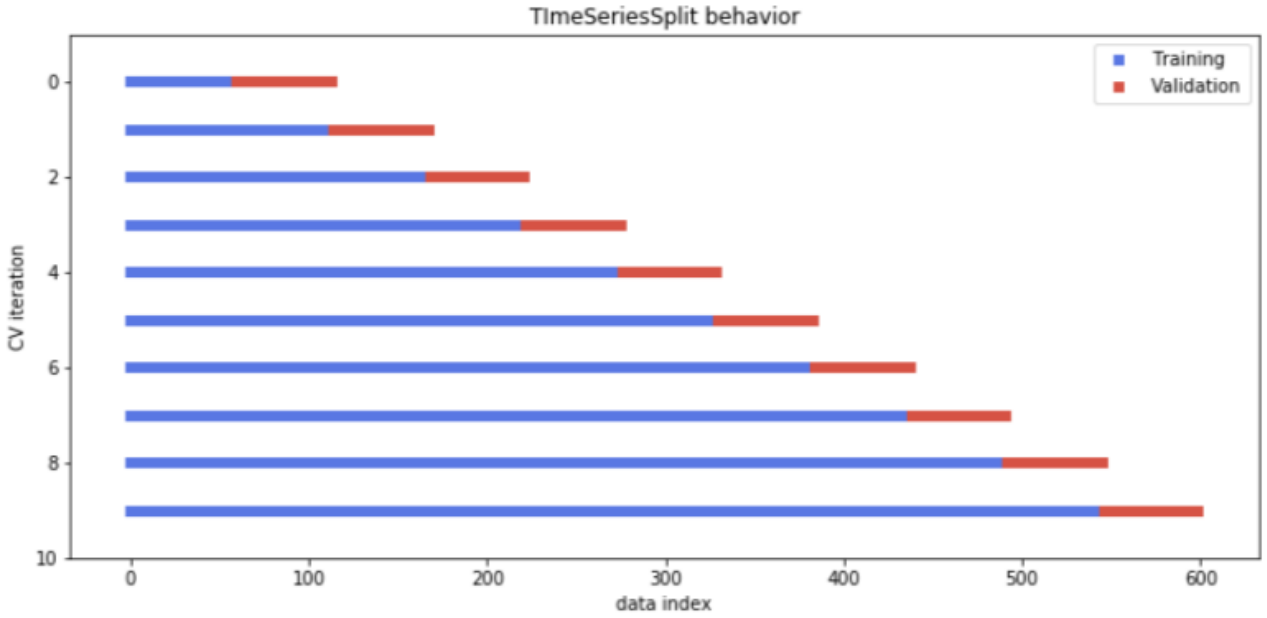
```
from sklearn.model_selection import ShuffleSplit

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

The output data no longer looks like a timeseries, because the temporal structure of the data has been destroyed. Some train info is now in the validation set, and you can no longer trust the score of your model.



The time series CV iterator always uses data from the past to predict timepoints in the future. Through CV iterations, a larger amount of training data is used to predict the next block of validation data, corresponding to the fact that more time has passed.

```
# 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'])
```



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
```

```
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
```

