Reference

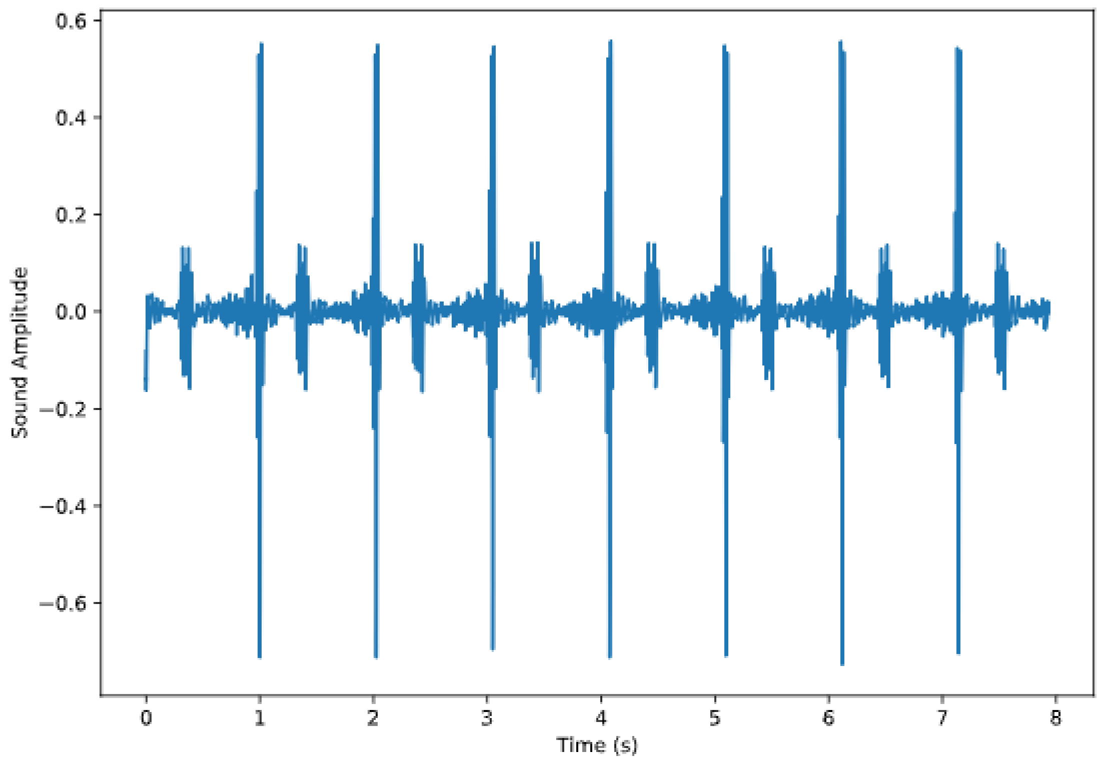
Machine Learning for Time Series Data in Python: DataCamp course

Time Series and Machine Learning Primer

Inspecting the classification data

- Explore a dataset for 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.
- This dataset is ideal for classification and was originally offered as a part of a public Kaggle competition.

In [ ]:
```
import librosa as lr
from glob import glob

# List all the wav files in the folder
audio_files = glob(data_dir + '/*.wav')
# audio_files: list, contains 21 file. So there are 21 similar figures as shown below.

# Read in the first audio file, create the time array
audio, sfreq = lr.load(audio_files[0])
#len(audio): 174980; type(audio): numpy.ndarray; sfreq: int, 22050, should be number of sampling per second.
time = np.arange(0, len(audio)) / sfreq
# type(time): numpy.ndarray; len(time): 174980
# (0, len(audio))/sfreq = (0,174980)/22050 = (0, 7.9355)

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

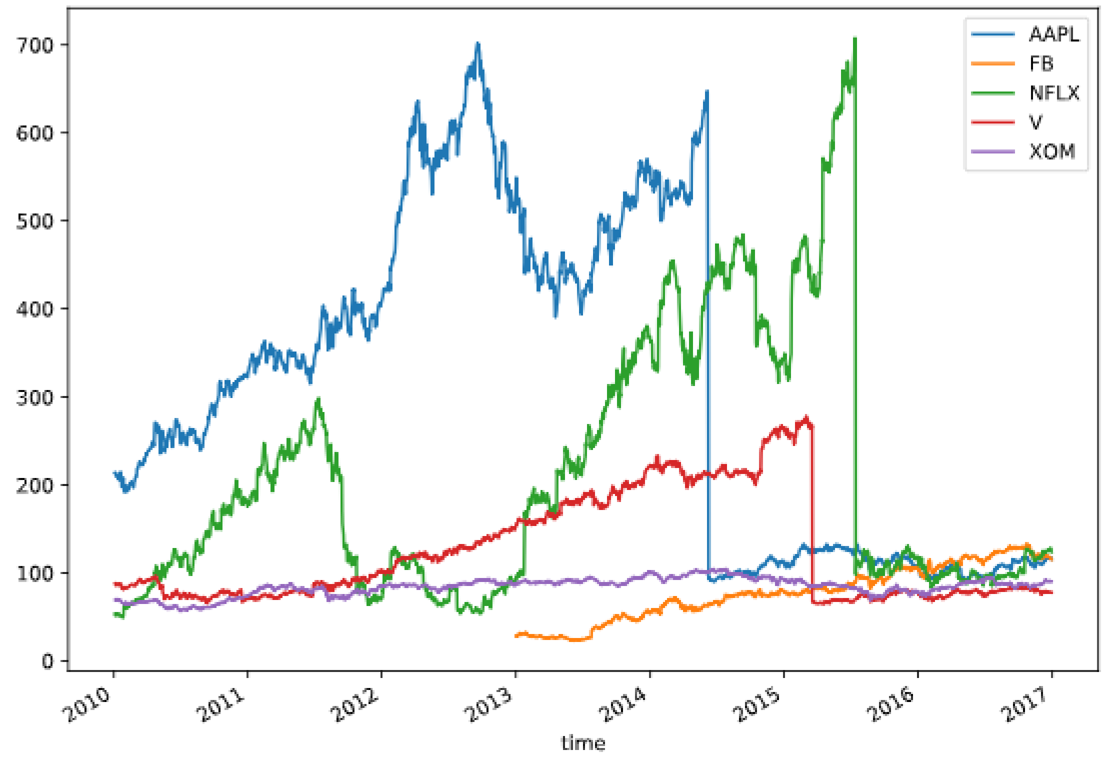


There are several seconds of heartbeat sounds here , where most of this time is in silence. A common procedure in machine learning is to separate the data points with lots of stuff happening from the ones that don't.

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 [ ]:
```
data = pd.read_csv('prices.csv', index_col=0)

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?

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.

Comments:

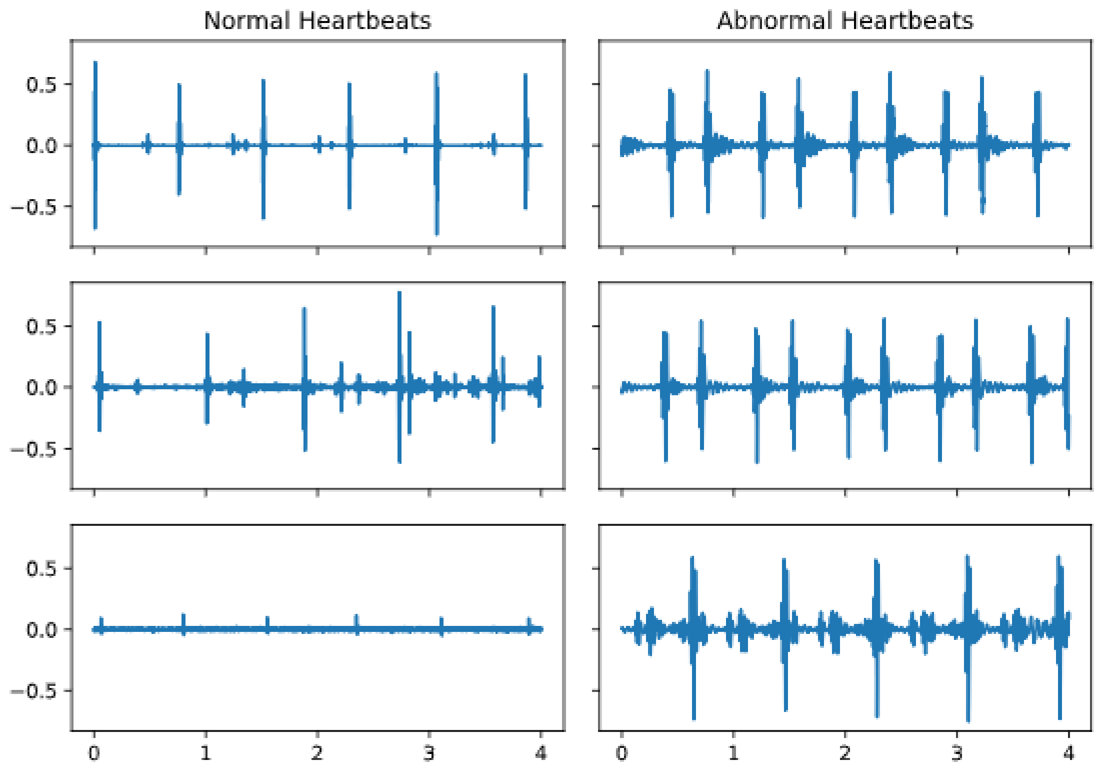
- The methods for creating feature and targets are usually very different for classification and regression (prediction) problems with time series.
- Even for regression (prediction) only, the approaches used to create feature and target can be very different. Compare the approach in this note to another note for feature-target creation under the same folder.

Many repetitions of sounds

- Start with the simplest classification technique: averaging across dimensions of a dataset and visually inspecting the result.

Two DataFrames are available, normal and abnormal, each with the shape of (n_times_points, n_audio_files) containing the audio for several heartbeats. 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.

