**Machine Learning for Time Series in Python**

**1. Timeseries kinds and applications**

00:00 - 00:17

Welcome to Introduction to Machine Learning for Timeseries Data. This course is focused on the intersection of Machine Learning and Time series data, and hence we expect you have taken introductory courses on Machine learning and time series analysis here on DataCamp.

**2. Time Series**

00:17 - 00:35

This course focuses on machine learning in the context of timeseries data. Put simply, a timeseries means data that changes over time. This can take many different forms, such as atmospheric CO2 over time, the waveform of my voice as I am speaking.

**3. Time Series**

00:35 - 00:42

the fluctuation of a stock's value over the year, or demographic information about a city.

**4. What makes a time series?**

00:42 - 01:04

Timeseries data consists of at least two things: One, an array of numbers that represents the data itself. Two, another array that contains a timestamp for each datapoint. The timestamps can include a wide range of time data, from months of the year to nanoseconds.

**5. Reading in a time series with Pandas**

01:04 - 01:18

Here we import timeseries data into a pandas DataFrame. Note that each datapoint has a corresponding time point (in this case, a date), though multiple datapoints may have the same time point.

**6. Plotting a pandas timeseries**

01:18 - 01:32

Here is the code to plot this timeseries data with Matplotlib and Pandas. We first create a figure and axis, then read in the data with Pandas and use the dot-plot method to plot the data on the axis.

**7. A timeseries plot**

01:32 - 01:46

The amount of time that passes between timestamps defines the "period" of the timeseries. In this case, it is about one day. This often helps us infer what kind of timeseries we're dealing with.

**8. Why machine learning?**

01:46 - 02:17

Machine learning has taken the world of data science by storm. In the last few decades, advances in computing power, algorithms, and community practices have made it possible to use computers to ask questions that were never thought possible. Machine learning is about finding patterns in data - often patterns that are not immediately obvious to the human eye. This is often because the data is either too large or too complex to be processed by a human.

**9. Why machine learning?**

02:17 - 02:34

Another crucial part of machine learning is that we can build a model of the world that formalizes our knowledge of the problem at hand. We can use this model to make predictions. Combined with automation, this can be a critical component of an organization's decision making.

**10. Why combine these two?**

02:34 - 03:07

Why should we treat timeseries any differently from another data set? Well, machine learning is all about finding patterns in data. Timeseries data always change over time, which turns out to be a useful pattern to utilize. For example, here is a raw waveform of someone speaking, and here is a collection of timeseries features that were extracted from it. As you can see, using timeseries-specific features lets us see a much richer representation of the raw data.

**11. A machine learning pipeline**

03:07 - 03:41

This course will focus on a simple machine learning pipeline in the context of timeseries data. This boils down to the following main steps. Feature extraction: what kinds of special features leverage a signal that changes over time? Model fitting: what kinds of models are suitable for asking questions with timeseries data? Validation: How can we validate a model that uses timeseries data? What considerations must we make because it changes in time?

**12. Let's practice!**

03:41 - 03:45

Let's start by plotting some time series data.

**Timeseries kinds and applications \_\_\_**

* data that changes over time
  + e.g., atmospheric changes, demographic information, financial data, voice wave forms
  + datapoints and timestamps for each data point
* in machine learning, changes over time shows useful patterns in machine learning
* a machine learning pipeline
  + feature extraction
  + model fitting
  + prediction and validation

**#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())

#################################################

#<script.py> output:

# symbol data\_values

# 0 214.009998

# 1 214.379993

# 2 210.969995

# 3 210.580000

# 4 211.980005

#################################################

# Print the first 5 rows of data2

#print(data2.head())

#################################################

#<script.py> output:

# data\_values

# 0 -0.006928

# 1 -0.007929

# 2 -0.008900

# 3 -0.009815

# 4 -0.010653

#################################################

# 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 begin by looking at your data
* scikit-learn data needs to be 2 dimensional
  + (samples, features)

#**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 below.