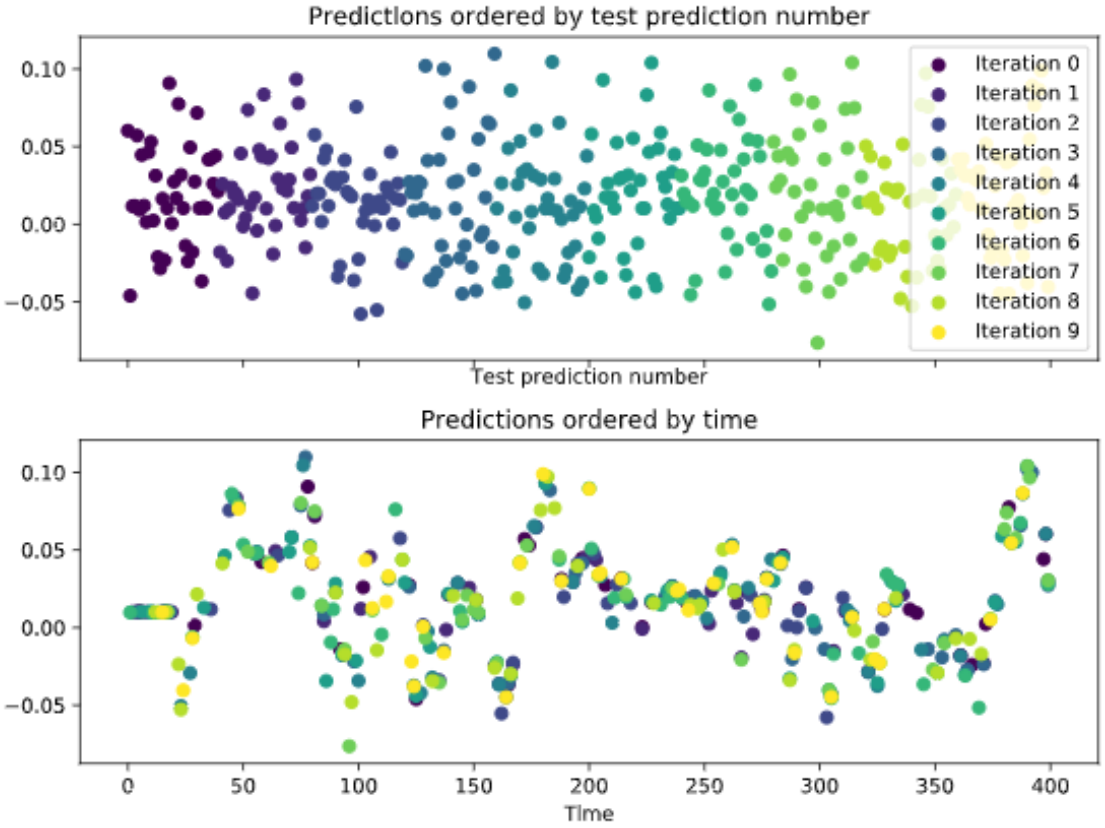
Cross-validation with shuffling

As you'll recall, cross-validation is the process of splitting your data into training and test sets multiple times. Each time you do this, you choose a *different* training and test set. In this exercise, you'll perform a traditional `ShuffleSplit` cross-validation on the company value data from earlier. Later we'll cover what changes need to be made for time series data. The data we'll use is the same historical price data for several large companies.

An instance of the Linear regression object (`model`) is available in your workspace along with the function `r2_score()` for scoring. Also, the data is stored in arrays `X` and `y`. We've also provided a helper function (`visualize_predictions()`) to help visualize the results.

```
# Import ShuffleSplit and create the cross-validation object
from sklearn.model_selection import ShuffleSplit
cv = ShuffleSplit(n_splits=10, random_state=1)
```

```
# Iterate through CV splits
results = []
for tr, tt in cv.split(X, y):
    # Fit the model on training data
    model.fit(X[tr], y[tr])

    # Generate predictions on the test data, score the predictions, and collect
    prediction = model.predict(X[tt])
    score = r2_score(y[tt], prediction)
    results.append((prediction, score, tt))

# Custom function to quickly visualize predictions
visualize_predictions(results)
```



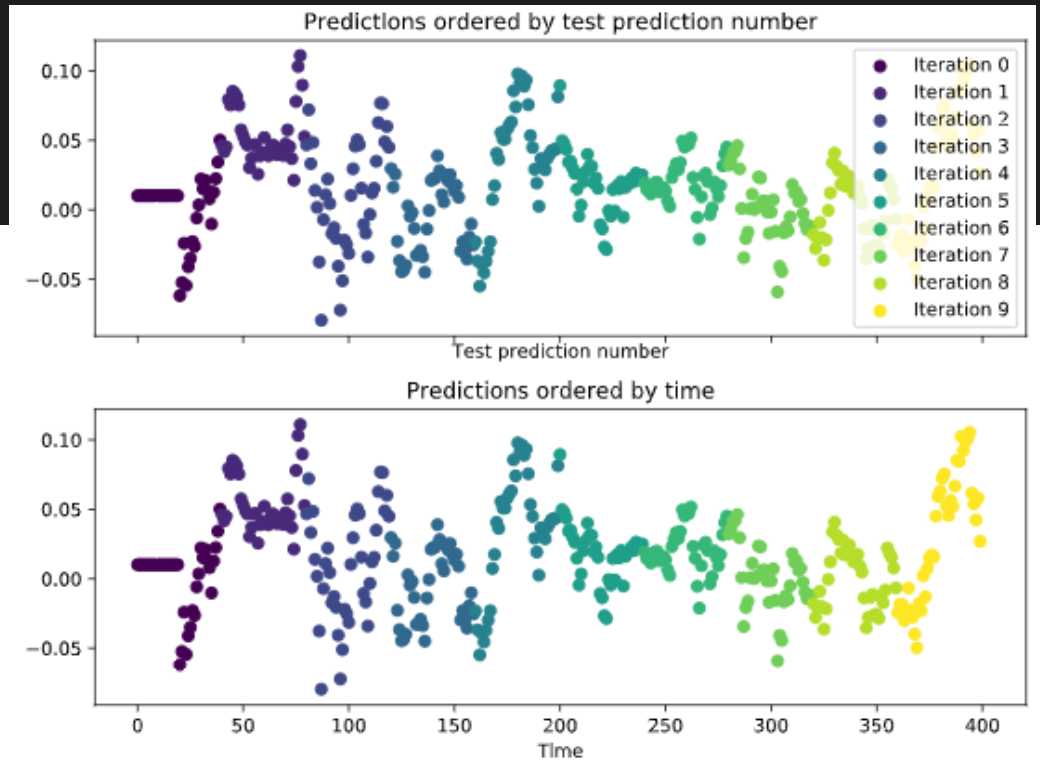
You've correctly constructed and fit the model. If you look at the plot, see that the order of datapoints in the test set is scrambled.