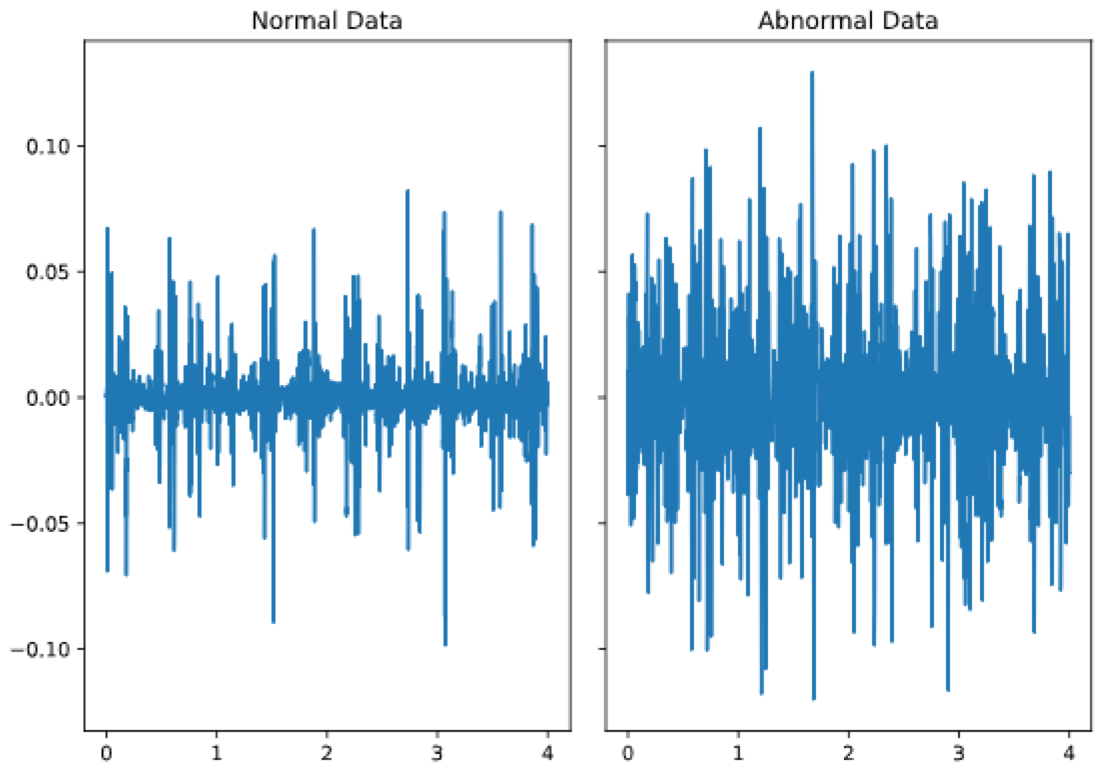
In [ ]:
```python
fig, axs = plt.subplots(3, 2, figsize=(15, 7), sharex=True, sharey=True)

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()
```



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).
- Below we average across many instances of each class of heartbeat sound.

In [ ]:
```
# Average across the time dimension 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()
```



There is indeed a noticeable difference between the two. But it's quite noisy. Let's see how you can dig into the data a bit further.

Build a classification model

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.

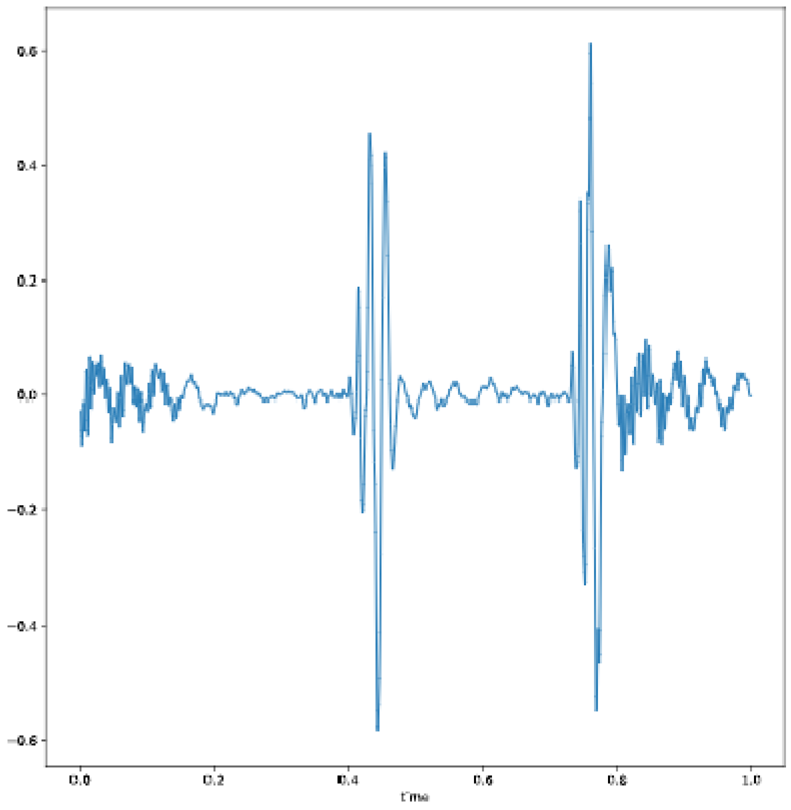
```
In [ ]: from sklearn.svm import LinearSVC

        # 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))
        # The result is 0.555555
```

The 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.

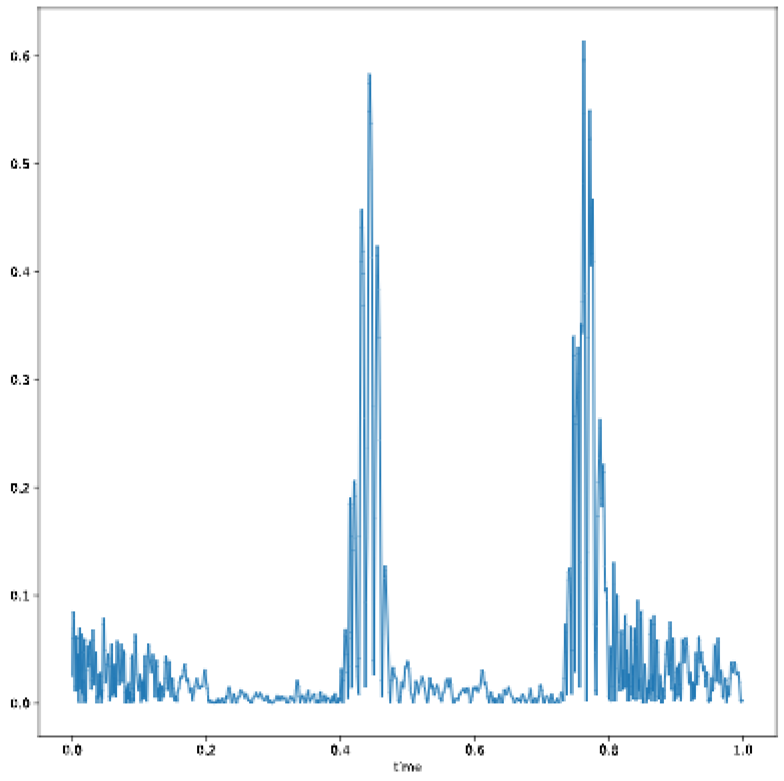
Calculating the envelope of sound

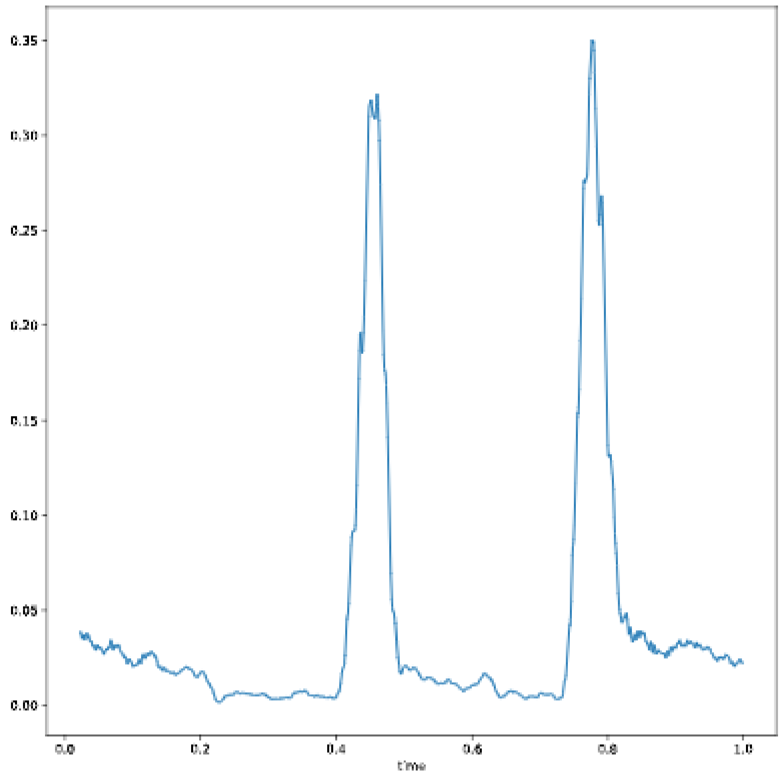
- One of the ways you can improve the features available to your model is to remove some of the noise present in the data.
- In audio data, a common way to do this is to smooth the data and then **rectify it** so that the total amount of sound energy over time is more distinguishable.