#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())

#################################################

#<script.py> output:

# sepal length (cm) sepal width (cm) petal length (cm) petal width (cm) \

# 50 7.0 3.2 4.7 1.4

# 51 6.4 3.2 4.5 1.5

# 52 6.9 3.1 4.9 1.5

# 53 5.5 2.3 4.0 1.3

# 54 6.5 2.8 4.6 1.5

#

# target

# 50 1

# 51 1

# 52 1

# 53 1

# 54 1

#################################################

#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()

#################################################

#<script.py> output:

# [2 2 2 1 1 2 2 2 2 1 2 1 1 2 1 1 2 1 2 2]

#################################################

#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** \_\_\_

* using audio data of heart sounds to detect who has a heart condition
* using new york stock exchange data to detect patterns in historical records that allow us to predict the value of companies in the future

#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. https://www.kaggle.com/kinguistics/heartbeat-sounds

#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.

#https://www.kaggle.com/dgawlik/nyse

#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()

#################################################

#<script.py> output:

# AAPL FB NFLX V XOM

# time

# 2010-01-04 214.009998 NaN 53.479999 88.139999 69.150002

# 2010-01-05 214.379993 NaN 51.510001 87.129997 69.419998

# 2010-01-06 210.969995 NaN 53.319999 85.959999 70.019997

# 2010-01-07 210.580000 NaN 52.400001 86.760002 69.800003

# 2010-01-08 211.980005 NaN 53.300002 87.000000 69.519997

#################################################

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?

**Classifying a time series** \_\_\_

* always visualize raw data before fitting models
* start with summary statistics

#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** \_\_\_

* The auditory envelope
  + smooth the data to calculate the auditory envelope
  + related to the amount of audio energy present at each moment in time
* smoothing over time
  + instead of averaging over time, we do a local average
  + this is called smoothing your timeseries
  + it removes short-term noise, while retaining the general pattern

#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()

# Rectify the audio signal

#audio\_rectified = audio.apply(np.abs)

# Plot the result

#audio\_rectified.plot(figsize=(10, 5))

#plt.show()

# 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** \_\_\_

* fourier transform
  + timeseries data can be described as a combination of quickly-changing and slowly-changing things
  + at each moment in time, we can describe the relative presence of fast- and slow-moving components
  + this converts a single timeseries into an array that describes the timeseries as a combination of oscillations
* short time (st) fft is squared = spectrogram
* spectral feature engineering
  + each timeseries has a different spectral pattern
  + we can calculate these spectral patterns by analyzing the spectrogram to describe where most of the energy is at each moment in time
    - **spectral bandwidth**
    - **spectral centroids**

#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 functions you'll use for the STFT

#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)

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 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))

#################################################

#<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.

**Predicting data over time** \_\_\_

* correlation between variables often changes over time
  + two timeseries that seem correlated at one moment may not remain so over time
* Coefficient of Determination (R squared)
  + 1 - (error of the model / variance of test data
  + "model accounts for % of variance of the model"

#Introducing the dataset

#As mentioned in the video, you'll 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)

# 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** \_\_\_

* cleaning messy data
  + interpolating missing values
  + transforming data to standardize variance
  + finding outliers (> 3 sd) in data
    - replace outliers with threeshold/median

#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)

#################################################

#<script.py> output:

# symbol

# EBAY 273

# NVDA 502

# YHOO 232

# dtype: int64

#################################################

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.

# 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** \_\_\_

* using .aggregate for feature extraction
* using .partial() functions in Python
* percentiles summarize your data
  + percentiles are a useful way to get more fine-grained summaries of your data (as opposed to using np.mean)
* calculating "date-based" features
  + using Pandas .index

#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()

In the next exercise, you will calculate the percentiles.

#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()

In the next exercise, you will extract the date components of the timestamps.

#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

#

# ... ... ... ... ...

#

# [504 rows x 4 columns]

#################################################

#This concludes the third chapter. In the next chapter, you will

#learn advanced techniques to validate and inspect your time series

#models.

**Creating features from the past** \_\_\_

* The past is useful
  + timeseries data almost always have information that is shared between timepoints
  + information in the past can help predict what happens in the future
  + often the features best suited to predict a timeseries are previous values of the same timeseries
* A note on smoothness and auto-correlation
  + a common question to ask of a timeseries: how smooth is the data?
  + or, how correlated is a timepoint with its neighboring timepoints (called **autocorrelation**)
  + the amount of auto-correlation in data will impact your models

#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. Now let's take a look at what your

#model found.

#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. (independent and identically distributed)

**Cross-validating time series data** \_\_\_

* K-fold cross-validation is usually used in timeseries data
* shuffling data in cross validation only works for data that is i.i.d.
  + do not use when making predictions with a timeseries
* using the time series CV iterator
  + always use data from the **past** *(training data)* to predict the **future** *(test data)*

#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 above, 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.