Cross-validation without shuffling

Now, re-run your model fit using block cross-validation (without shuffling all datapoints). In this case, neighboring time-points will be kept close to one another. How do you think the model predictions will look in each cross-validation loop?

An instance of the Linear regression `model` object is available in your workspace. Also, the arrays `X` and `y` (training data) are available too.

```
# Create KFold cross-validation object
from sklearn.model_selection import KFold
cv = KFold(n_splits=10, shuffle=False, random_state=1)

# Iterate through CV splits
results = []
for tr, tt in cv.split(X, y):
    # Fit the model on training data
    model.fit(X[tr], y[tr])

    # Generate predictions on the test data and collect
    prediction = model.predict(X[tt])
    results.append((prediction, tt))

# Custom function to quickly visualize predictions
visualize_predictions(results)
```



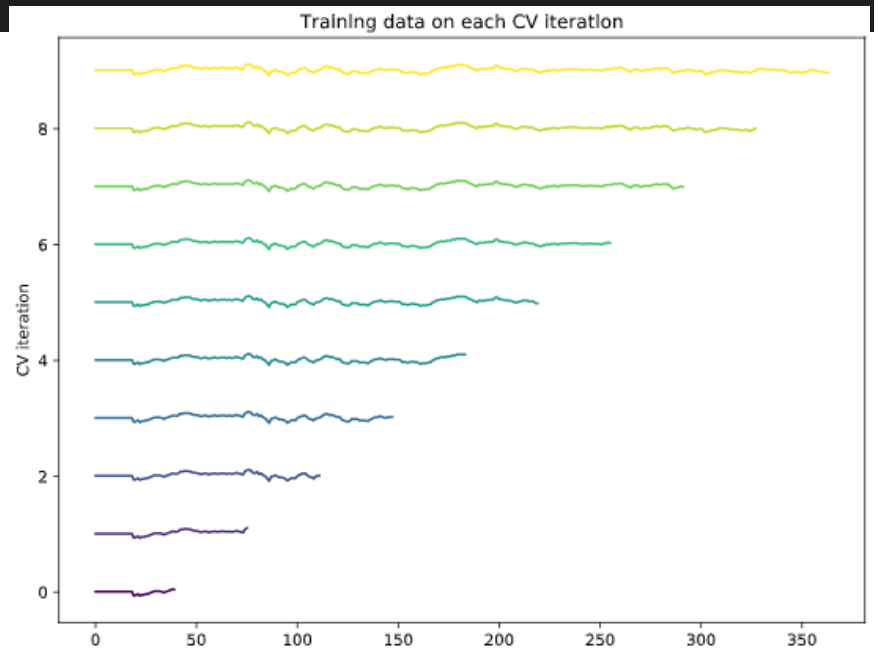
This time, the predictions generated within each CV loop look 'smoother' than they were before - they look more like a real time series because you didn't shuffle the data. This is a good sanity check to make sure your CV splits are correct.

Time-based cross-validation

Finally, let's visualize the behavior of the *time series cross-validation iterator* in scikit-learn. Use this object to iterate through your data one last time, visualizing the training data used to fit the model on each iteration.

An instance of the Linear regression `model` object is available in your workpsace. Also, the arrays `X` and `y` (training data) are available too.