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

        # Rectify the audio signal. It means taking the absolute value.
        audio_rectified = audio.apply(np.abs)
        audio_rectified.plot(figsize=(10, 5))
        plt.show()

        # Smooth by applying a rolling mean. In Python, just one sentence to do what we do a lot in c
        audio_rectified_smooth = audio_rectified.rolling(50).mean()
        audio_rectified_smooth.plot(figsize=(10, 5))
        plt.show()
```



0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.2
0.4
0.6
0.8
1.0
time


0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8
1.0
time



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.

In [ ]:
```python
# 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))
```

```
0.714219114219
```

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.

In [ ]:
```
# 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))
```

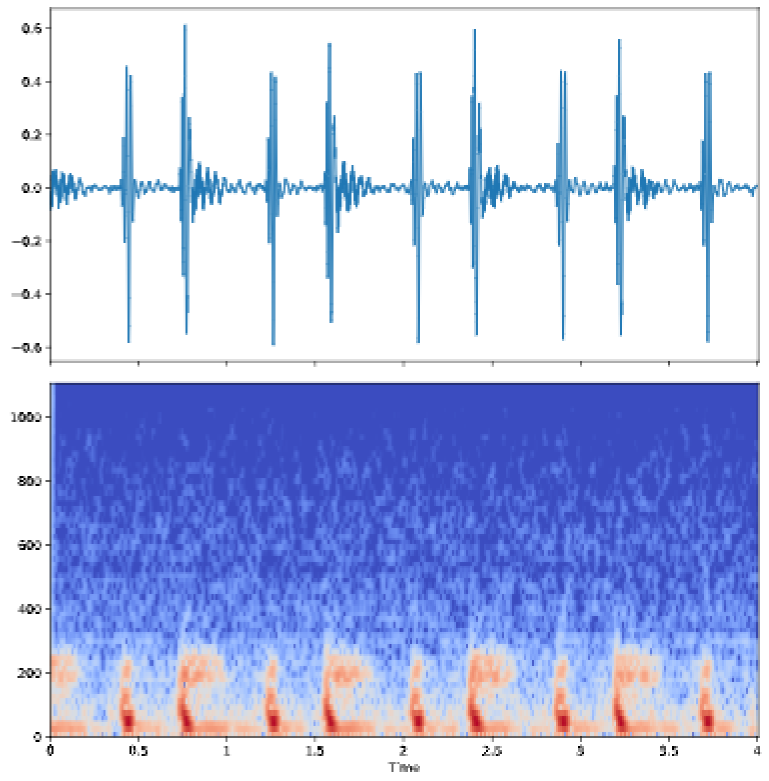
```
0.53344988345
```

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.

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.

```
In [ ]: from librosa.core import 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()
```

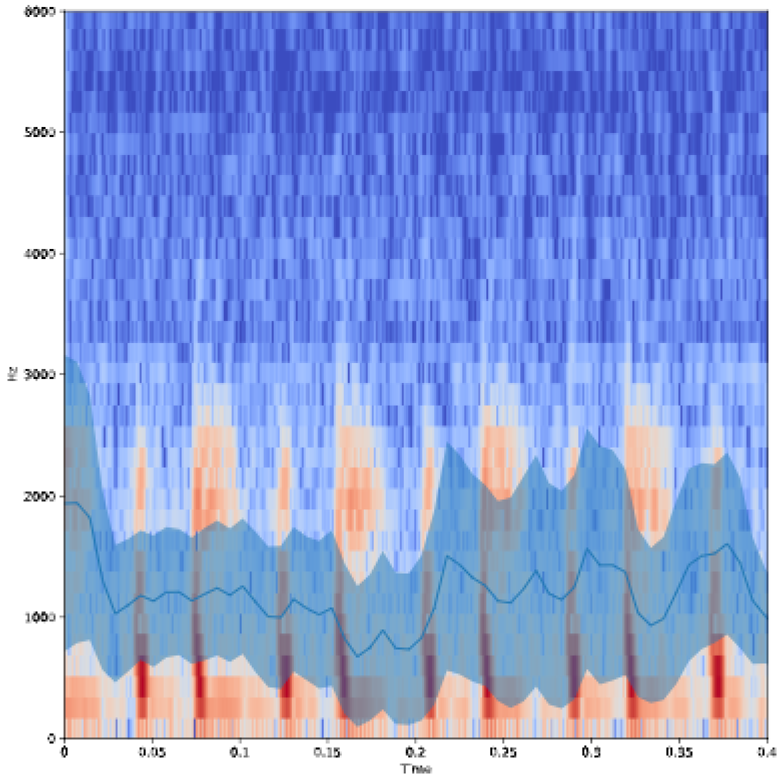


Do you notice that the heartbeats come in pairs, as seen by the vertical lines in the spectrogram?

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.

In [ ]:
```
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

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.

In [ ]:
```python
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 the 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))
```

```
0.483216783217
```

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.z To improve the accuracy, you want to find the right features that provide relevant information and also build models on much larger data.

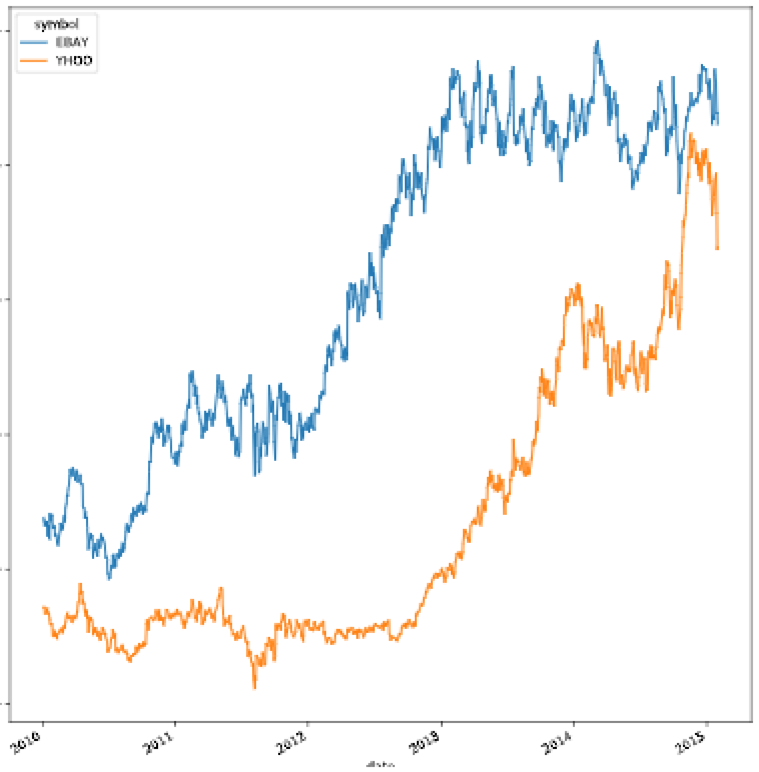
Predicting Time Series Data

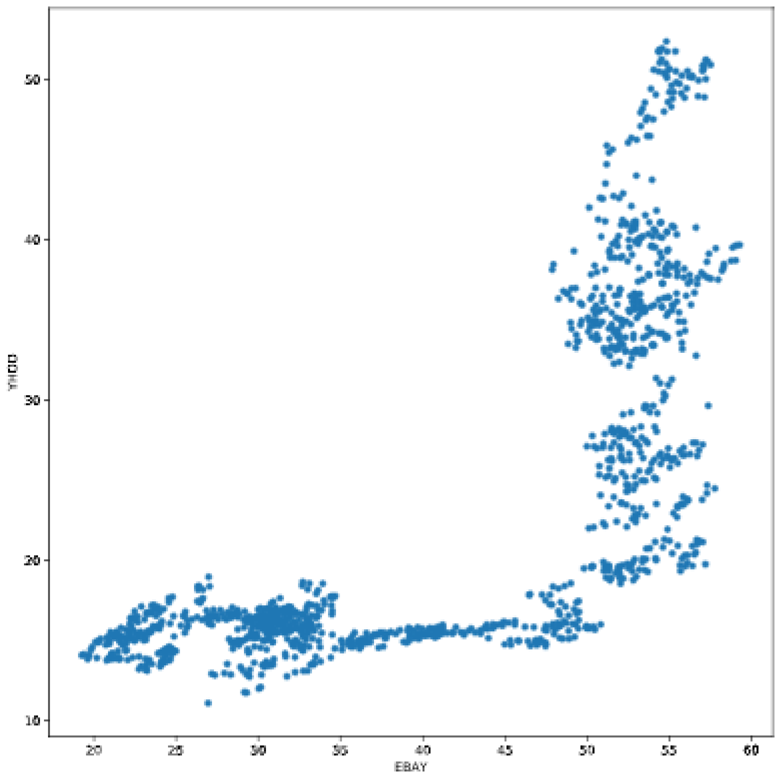
This is different from the classification of time series data in the previous chapter.

Introducing the dataset

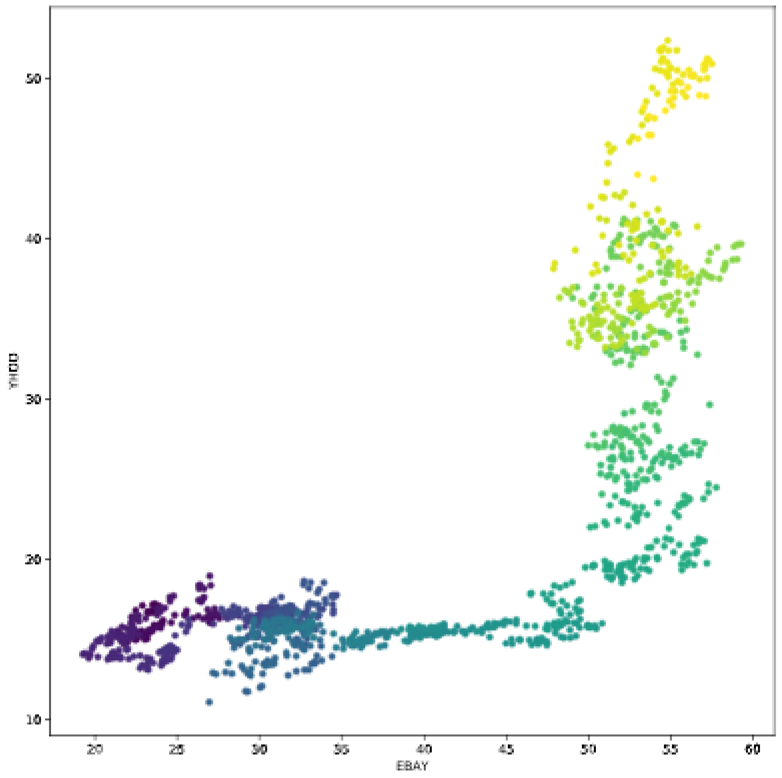
- Show historical prices from two tech companies (Ebay and Yahoo).
- Generate a scatter plot showing how the values for each company compare with one another.
- Add in a "time" dimension to the scatter plot so you can see how this relationship changes over time.

In [ ]:
```
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()
```





50
40
30
20
10
YHOO
20
25
30
35
40
45
50
55
60
EBAY



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

Comments:

- Using color as a third axis is a nice way to show time-dependent behavior.
- This is a nice way to show how correlation of two variables 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.**What is the point of doing this?** 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 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.