# 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
  + stationary timeseries do not change their statistical properties over time -e.g., mean, standard deviation, trends
  + most timeseries are non-stationary to some extent
* stability
  + non-stationary data results in variability in our model
  + the statistical properties the model finds may change with the data
  + we will be less certain about the correct values of model parameters
* cross-validation to quantify parameter stability
  + calculate model parameters on each iteration
  + assess parameter stability across all CV splits
* bootstrapping the mean
  + a common way to assess variability
  + take a random sample of data **with replacement**
  + calculate the mean of the sample
  + repeat this process thousands of times
  + calculate the percentiles of the result (usually 2.5, 97.5)
  + the result is a *95% confidence interval* of the mean of each coefficient
* assessing model performance stability
  + if using TimeSeriesSplit, we can *plot* the model's score over time
  + this is useful in finding certain regions of time that hurt the score
  + also useful to find non-stationary signals

#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

#bootstrap 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 workspace. 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()

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.

**Wrap-up** \_\_\_

* Timeseries and machine learning
  + the many applications of time series + machine learning
  + always visualize the data first
  + the scikit-learn API standardizes this process
* Feature extraction and classification
  + summary statistics for time series classification
  + combining multiple features into a single input matrix
  + feature extraction for time series data
* Model fitting and improving data quality
  + time series features for regression
  + generating predictions over time
  + cleaning and improving time series data
* Validating and assessing our model performance
  + cross-validation with time series data (don't shuffle the data!)
  + time series stationarity
  + assessing model coefficient and score stability
* Advanced concepts in time series
  + advanced window functions
  + signal processing and filtering details
  + spectral analysis
* Advanced machine learning
  + advanced time series feature extraction
    - e.g., tsfresh
  + more complex model architectures for regression and classification
  + production-ready pipelines for time series analysis
* Ways to practice
  + Kaggle
  + Quantopian

# 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.**

**Yes! You don't have timestamps for each data point, so it is not a time series.**

## Exercise

# 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](https://www.datacamp.com/courses/introduction-to-time-series-analysis-in-python) and [Visualizing Time Series Data with Python](https://www.datacamp.com/courses/visualizing-time-series-data-in-python) courses.

## Instructions 1/3

Print the first five rows of data.

# Print the first 5 rows of data

print(\_\_\_\_)

# Print the first 5 rows of data

print(data.head())

symbol data\_values

0 214.01

1 214.38

2 210.97

3 210.58

4 211.98

# Print the first 5 rows of data

print(data.head())

Print the first five rows of data2.

# Print the first 5 rows of data2

print(\_\_\_\_)

**# Print the first 5 rows of data2**

**print(data2.head())**

**data\_values**

**0 -0.007**

**1 -0.008**

**2 -0.009**

**3 -0.010**

**4 -0.011**

# Print the first 5 rows of data2

print(data2.head())

Plot the values column of both the data sets on top of one another, one per axis object.

# Plot the time series in each dataset

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

data.iloc[:1000].\_\_\_\_(y=\_\_\_\_, ax=axs[0])

data2.iloc[:1000].\_\_\_\_(y=\_\_\_\_, ax=axs[1])

plt.show()

**Plotting a Timeseries(I)**

# 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()

# 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()

**Good job! What kind of data do you think each plot represents?**

## Exercise

# 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.

## Instructions

100 XP

* Plot data and data2 on top of one another, one per axis object.
* The x-axis should represent the time stamps and the y-axis should represent the dataset values.

# Plot the time series in each dataset

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

data.iloc[:1000].plot(x=\_\_\_\_, y=\_\_\_\_, ax=axs[0])

data2.iloc[:1000].plot(x=\_\_\_\_, y=\_\_\_\_, ax=axs[1])

plt.show()

# 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()

# 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()

**Correct! 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.**

**1. Machine learning basics**

00:00 - 00:15

Now we'll cover the basics of Machine Learning. This should be a recap of material that you've already covered in previous DataCamp courses. We'll start with the basics of how to fit and predict a model using scikit-learn.

**2. Always begin by looking at your data**

00:15 - 00:32

Before performing any data analysis, you should always take a look at your raw data. This gives you a quick high-level take on the quality/kind of your data. In Numpy, you can do so by printing out the first few rows of the data.