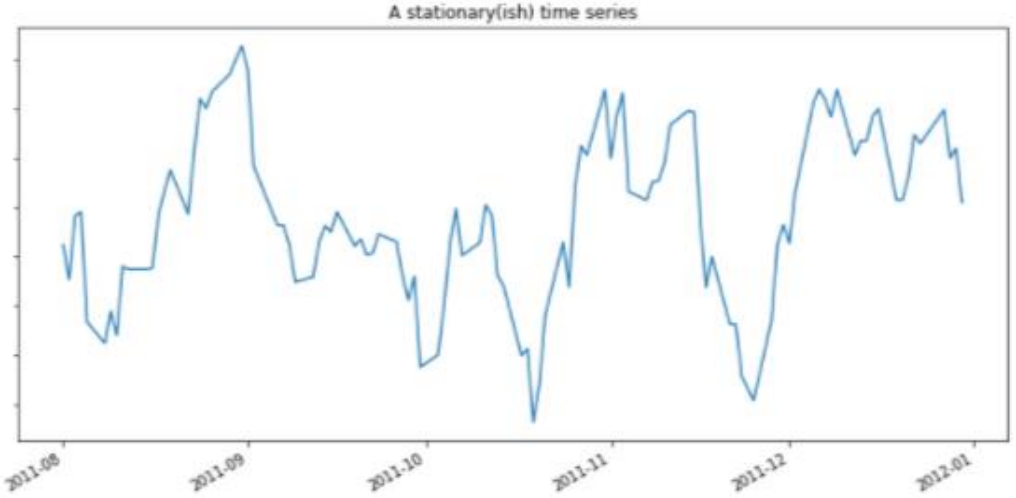
```
# Import TimeSeriesSplit
from sklearn.model_selection import TimeSeriesSplit

# Create time-series cross-validation object
cv = TimeSeriesSplit(n_splits=10)

# Iterate through CV splits
fig, ax = plt.subplots()
for ii, (tr, tt) in enumerate(cv.split(X, y)):
    # Plot the training data on each iteration, to see the behavior of the CV
    ax.plot(tr, ii + y[tr])

ax.set(title='Training data on each CV iteration', ylabel='CV iteration')
plt.show()
```



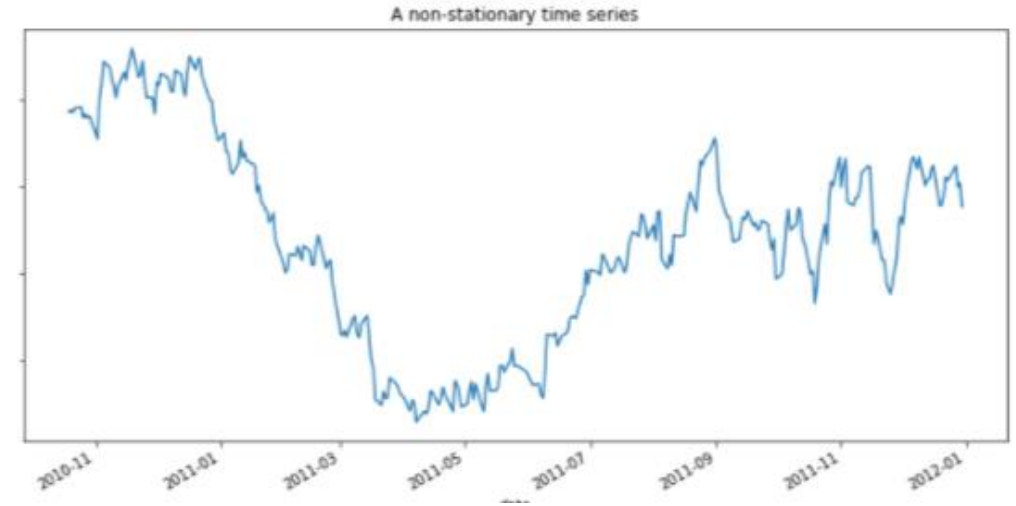
Note that the size of the training set grew each time when you used the time series cross-validation object. This way, the time points you predict are always *after* the timepoints we train on.

Stationarity and stability

Stationarity = do not change statistical (example: mean, stdev, trends) properties over time. Most timeseries are non-stationary to some extent.



A stationary timeseries generally does not change its structure. Its variability is constant throughout time.



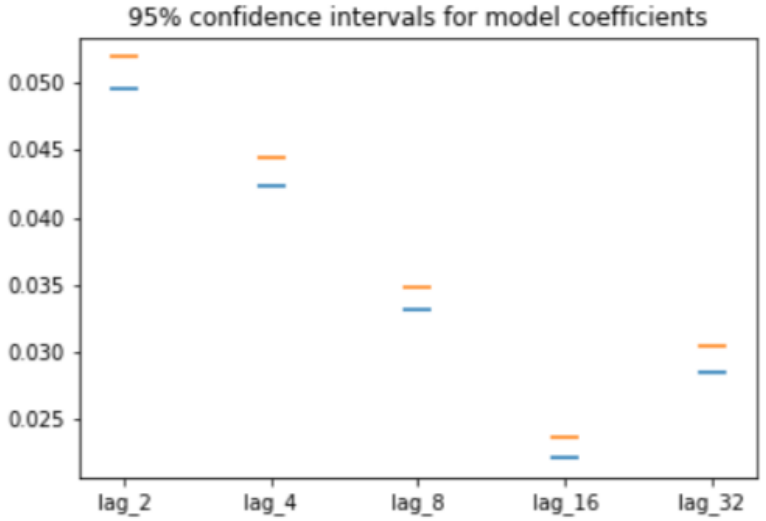
A non-stationary timeseries has its variance and trends change over time, like in all real-world data.