In [ ]:
```python
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)
```

```
[-6.09050633 -0.3179172 -3.72957284]
```

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

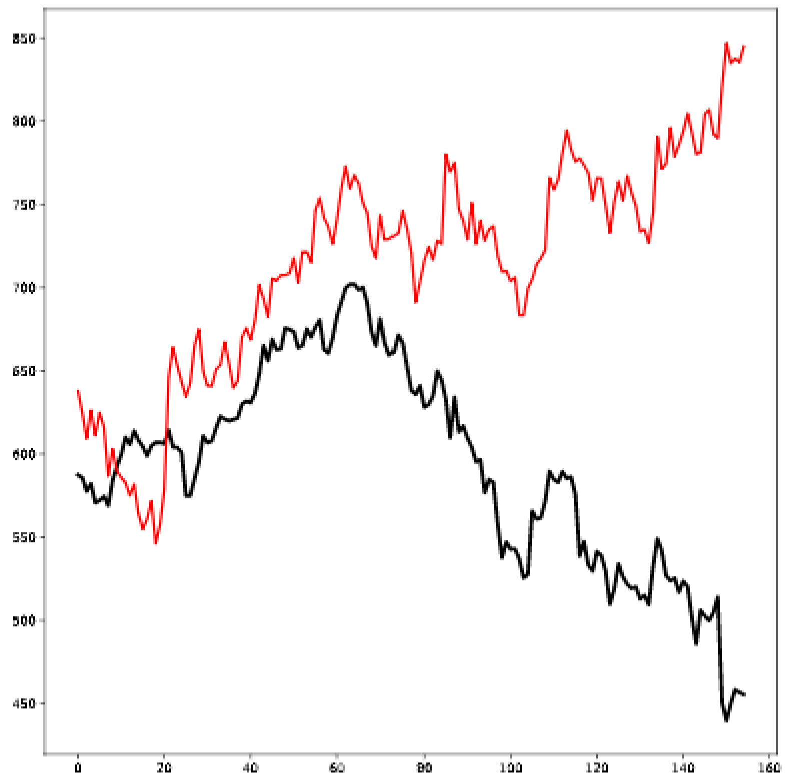
Visualizing predicted values

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.

In [ ]:
```python
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)
# output -5.70939901949
```

In [ ]:
```python
# 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.

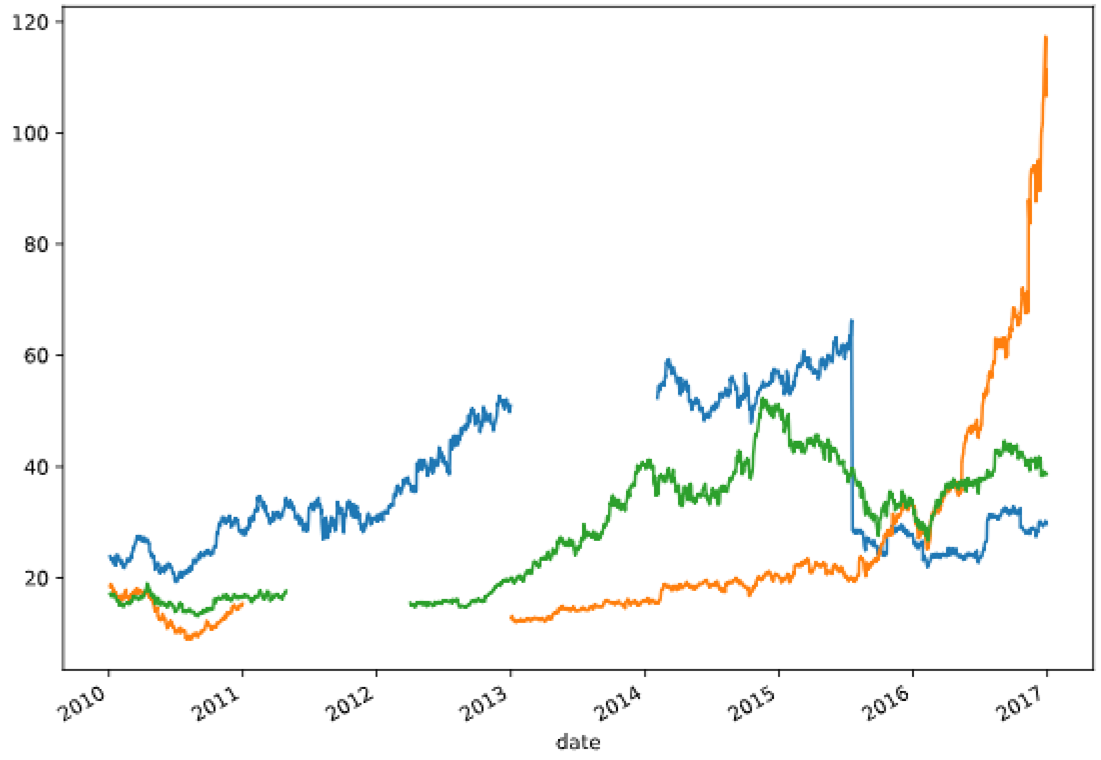
Visualizing messy data

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.

In [ ]:
```
# 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. How can you deal with this?

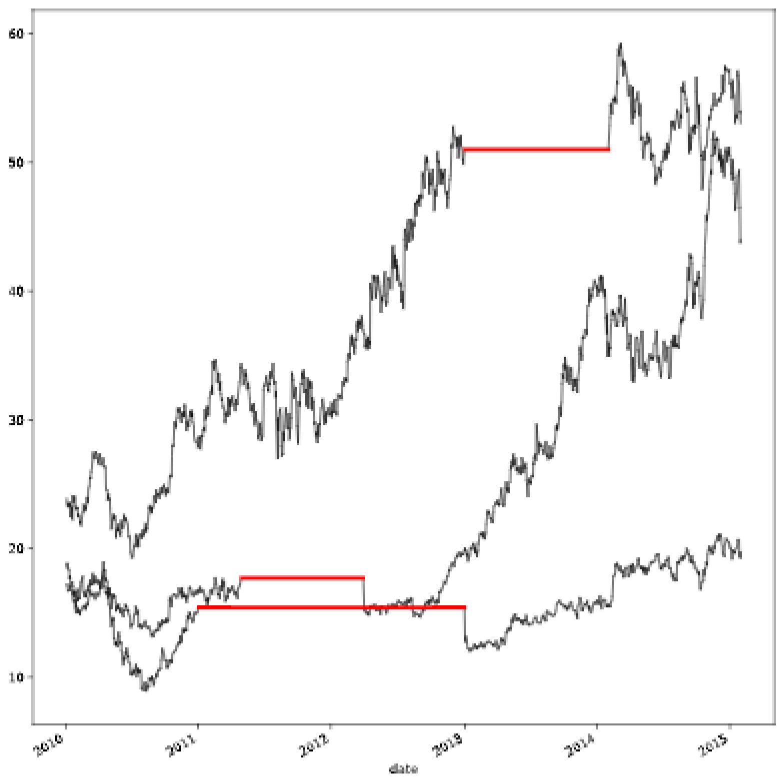
Imputing missing values

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

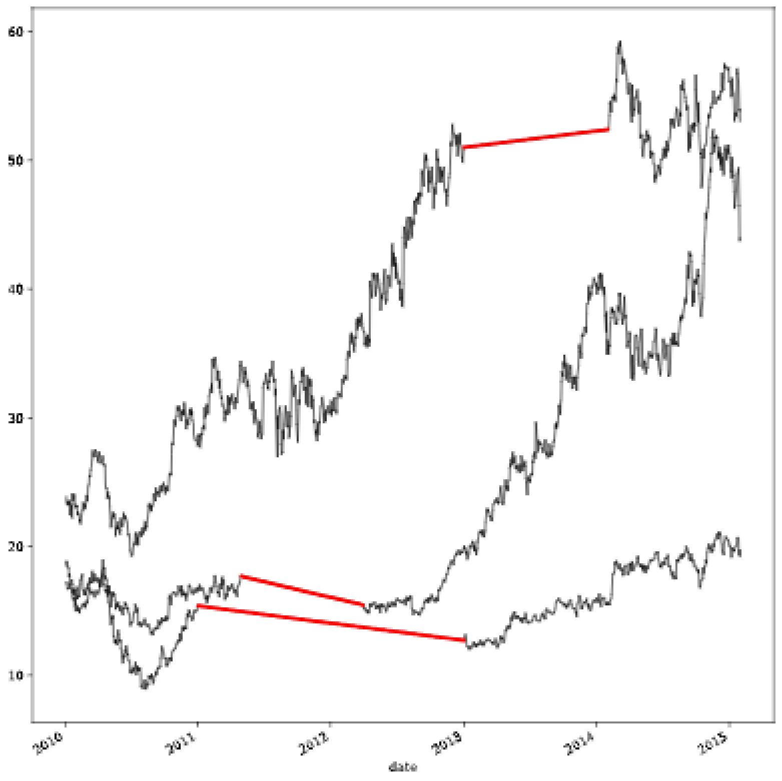
In this exercise, you'll practice using different interpolation methods to fill in some missing values, visualizing the result each time. But first, you will create the function (interpolate_and_plot()) you'll use to interpolate missing data points and plot them.