**3. Always begin by looking at your data**

00:32 - 00:41

In Pandas, this can be done by using the dot-head method, which shows the first five rows and all columns by default.

**4. Always visualize your data**

00:41 - 01:05

It is also crucial to visualize your data. The proper visualization will depend on the kind of data you've got, though histograms and scatterplots are a good place to start. Look at the distribution of your data. Does it seem reasonable? Are there any outliers? Are you missing data? Each of these questions is important to answer before doing any analysis.

**5. Scikit-learn**

01:05 - 01:26

Once you've gotten to know your data, it's time to start modeling it. The most popular library for machine learning in Python is called "scikit-learn". It has a standardized API so that you can fit many different models with a similar code structure. Here, we import Support Vector Machine to classify datapoints.

**6. Preparing data for scikit-learn**

01:26 - 01:55

scikit-learn expects data to have a particular shape. Before using scikit-learn, your data should be two-dimensional. The first axis should correspond to sample number, and the second should correspond to feature number. This pattern is used in almost all scikit-learn functions. If your data is not in this shape, there are a few options for reshaping it so that you can use it with scikit-learn.

**7. If your data is not shaped properly**

01:55 - 02:06

The most common approach is to "transpose" your data. This will swap the first and last axis. This is most useful when your data is two-dimensional.

**8. If your data is not shaped properly**

02:06 - 02:14

Another option is to use the dot-reshape method, which lets you specify the shape you want.

**9. Fitting a model with scikit-learn**

02:14 - 02:39

Now that your data has the correct shape, it's time to fit a model. First we must create an instance of the model we've imported (in this case, a support-vector classifier). You can call the method dot-fit on this instance to train the model. Here we show how you can input X (training data) and y (labels for each datapoint) to fit the model.

**10. Investigating the model**

02:39 - 02:55

It is often useful to investigate what kind of pattern the model has found. Most models will store this information in attributes that are created after calling dot-fit. Here we show the coefficients the model has given to each feature.

**11. Predicting with a fit model**

02:55 - 03:02

Once your model is fit, you can call the dot-predict method on the model to determine labels for unseen datapoints.

**12. Let's practice**

03:02 - 03:10

Now that we've practiced loading and preparing data, and fitting the model, it's time to put this into practice.

## Exercise

# 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](https://www.datacamp.com/courses/supervised-learning-with-scikit-learn) and [Preprocessing for Machine Learning in Python](https://www.datacamp.com/courses/preprocessing-for-machine-learning-in-python) courses.

## Instructions 1/2

Print the first five rows of data.

# Print the first 5 rows for inspection

print(\_\_\_\_)

**# Print the first 5 rows for inspection**

**print(data.head())**

**sepal length (cm) sepal width (cm) petal length (cm) petal width (cm) target**

**50 7.0 3.2 4.7 1.4 1**

**51 6.4 3.2 4.5 1.5 1**

**52 6.9 3.1 4.9 1.5 1**

**53 5.5 2.3 4.0 1.3 1**

**54 6.5 2.8 4.6 1.5 1**

* Extract the "petal length (cm)" and "petal width (cm)" columns of data and assign it to X.
* Fit a model on X and y.
* Extract the "petal length (cm)" and "petal width (cm)" columns of data and assign it to X.
* Fit a model on X and y.

from sklearn.svm import LinearSVC

# Construct data for the model

X = \_\_\_\_

y = data[['target']]

# Fit the model

model = LinearSVC()

\_\_\_\_(X, y)

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)

LinearSVC()

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)

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

 /

  /

1. 

## Exercise

# 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.

## Instructions

100 XP

* Predict the flower type using the array X\_predict.
* Run the given code to visualize the predictions.

# Create input array

X\_predict = targets[['petal length (cm)', 'petal width (cm)']]

# Predict with the model

predictions = \_\_\_\_

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()

**# 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()**

**[2 2 2 1 1 2 2 2 2 1 2 1 1 2 1 1 2 1 2 2]**

# 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()

Good job! Note that the output of your predictions are all integers, representing that datapoint's predicted class.