Stability = relationship between inputs and outputs is static. But not the case for non-stationary timeseries. To use cross-validation, which yields a set of model coefficients per iteration. Then quantify the variability of these coefficients across iterations.

Bootstrapping the mean to assess variability, to estimate the confidence interval of the mean of each coefficient.

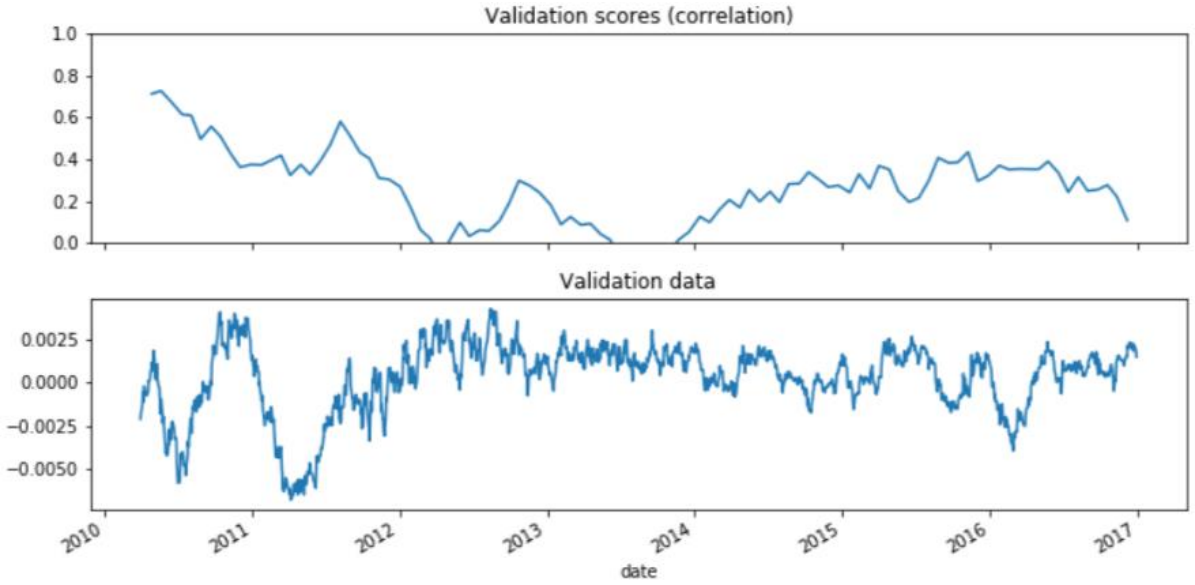
```
from sklearn.utils import resample

# 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)
```

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



```python
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)
```

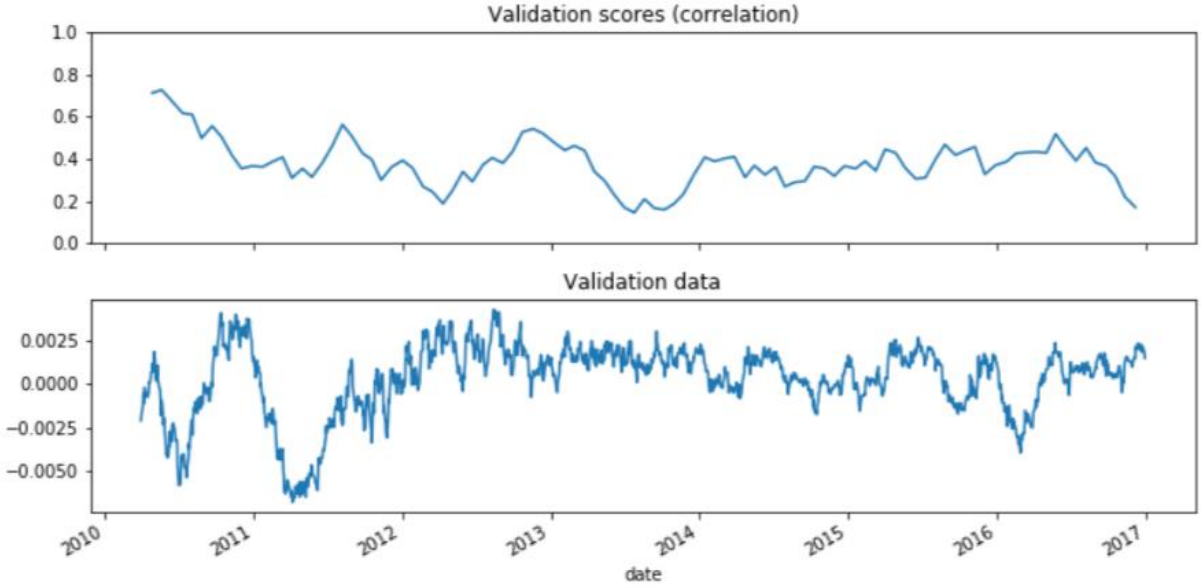
```python
fig, axs = plt.subplots(2, 1, figsize=(10, 5), sharex=True)

# 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')
```



There is a clear dip in the middle, probably because the statistics of the data changed.