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

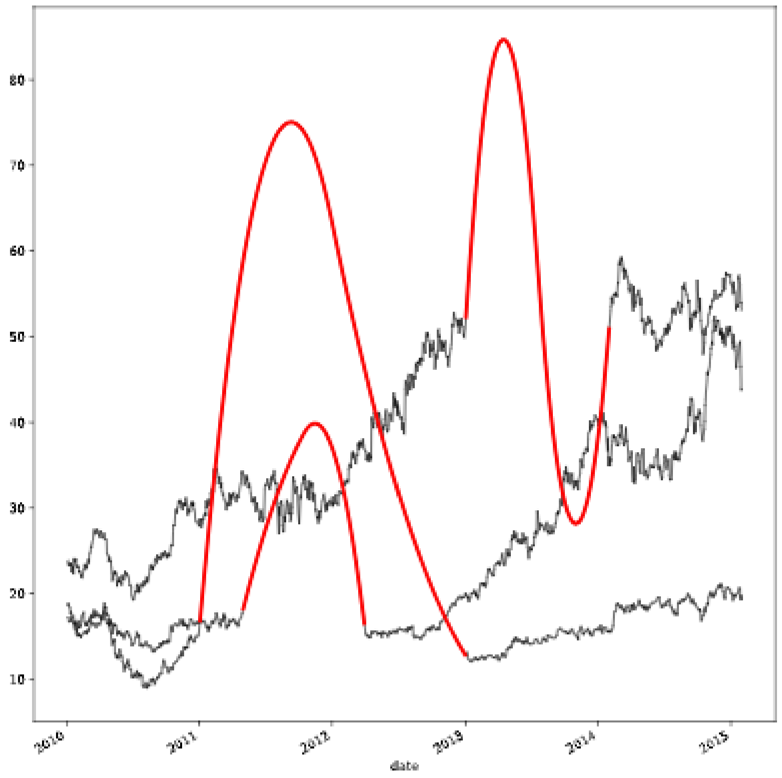
In [ ]:
```python
# Create a function we'll use to interpolate and plot
def interpolate_and_plot(prices, interpolation):

    # 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)
```



In [ ]:
```
# Interpolate with a quadratic function
interpolation_type = 'quadratic'
interpolate_and_plot(prices, interpolation_type)
```



60
50
40
30
20
10
2010
2011
2012
2013
2014
2015
date

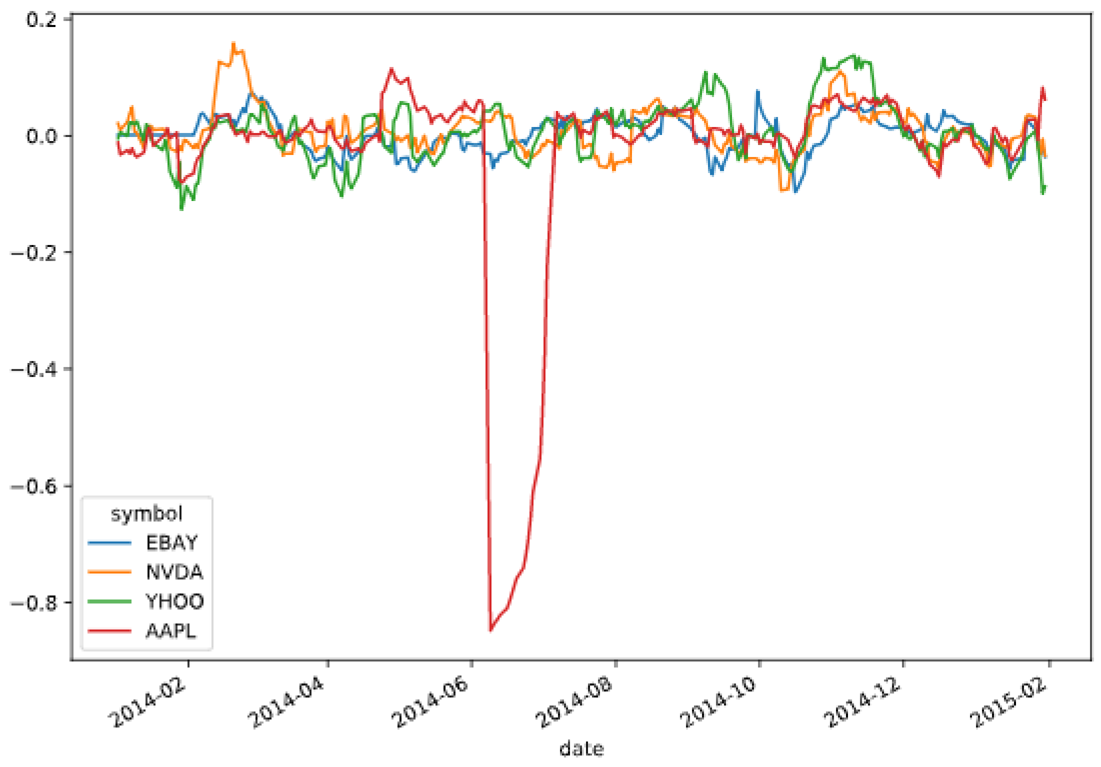


Transforming raw data

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. Also note here it is a special percent_change, but not the current one relative to the previous one.

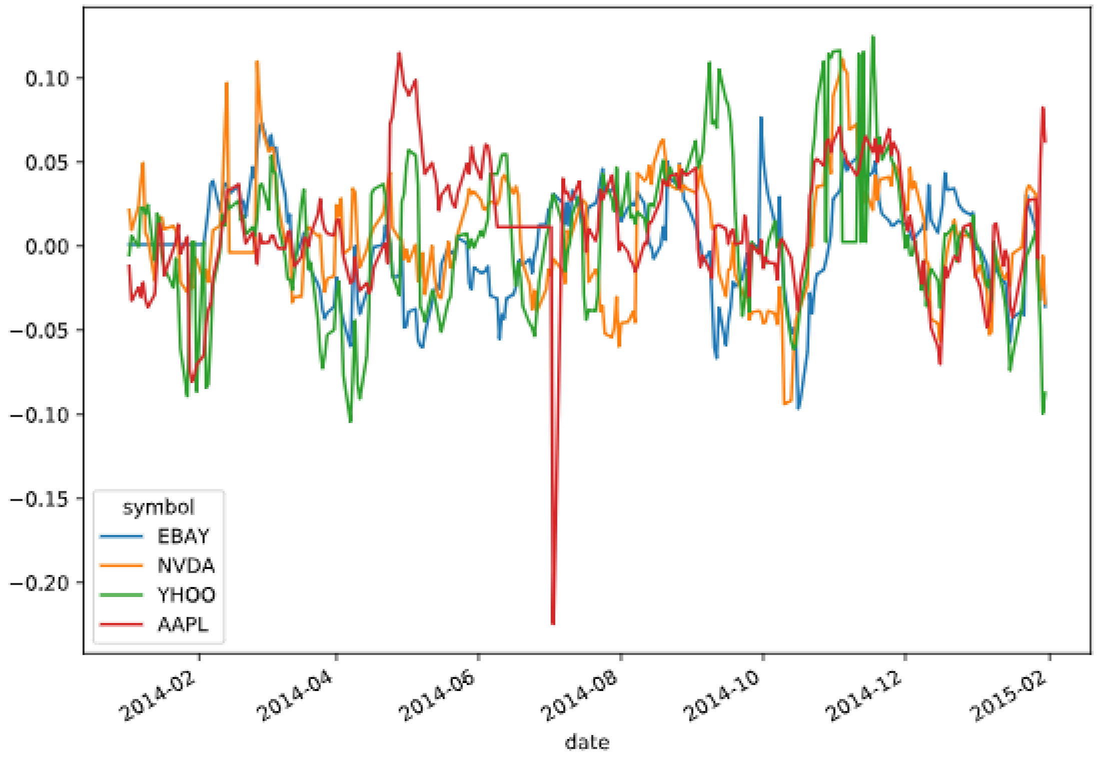
In [ ]:
```
# Your custom function
def percent_change(series):
    # Collect all *but* the last value of this window, then the final value
    previous_values = series[:-1]
    last_value = series[-1]

    # 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()
```



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.

In [ ]:
```python
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()
```

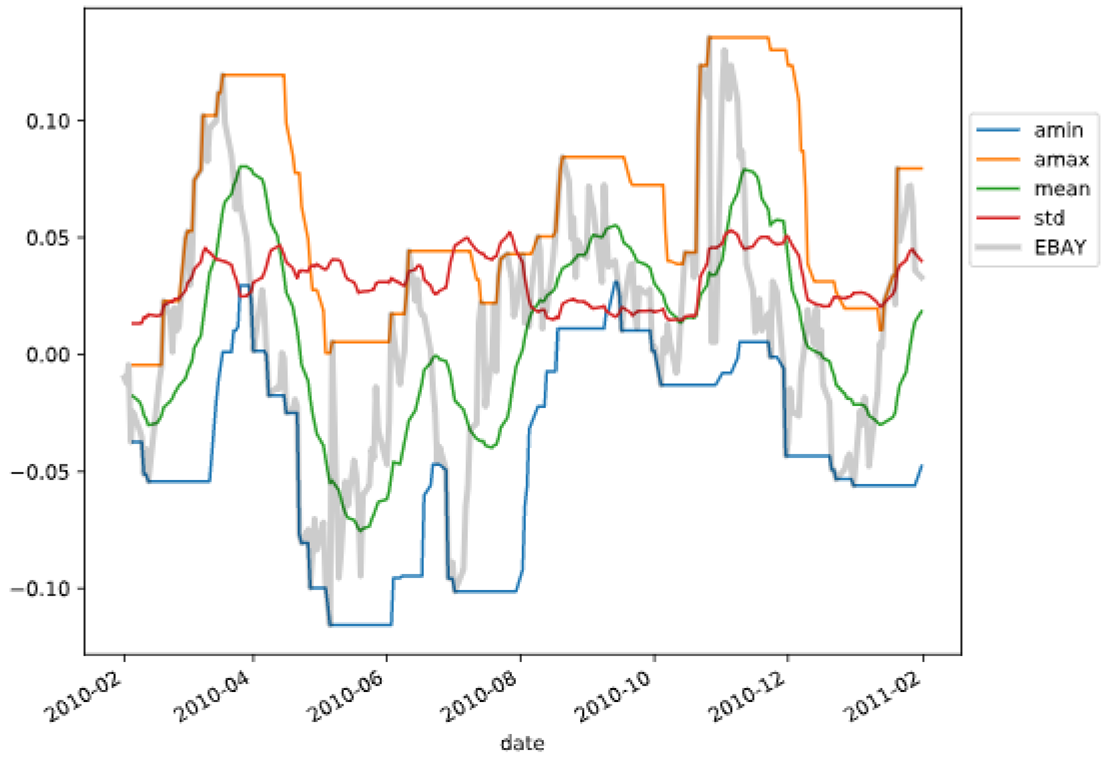


Since you've converted the data to % change over time, it was easier to spot and correct the outliers.

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.