## Exercise

# Fitting a simple model: regression

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

* "MedHouseVal": the median house value for California districts (in $100,000s of dollars)
* "AveRooms" : average number of rooms per dwelling

## Instructions

* Prepare X and y DataFrames using the data in housing.
  + X should be the Median House Value, y average number of rooms per dwelling.
* Fit a regression model that uses these variables (remember to shape the variables correctly!).
* Don't forget that each variable must be the correct shape for scikit-learn to use it!

from sklearn import linear\_model

# Prepare input and output DataFrames

X = \_\_\_\_

y = \_\_\_\_

# Fit the model

model = linear\_model.LinearRegression()

model.fit(\_\_\_\_)

from sklearn import linear\_model

# Prepare input and output DataFrames

X = housing[['MedHouseVal']]

y = housing[['AveRooms']]

# Fit the model

model = linear\_model.LinearRegression()

model.fit(X, y)

LinearRegression()

from sklearn import linear\_model

# Prepare input and output DataFrames

X = housing[['MedHouseVal']]

y = housing[['AveRooms']]

# Fit the model

model = linear\_model.LinearRegression()

model.fit(X, y)

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

## Exercise

# Predicting using a regression model

Now that you've fit a model with the California 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 "MedHouseVal" (median house value) is available in your workspace along with the model you fit in the previous exercise.

## Instructions

* Review new\_inputs in the shell.
* Reshape new\_inputs appropriately to generate predictions.
* Run the given code to visualize the predictions.

# Generate predictions with the model using those inputs

predictions = \_\_\_\_

# Visualize the inputs and predicted values

plt.scatter(new\_inputs, predictions, color='r', s=3)

plt.xlabel('inputs')

plt.ylabel('predictio

ns')

plt.show()

**# Generate predictions with the model using those inputs**

**predictions =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()**

**# 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()**

# 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()

**Good job! Here the red line shows the relationship that your model found. As the number of rooms grows, the median house value rises linearly.**

**1. Combining timeseries data with machine learning**

00:00 - 00:10

In the final lesson of this chapter, we'll discuss the interaction between machine learning and timeseries data, and introduce why they're worth thinking about in tandem.

**2. Getting to know our data**

00:10 - 00:22

First, let's give a quick overview of the data we'll be using. They're both freely available online, and come from the excellent website Kaggle-dot-com.

**3. The Heartbeat Acoustic Data**

00:22 - 00:48

Audio is a very common kind of timeseries data. Audio tends to have a very high sampling frequency (often above 20,000 samples per second!). Our first dataset is audio data recorded from the hearts of medical patients. A subset of these patients have heart abnormalities. Can we use only this heartbeat data to detect which subjects have abnormalities?

**4. Loading auditory data**

00:48 - 01:07

Audio data is often stored in "wav" files. We can list all of these files using the "glob" function. It lists files that match a given pattern. Each of these files contains the auditory data for one heartbeat session, as well as the sampling rate for that data.

**5. Reading in auditory data**

01:07 - 01:42

We'll use a library called "librosa" to read in the audio dataset. Librosa has functions for extracting features, visualizations, and analysis for auditory data. We can import the data using the "load" function. The data is stored in audio and the sampling frequency is stored in sfreq. Note that the sampling frequency here is 2205, which means 2205 samples are recorded per second.

**6. Inferring time from samples**

01:42 - 01:51

Using only the sampling frequency, we can infer the timepoint of each datapoint in our audio file, relative to the start of the file.

**7. Creating a time array (I)**

01:51 - 02:12

Now we'll create an array of timestamps for our data. To do so, you have two options. The first is to generate a range of indices from zero to the number of datapoints in your audio file, divide each index by the sampling frequency, and you have a timepoint for each data point.

**8. Creating a time array (II)**

02:12 - 02:32

The second option is to calculate the final timepoint of your audio data using a similar method. Then, use the linspace function to generate evenly-spaced numbers between 0 and the final timepoint. In either case, you should have an array of numbers of the same length as your audio data.

**9. The New York Stock Exchange dataset**

02:32 - 03:01

Next, we'll explore data from the New York Stock Exchange. It runs over a much longer timespan than our audio data, and has a sampling frequency on the order of one sample per day (compared with 2,205 samples per second with the audio data). Our goal is to predict the stock value of a company using historical data from the market. As we are predicting a continuous output value, this is a regression problem.

**10. Looking at the data**

03:01 - 03:11

Let's take a look at the raw data. Each row is a sample for a given day and company. It seems that the dates go back all the way to 2010.