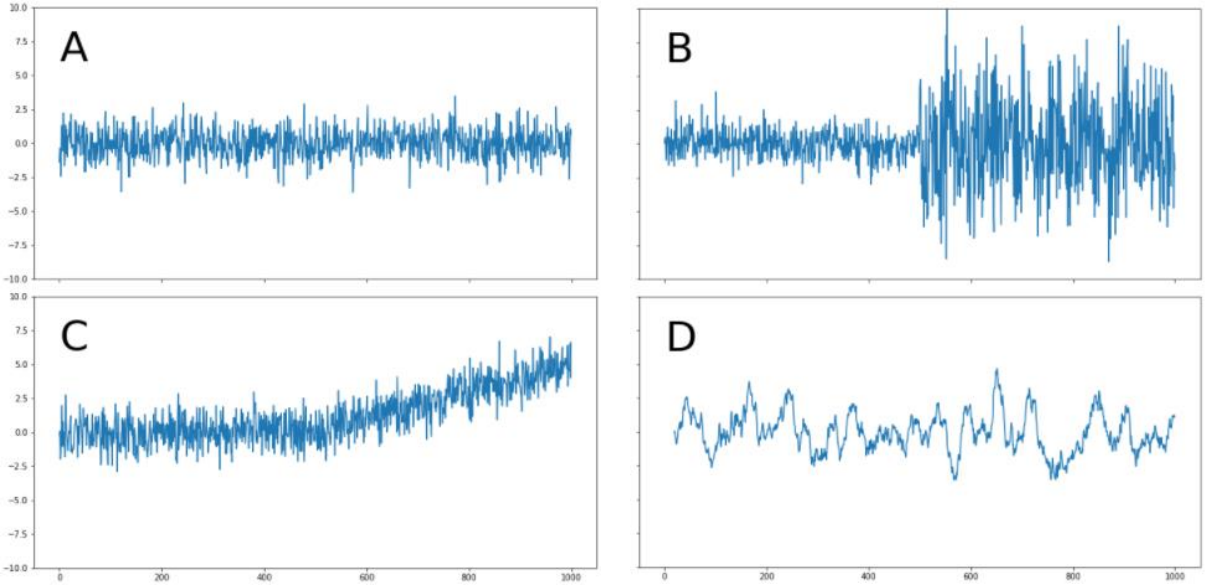
To overcome this, restrict the size of the training window, to ensure that only the latest timepoints are used in the training. Use fixed windows with timeseries cross-validation.

```python
# 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)
```



Stationarity

First, let's confirm what we know about stationarity. Take a look at these time series. Which of the following time series do you think are not stationary?



Possible Answers

- C only
- D and C
- B only
- B and C

Answer: B & C

C begins to trend upward partway through, while B shows a large increase in variance mid-way through, making both of them non-stationary.

Bootstrapping a confidence interval

A useful tool for assessing the variability of some data is the bootstrap. In this exercise, you'll write your own bootstrapping function that can be used to return a bootstrapped confidence interval.