In [ ]:
```python
# 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()
```



Comments:

- The features calculated above are similar to the other note for feature creation: defining a time window and aggregating on this window.

- For a specific date (x-axis), we have four features.

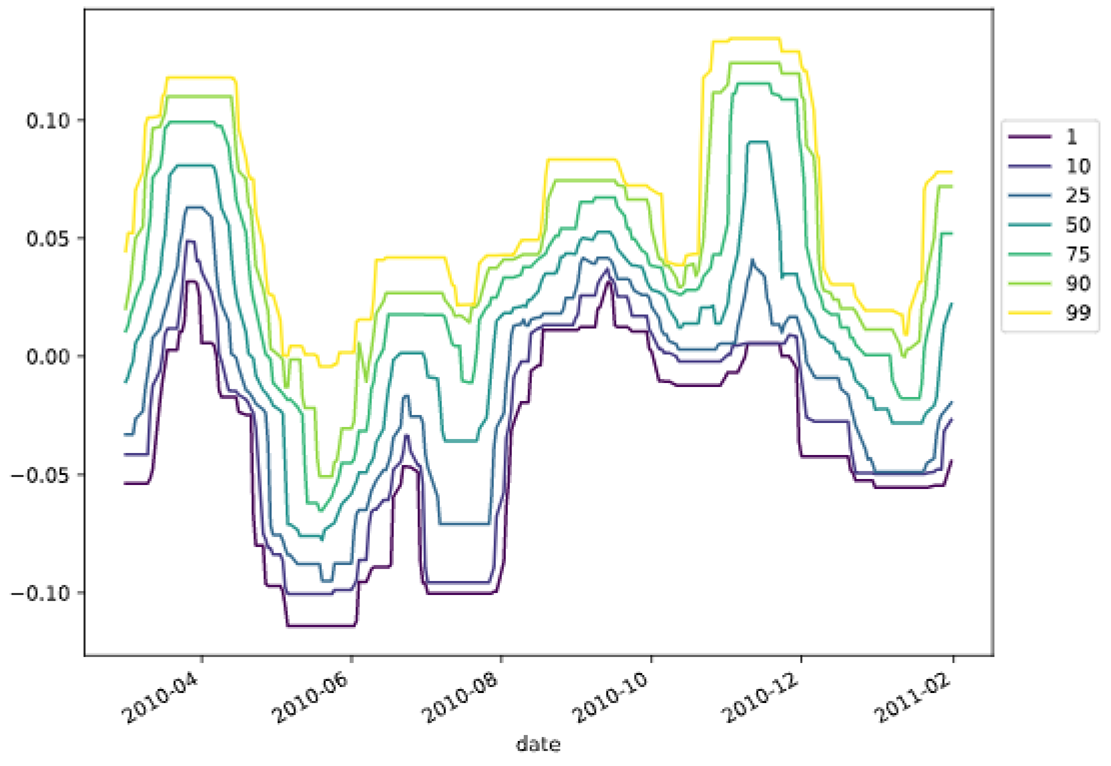
Percentiles and partial functions

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.

Side note: an example for partial.

In [3]:
```
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
cube = partial(power, exponent=3)

def test_partials():
    assert square(2) == 4
    assert cube(2) == 8
```

In [ ]:
```
# 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.

In [ ]:
```
# 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)
```

Comments
In other feature-target creation note, these features are proved not so important by a random-forest model, at least to the specific problem there.

Validating and Inspecting Time Series Models

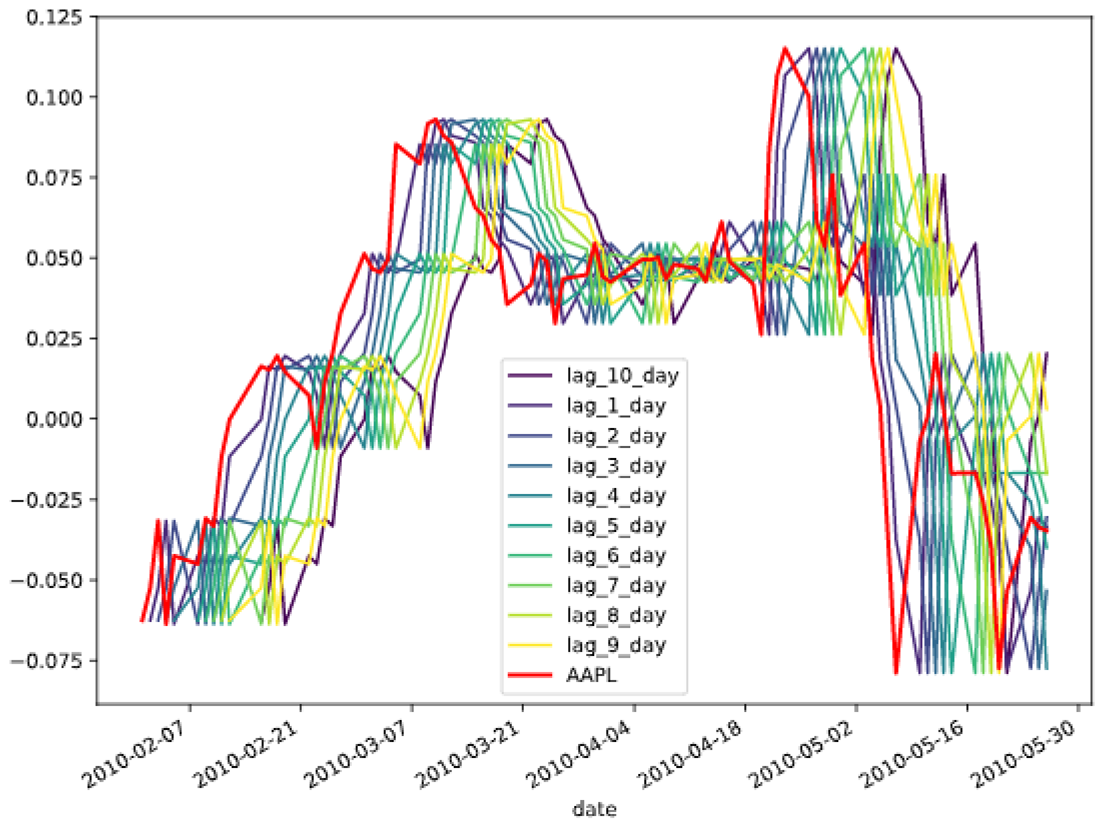
Generate predictions with models in order to validate them against "test" data.

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.

Here we "shift"the raw data and visualize the results. 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. **This should depend on specific situations. Sometimes een day-to-day fluctuation is still too long**.

In [ ]:
```
import numpy as np
# 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

With created time-shifted versions of a single time series, we 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. Using previous values of a timeseries to predict current values of the same timeseries is thus called **auto-regressive**. Similarly we have **autocorrelation**.

Machine learning and time series

- Auto-regressive should be the simplest machine learning model, just as linear regression. By creating features (here by shifting), We now connect the time series to the simplest model in machine learning.
- By creating more features, we can also connect time series to other machine learning models including linear and nonlinear. So for now, machine learning on time series is put within the general picture of other machine learning problems.
- There are many ways of feature generating, even for the same linear regression. For example, the regression approach I used before, and the way below.

- Using ACF can determine whether a stock is mean reverting or trending. Here Auto-regressive models should be able to predict continuous prices of a stock.

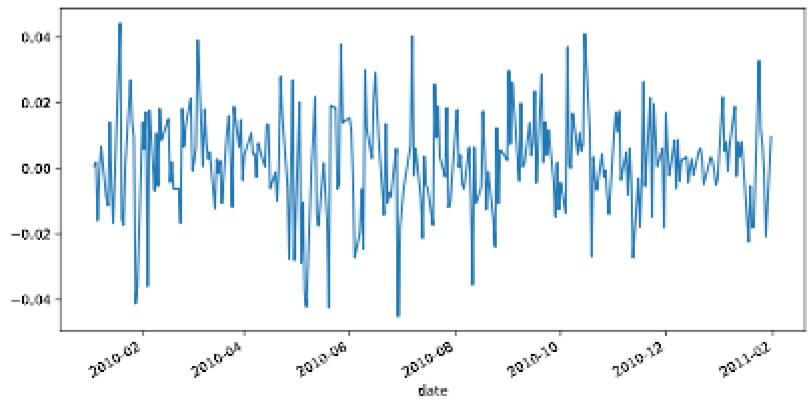
In [ ]:
```
# Replace missing values with the median for each column, and therefore the model will
# behaves well with scikit-learn
X = prices_perc_shifted.fillna(np.nanmedian(prices_perc_shifted))
y = prices_perc.fillna(np.nanmedian(prices_perc))

model = Ridge()
model.fit(X, y)
```

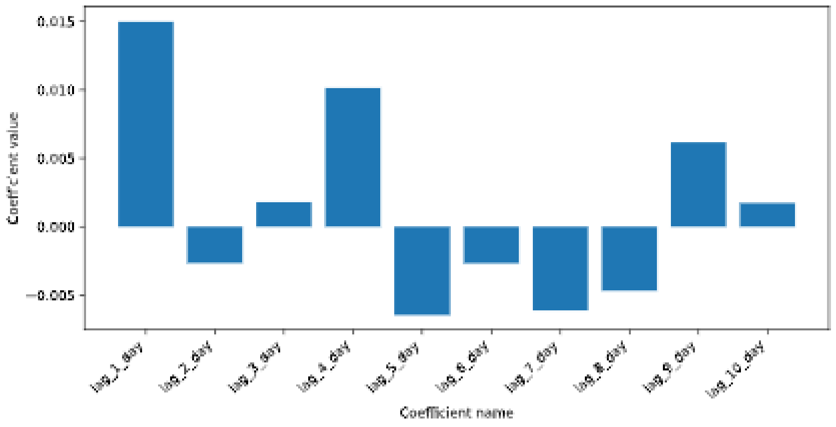
Visualize regression coefficients

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 the workspace.