**11. Timeseries with Pandas DataFrames**

03:11 - 03:33

It is useful to investigate the "type" of data in each column. Numpy or Pandas may treat an array of data in special ways depending on its type. We can print the type of each column by looking at the dot-dtypes attribute. Here we see that the type of each column is "object", which is a generic data type.

**12. Converting a column to a time series**

03:33 - 03:47

Since we know one column is actually a list of dates, let's change the column type to "datetime" using the to\_datetime function. This will help us perform visualization and analysis later on.

**13. Let's practice!**

03:47 - 03:58

Now let's get our hands dirty with these two datasets. We'll practice loading in and exploring the data, then move on to some simple analysis.

## Exercise

# 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](https://www.kaggle.com/kinguistics/heartbeat-sounds).

## Instructions

100 XP

* Use glob to return a list of the .wav files in data\_dir directory.
* Import the first audio file in the list using librosa.
* Generate a time array for the data.
* Plot the waveform for this file, along with the time array.

mport librosa as lr

from glob import glob

# List all the wav files in the folder

audio\_files = \_\_\_\_(data\_dir + '/\*.wav')

# Read in the first audio file, create the time array

audio, sfreq = lr.load(\_\_\_\_)

time = np.arange(0, len(audio)) / \_\_\_\_

# Plot audio over time

fig, ax = plt.subplots()

ax.plot(\_\_\_\_, \_\_\_\_)

ax.set(xlabel='Time (s)', ylabel='Sound Amplitude')

plt.show()

Check your call of lr.load(). Did you correctly specify the first argument? Expected './files/murmur\_\_201108222238.wav', but got '/\*.wav'.

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

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()

**Good job! 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.**

**Exercise**

# 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](https://www.kaggle.com/dgawlik/nyse).

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.

## Instructions

100 XP

* Import the data with Pandas (stored in the file 'prices.csv').
* Convert the index of data to datetime.
* Loop through each column of data and plot the the column's values over time.

# Read in the data

data = pd.\_\_\_\_('prices.csv', index\_col=0)

# Convert the index of the DataFrame to datetime

data.index = \_\_\_\_(data.index)

print(data.head())

# Loop through each column, plot its values over time

fig, ax = plt.subplots()

for column in \_\_\_\_:

    data[column].plot(ax=ax, label=column)

ax.legend()

plt.show()

# 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:

data[column].plot(ax=ax, label=column)

ax.legend()

plt.show()

AAPL FB NFLX V XOM

time

2010-01-04 214.01 NaN 53.48 88.14 69.15

2010-01-05 214.38 NaN 51.51 87.13 69.42

2010-01-06 210.97 NaN 53.32 85.96 70.02

2010-01-07 210.58 NaN 52.40 86.76 69.80

2010-01-08 211.98 NaN 53.30 87.00 69.52

# 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:

    data[column].plot(ax=ax, label=column)

ax.legend()

plt.show()

**LECTURE**

**1. Classification and feature engineering**

00:00 - 00:12

We'll now discuss one of the most common categories of machine learning problems: classification. We'll also discuss the concept of feature engineering in the context of time series data.

**2. Always visualize raw data before fitting models**

00:12 - 00:26

Before we begin, let's take a moment to once again visualize the data we're dealing with. There is a lot of complexity in any machine learning step, and visualizing your raw data is important to make sure you know where to begin.

**3. Visualize your timeseries data!**

00:26 - 00:50

To plot raw audio, we need two things: the raw audio waveform, usually in a 1- or 2-dimensional array. We also need the timepoint of each sample. We can calculate the time by dividing the index of each sample by the sampling frequency of the timeseries. This gives us the time for each sample relative to the beginning of the audio.

**4. What features to use?**

00:50 - 01:08

As we saw in the introduction, using raw data as input to a classifier is usually too noisy to be useful. An easy first step is to calculate summary statistics of our data, which removes the "time" dimension and give us a more traditional classification dataset.

**5. Summarizing timeseries with features**

01:08 - 01:28

Here we see a description of this process. For each timeseries, we calculate several summary statistics. These then can be used as features for a model. We have expanded a single feature (raw audio amplitude) to several features (here, the min, max, and average of each sample).

**6. Calculating multiple features**

01:28 - 01:44

Here we show how to calculate multiple features for a several timeseries. By using the "axis equals -1" keyword, we collapse across the last dimension, which is time. The result is an array of numbers, one per timeseries.