This function takes three parameters: a 2-D array of numbers (`data`), a list of percentiles to calculate (`percentiles`), and the number of boostrap iterations to use (`n_boots`). It uses the `resample` function to generate a bootstrap sample, and then repeats this many times to calculate the confidence interval.

```
from sklearn.utils import resample

def bootstrap_interval(data, percentiles=(2.5, 97.5), n_boots=100):
    """Bootstrap a confidence interval for the mean of columns of a 2-D dataset."""
    # Create empty array to fill the results
    bootstrap_means = np.zeros([n_boots, data.shape[-1]])
    for ii in range(n_boots):
        # Generate random indices for data *with* replacement, then take the sample mean
        random_sample = resample(data)
        bootstrap_means[ii] = random_sample.mean(axis=0)

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

You can use this function to assess the variability of your model coefficients.

Calculating variability in model coefficients

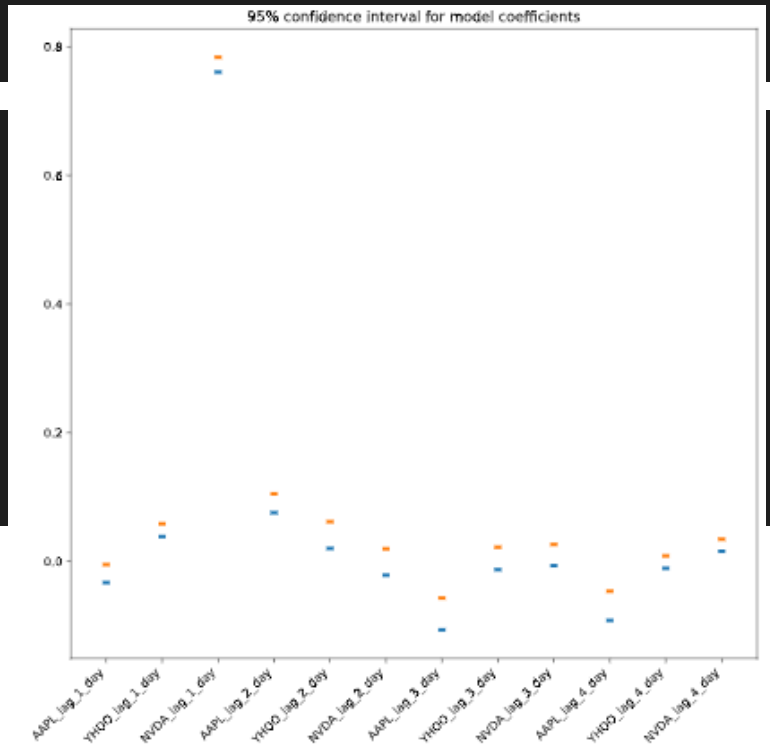
In this lesson, you'll re-run the cross-validation routine used before, but this time paying attention to the model's stability over time. You'll investigate the coefficients of the model, as well as the uncertainty in its predictions.

Begin by assessing the *stability* (or uncertainty) of a model's coefficients across multiple CV splits. Remember, the coefficients are a reflection of the pattern that your model has found in the data.

An instance of the Linear regression object (`model`) is available in your workpsace. Also, the arrays `X` and `y` (the data) are available too.

```
# Iterate through CV splits
n_splits = 100
cv = TimeSeriesSplit(n_splits=n_splits)

# Create empty array to collect coefficients
coefficients = np.zeros([n_splits, X.shape[1]])

for ii, (tr, tt) in enumerate(cv.split(X, y)):
    # Fit the model on training data and collect the coefficients
    model.fit(X[tr], y[tr])
    coefficients[ii] = model.coef_
```

```
# Calculate a confidence interval around each coefficient
bootstrapped_interval = bootstrap_interval(coefficients)

# Plot it
fig, ax = plt.subplots()
ax.scatter(feature_names, bootstrapped_interval[0], marker='_', lw=3)
ax.scatter(feature_names, bootstrapped_interval[1], marker='_', lw=3)
ax.set(title='95% confidence interval for model coefficients')
plt.setp(ax.get_xticklabels(), rotation=45, horizontalalignment='right')
plt.show()
```



You've calculated the variability around each coefficient, which helps assess which coefficients are more stable over time!

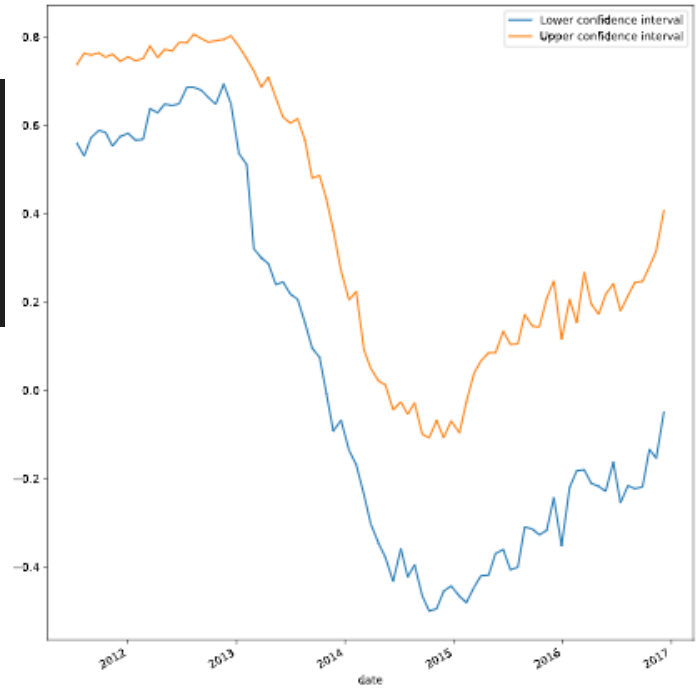
Visualizing model score variability over time

Now that you've assessed the variability of each coefficient, let's do the same for the performance (scores) of the model. Recall that the `TimeSeriesSplit` object will use successively-later indices for each test set. This means that you can treat the *scores* of your validation as a time series. You can visualize this over time in order to see how the model's performance changes over time.

An instance of the Linear regression model object is stored in `model`, a cross-validation object in `cv`, and data in `X` and `y`.