In [ ]:
```
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()
```





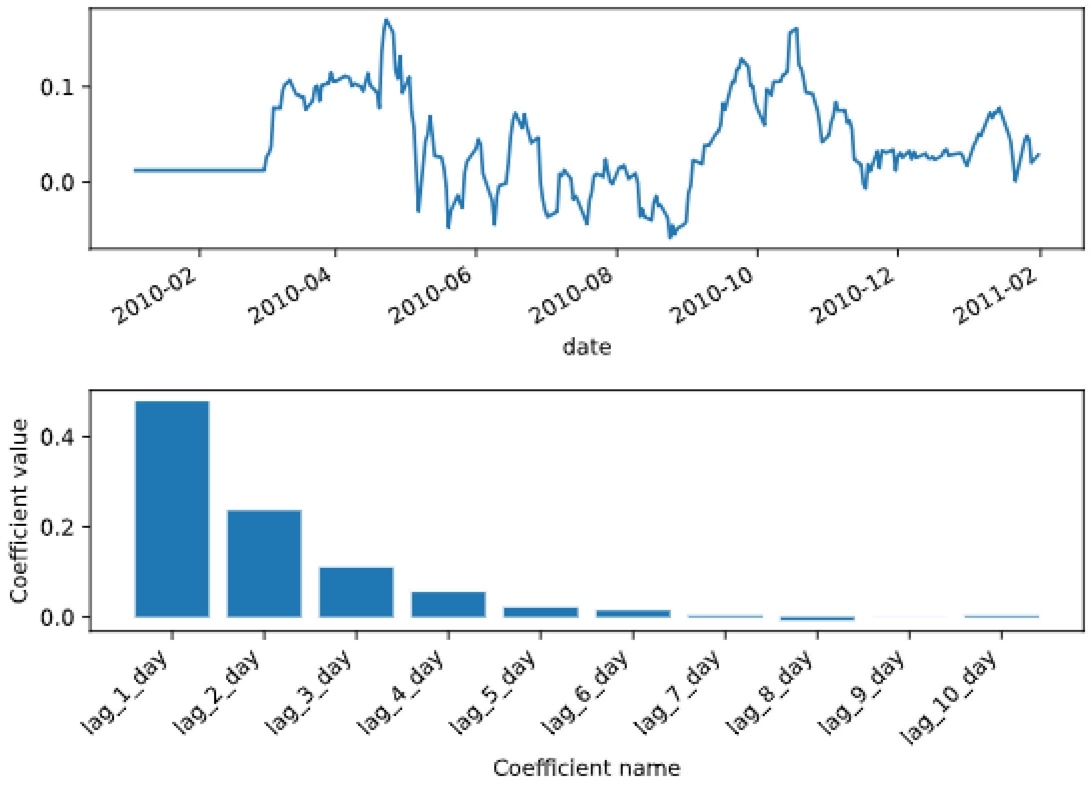
When using time-lagged features on the raw data, we see that the highest coefficient by far is the first one. This means that the N-1th time point is useful in predicting the Nth timepoint, but no other points are useful.

Auto-regression with a smoother time series

Now, let's re-run the same procedure using a smoother signal. 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.

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

In [ ]:
```python
# 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()
```



By transforming your data with a larger window, we'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.** How to understand this? **Is the above related to the rolling windown, where bigger window gives smoother data?** See earlier chapters for the size of windows.

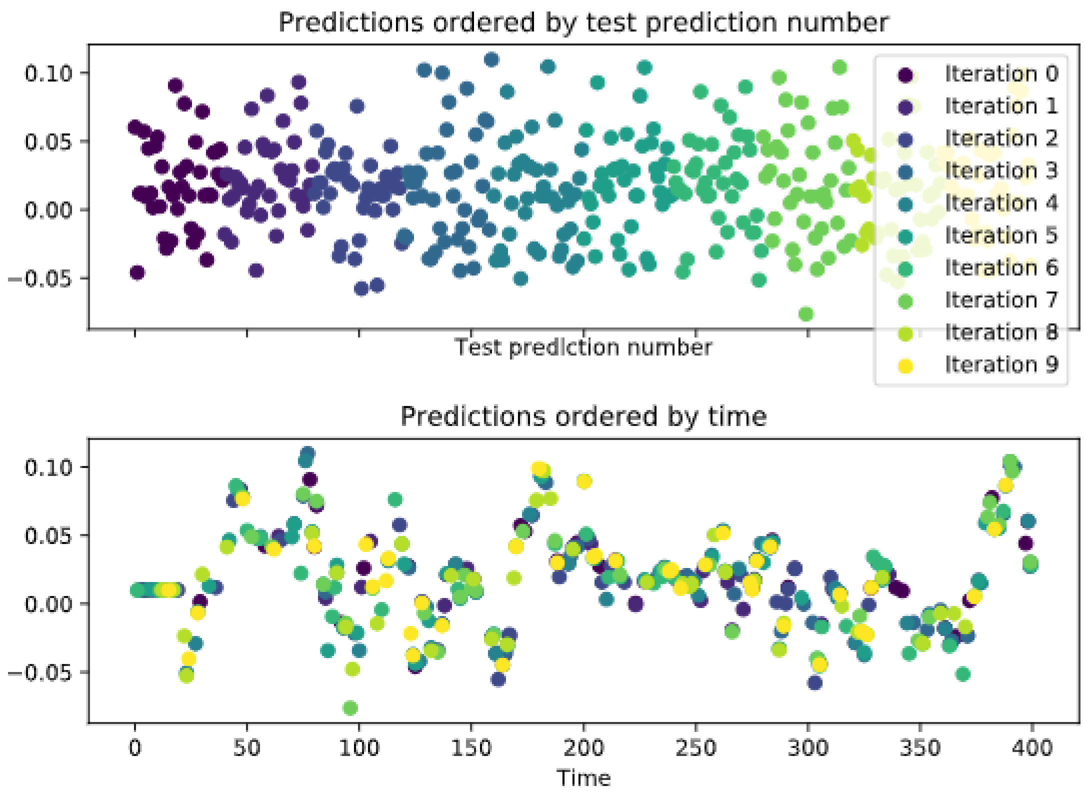
Cross-validation with shuffling

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.

```
In [ ]:
```

```python
# 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)
```



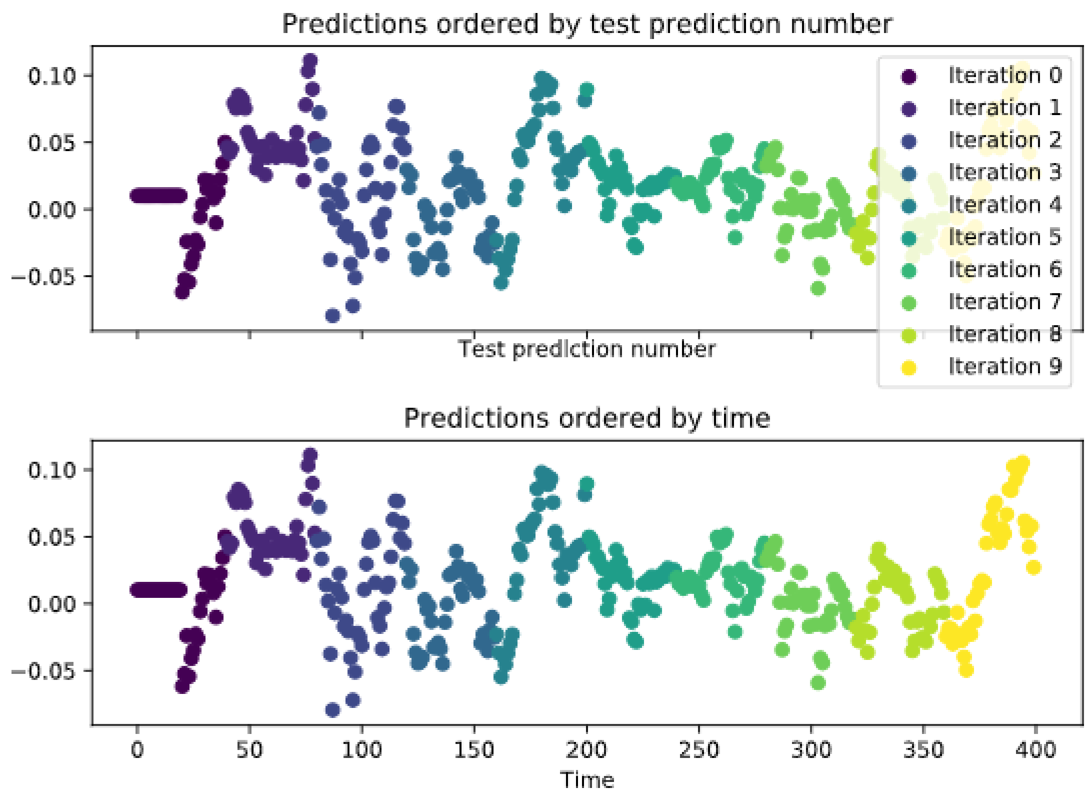
If you look at the plot to the right, see that the order of datapoints in the test set is scrambled. Let's see how it looks when we shuffle the data in blocks.