**7. Fitting a classifier with scikit-learn**

01:44 - 02:03

In the last step, we collapsed a two-dimensional array into a one-dimensional array for each feature of interest. We can then combine these as inputs to a model. In the case of classification, we also need a label for each timeseries that allows us to build a classifier.

**8. Preparing your features for scikit-learn**

02:03 - 02:33

In order to prepare your data for scikit-learn, remember to ensure that it has the correct shape, which is samples by features. Here we can use the column\_stack function, which lets us stack arrays by turning them into the columns of a two-dimensional array. In addition, the labels array is 1-dimensional, so we reshape it so that it is two dimensions. Finally, we fit our model to these arrays, X and y.

**9. Scoring your scikit-learn model**

02:33 - 03:13

Now that we've fit our model, we'll score the classifier. There are many ways that we can score a classifier with scikit-learn. First, we show how to generate predictions with a model that has been fit to data. If we have separate test data, we can use the "predict" method to generate a predicted list of classes for each sample. We can then calculate a score by dividing the total number of correct predictions by the total number of test samples. Alternatively, we can use the accuracy\_score function that's built into scikit-learn by passing the test set labels and the predictions.

**10. Let's practice!**

03:13 - 03:24

Now it's your turn. We'll practice visualizing our raw audio data, then creating some summary features from it that we'll feed into a classifier.

## Exercise

# 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.

## Instructions

100 XP

* First, create the time array for these audio files (all audios are the same length).
* Then, stack the values of the two DataFrames together (normal and abnormal, in that order) so that you have a single array of shape (n\_audio\_files, n\_times\_points).
* Finally, use the code provided to loop through each list item / axis, and plot the audio over time in the corresponding axis object.
* You'll plot normal heartbeats in the left column, and abnormal ones in the right column

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

# Calculate the time array

time = np.arange(\_\_\_\_) / \_\_\_\_

# Stack the normal/abnormal audio so you can loop and plot

stacked\_audio = np.hstack([\_\_\_\_, \_\_\_\_]).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()

**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()**

**Correct! 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.**

## Exercise

# 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.

## Instructions

100 XP

* Average across the audio files contained in normal and abnormal, leaving the time dimension.
* Visualize these averages over time.

**# Average across the audio files of each DataFrame**

**mean\_normal = np.mean(normal, axis=\_\_\_\_)**

**mean\_abnormal = np.mean(abnormal, axis=\_\_\_\_)**

**# Plot each average over time**

**fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 3), sharey=True)**

**ax1.plot(\_\_\_\_, \_\_\_\_)**

**ax1.set(title="Normal Data")**

**ax2.plot(\_\_\_\_, \_\_\_\_)**

**ax2.set(title="Abnormal Data")**

**plt.show()**

# 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(mean\_normal, time) ax1.set(title="Normal Data") ax2.plot(mean\_abnormal, time) ax2.set(title="Abnormal Data") plt.show()

**# 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()**

**Correct! 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.**

## Exercise

# 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.

## Instructions

100 XP

* Create an instance of the Linear SVC model and fit the model using the training data.
* Use the testing data to generate predictions with the model.
* Score the model using the provided code.

**from sklearn.svm import LinearSVC**

**# Initialize and fit the model**

**model = \_\_\_\_**

**model.\_\_\_\_**

**# Generate predictions and score them manually**

**predictions = model.\_\_\_\_**

**print(sum(predictions == y\_test.squeeze()) / len(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))

0.5555555555555556

**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))**

**Correct! 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.**

## 1. Improving the features we use for classification

00:00 - 00:09

What we've just performed is feature engineering of our audio data. Next, we'll cover a few more features that are more unique to timeseries data.

## 2. The auditory envelope

00:09 - 00:28

We'll begin by calculating the "envelope" of each heartbeat sound. The envelope throws away information about the fine-grained changes in the signal, focusing on the general shape of the audio waveform. To do this, we'll need to calculate the audio's amplitude, then smooth it over time.

## 3. Smoothing over time

00:28 - 00:44

First, we'll remove noise in timeseries data by smoothing it with a rolling window. This means defining a window around each timepoint, calculating the mean of this window, and then repeating this for each timepoint.

## 4. Smoothing your data

00:44 - 01:02

For example, on the left we have a noisy timeseries as well as an overlay of several small windows. Each timepoint will be replaced by the mean of the window just before it. The result is a smoother signal over time which you can see on the right.