```
from sklearn.model_selection import cross_val_score

# Generate scores for each split to see how the model performs over time
scores = cross_val_score(model, X, y, cv=cv, scoring=my_pearsonr)

# Convert to a Pandas Series object
scores_series = pd.Series(scores, index=times_scores, name='score')

# Bootstrap a rolling confidence interval for the mean score
scores_lo = scores_series.rolling(20).aggregate(partial(bootstrap_interval, percentiles=2.5))
scores_hi = scores_series.rolling(20).aggregate(partial(bootstrap_interval, percentiles=97.5))
```

```
# Plot the results
fig, ax = plt.subplots()
scores_lo.plot(ax=ax, label="Lower confidence interval")
scores_hi.plot(ax=ax, label="Upper confidence interval")
ax.legend()
plt.show()
```



You plotted a rolling confidence interval for scores over time.

This is useful in seeing when your model predictions are correct.

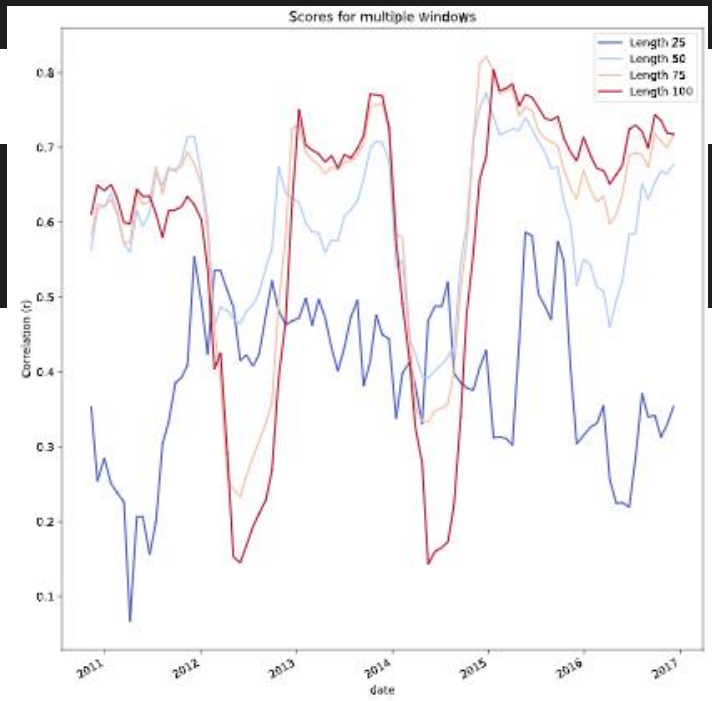
Accounting for non-stationarity

In this exercise, you will again visualize the variations in model scores, but now for data that changes its statistics over time.

An instance of the Linear regression model object is stored in `model`, a cross-validation object in `cv`, and the data in `X` and `y`.

```
# Pre-initialize window sizes
window_sizes = [25, 50, 75, 100]

# Create an empty DataFrame to collect the stores
all_scores = pd.DataFrame(index=times_scores)

# Generate scores for each split to see how the model performs over time
for window in window_sizes:
    # Create cross-validation object using a limited lookback window
    cv = TimeSeriesSplit(n_splits=100, max_train_size=window)

    # Calculate scores across all CV splits and collect them in a DataFrame
    this_scores = cross_val_score(model, X, y, cv=cv, scoring=my_pearsonr)
    all_scores['Length {}'.format(window)] = this_scores
```

```
# Visualize the scores
ax = all_scores.rolling(10).mean().plot(cmap=plt.cm.coolwarm)
ax.set(title='Scores for multiple windows', ylabel='Correlation (r)')
plt.show()
```



Wonderful - notice how in some stretches of time, longer windows perform worse than shorter ones. This is because the statistics in the data have changed, and the longer window is now using outdated information.

Course completed!

Wrap-up on the topics covered:

- Applying machine learning concepts to timeseries data, eg. Classifying heartbeat sounds as normal or abnormal, building predictive models of a company's value over time
- Always visualise your raw data first
- Feature extraction, collapsing a timeseries array into a single summary statistic, and combining many statistics of a timeseries into a single feature matrix for timeseries classification
- Spectrogram for feature extraction, only for timeseries data
- Generating model predictions that change over time, using regression features such as time-shifted versions of an input
- Cleaning and improving timeseries data
- Validating ML models with timeseries data, using cross-validation without shuffling the data but with CV iterator that is unique to timeseries data
- Calculate and visualise model stability over subsets of data
- Advanced window functions can improve the "rolling window" statistics for signal processing
- Spectral analysis
- Advanced timeseries feature extraction (eg. tsfresh)
- Production-ready pipelines for timeseries analysis

Happy learning!