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.

```
In [ ]: # 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)
```



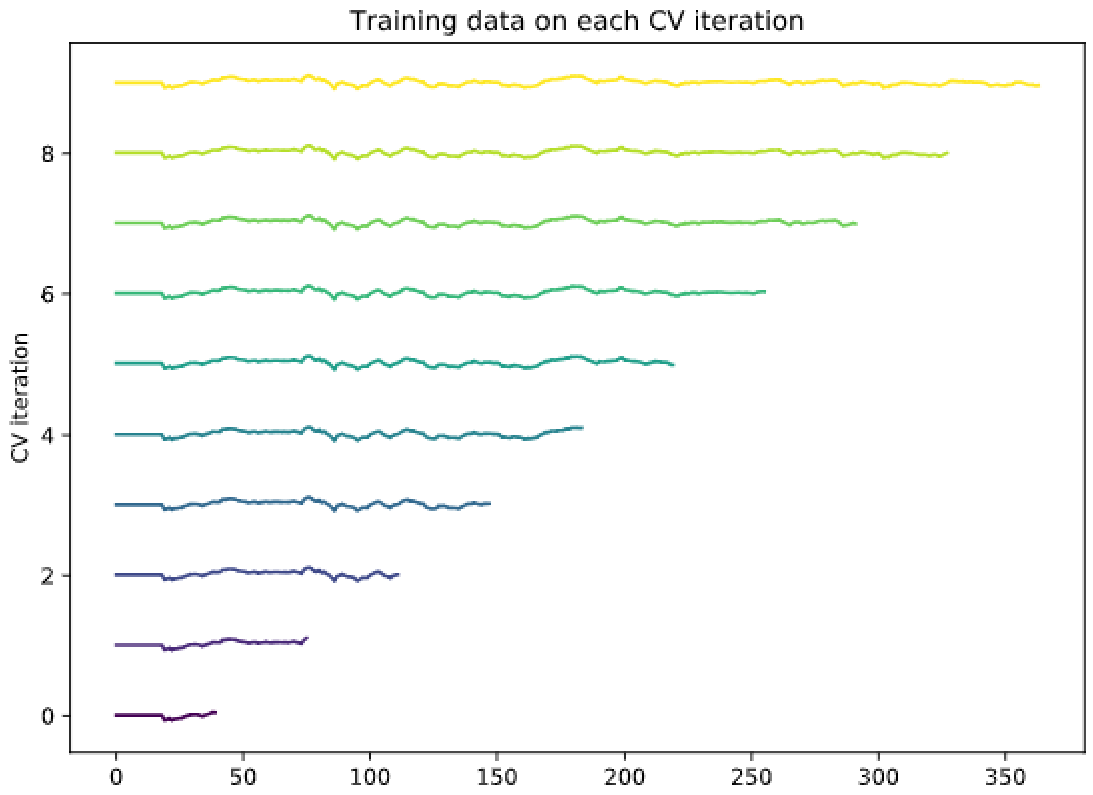
This time, the predictions generated within each CV loop look 'smoother' than they were before - they look more like a real time series because you didn't shuffle the data. This is a good sanity check to make sure your CV splits are correct.

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.

```
In [ ]: # 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.

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.

In [ ]:
```
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
```

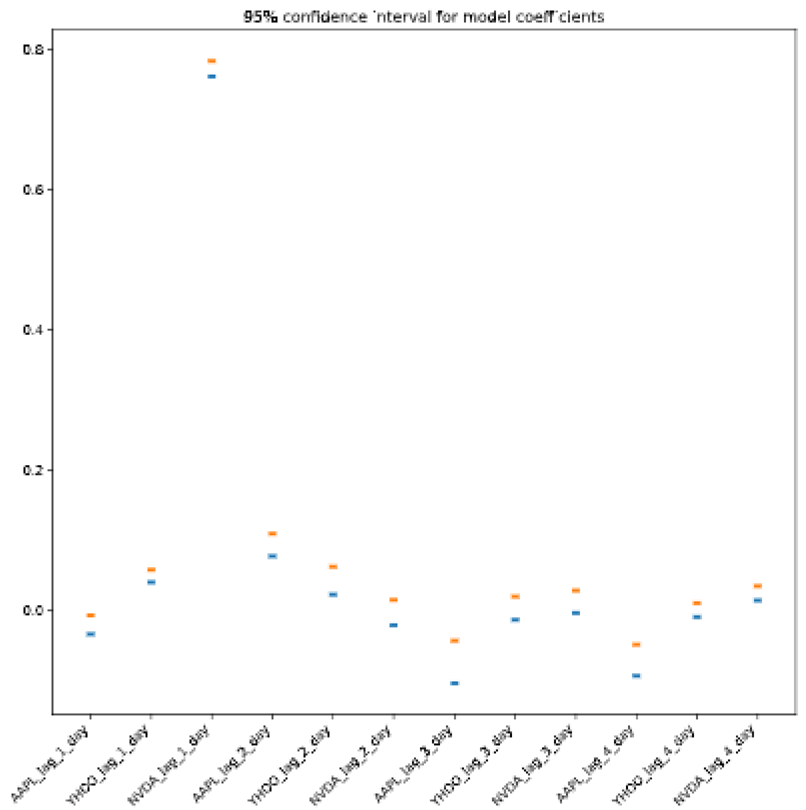
Calculating variability in model coefficients

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.

In [ ]:
```python
# 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()
```



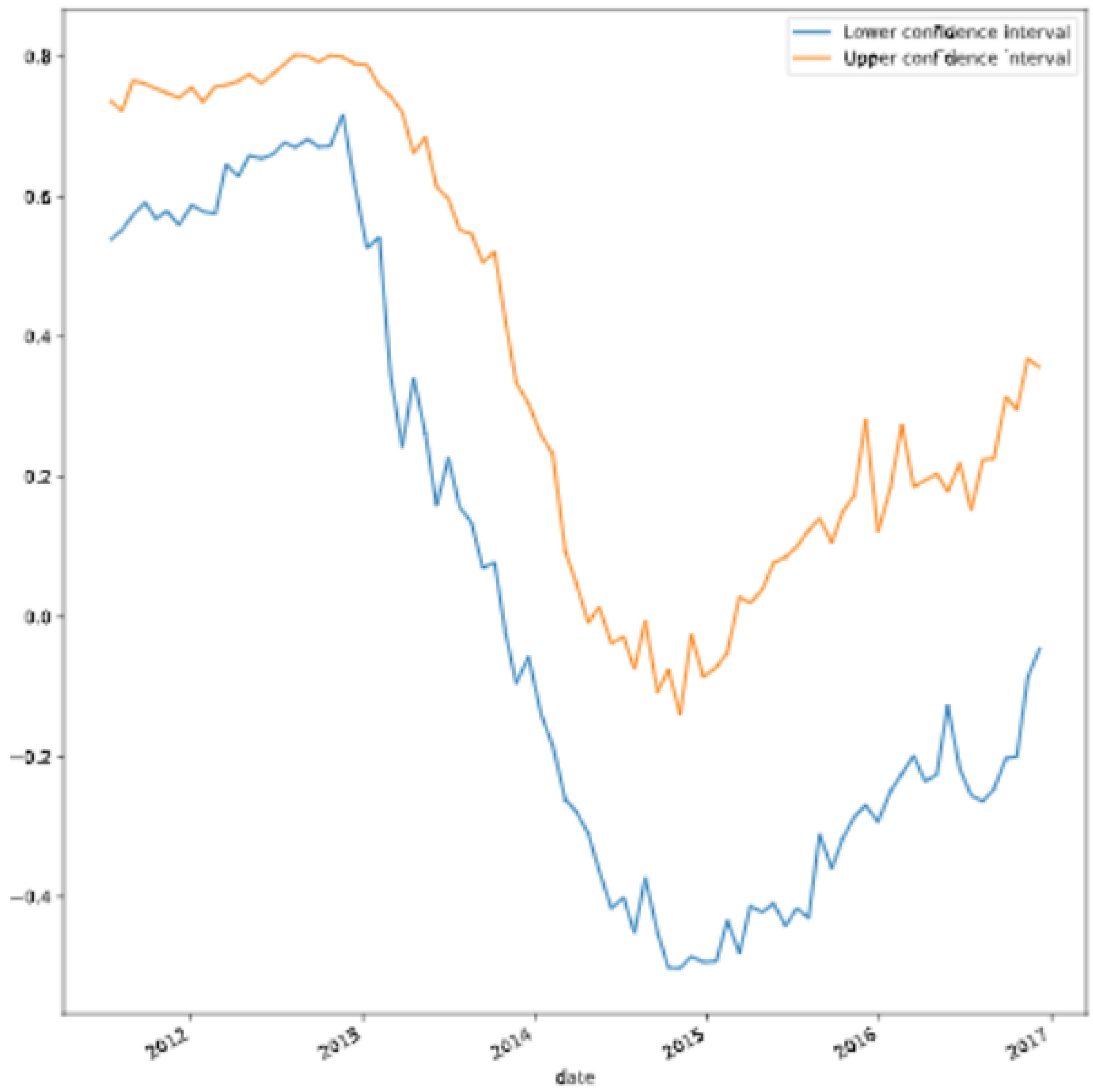
You've calculated the variability around each coefficient, which helps assess which coefficients are more stable over time!

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.

In [ ]:
```
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()
```



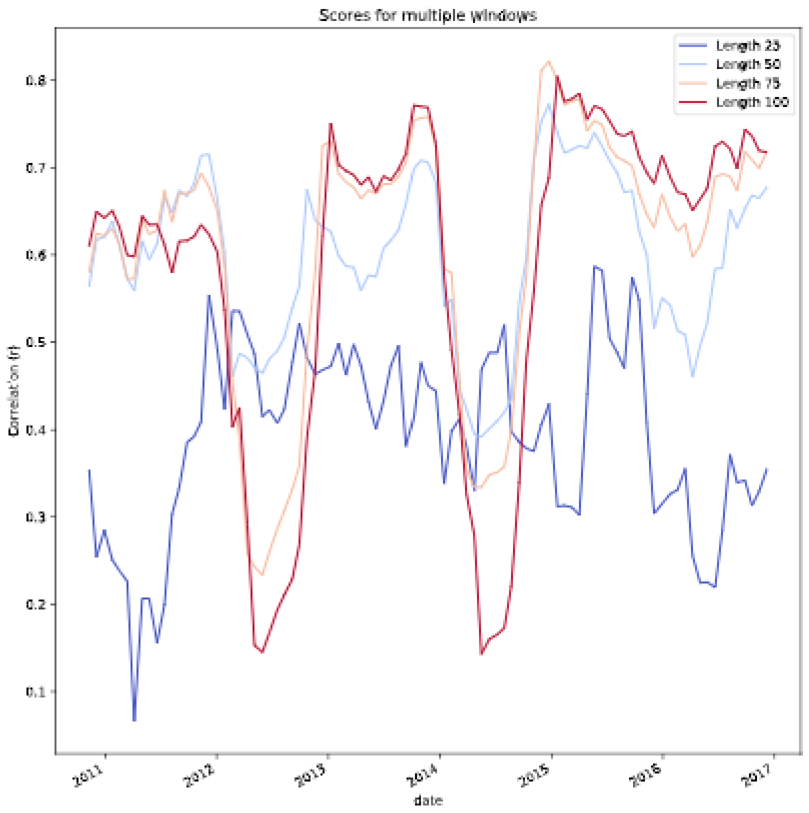
You plotted a rolling confidence interval for scores over time. This is useful in seeing when your model predictions are correct.

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.

In [ ]:
```
# 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()
```



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.

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.

Time series for deep learning in many other places.