## 5. Calculating a rolling window statistic

01:02 - 01:22

Let's cover how to do this with Pandas. We first use the dot-rolling method of our dataframe, which returns an object that can be used to calculate many different statistics within each window. The window parameter tells us how many timepoints to include in each window. The larger the window, the smoother the result will be.

## 6. Calculating the auditory envelope

01:22 - 01:45

Now that we know how to smooth our data, we can calculate the auditory envelope of our signal. First, we calculate the "absolute value" of each timepoint. This is also called "rectification", because you ensure that all time points are positive. Next, we calculate a rolling mean to smooth the signal. Let's see what these transformations look like.

## 7. The raw signal

01:45 - 01:50

First, we'll take a look at the raw audio signal.

## 8. Rectify the signal

01:50 - 01:54

Next, we take the absolute value of each timepoint.

## 9. Smooth the signal

01:54 - 02:04

Finally, we smooth the rectified signal. The result is a smooth representation of how the audio energy changes over time.

## 10. Feature engineering the envelope

02:04 - 02:19

Once we've calculated the acoustic envelope, we can create better features for our classifier. Here we'll calculate several common statistics of each auditory envelope, and combine them in a way that scikit-learn can use.

## 11. Preparing our features for scikit-learn

02:19 - 02:35

We'll then stack these features together with the same function we've used before. Even though we're calculating the same statistics (avg, standard deviation, and max), they are on different features, and so have different information about the stimulus.

## 12. Cross validation for classification

02:35 - 03:01

Now that our features are defined, lets fit a classifier and see how it performs. We'll use cross-validation in order to train and test the model on different subsets of data. We can use a single function to combine the steps of splitting data into training and validation sets, fitting the model on training data, and scoring predictions on validation data. Using "cross\_val\_score" will generate a list of scores across different "splits" of our data.

## 13. Using cross\_val\_score

03:01 - 03:24

To use it, pass an instance of a scikit-learn model as the first parameter, and the X and y data as second and third parameters. You can configure the strategy that scikit-learn uses to split the data with the CV parameter. Passing an integer will determine the number of splits that are made (and the number of scores generated).

## 14. Auditory features: The Tempogram

03:24 - 03:46

There are several more advanced features that can be calculated with timeseries data. Each attempts to detect particular patterns over time, and summarize them statistically. For example, a tempogram tells us the "tempo" of the sound at each moment. We'll show how to calculate it using a popular tool for audio analysis in Python called librosa.

## 15. Computing the tempogram

03:46 - 04:01

Here we show how librosa can be used to extract the tempogram from an audio array. This tells us the moment-by-moment tempo of the sound. We can then use this to calculate features for our classifier.

## 16. Let's practice!

04:01 - 04:14

Now it's your turn. We'll practice some simple feature engineering on auditory data using the techniques that we've discussed in this lesson. We'll then combine these features in a scikit-learn classifier.

## Exercise

# 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.

## Instructions 1/3

Visualize the raw audio you'll use to calculate the envelope.

**Plot the raw data first**

**audio.\_\_\_\_(figsize=(10, 5))**

**plt.show()**

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

# Plot the raw data first

audio.plot(figsize=(10, 5))

plt.show()

## Instructions 2/3

Rectify the audio.

Plot the result.

# Rectify the audio signal

audio\_rectified = audio.apply(\_\_\_\_)

# Plot the result

audio\_rectified.plot(figsize=(10, 5))

plt.show()

**# Rectify the audio signal**

**audio\_rectified = audio.apply(np.abs)**

**# Plot the result**

**audio\_rectified.plot(figsize=(10, 5))**

**plt.show()**

## Instructions 3/3

Smooth the audio file by applying a rolling mean.

Plot the result.

# Smooth by applying a rolling mean

audio\_rectified\_smooth = audio\_rectified.\_\_\_\_(50).\_\_\_\_()

# Plot the result

audio\_rectified\_smooth.plot(figsize=(10, 5))

plt.show()

# 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()

**# 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()**

**Yes! 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.**

## Exercise

# 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.

## Instructions

* Calculate the mean, standard deviation, and maximum value for each heartbeat sound.
* Column stack these stats in the same order.
* Use cross-validation to fit a model on each CV iteration.

# Calculate stats

means = np.\_\_\_\_(audio\_rectified\_smooth, axis=0)

stds = \_\_\_\_(audio\_rectified\_smooth, axis=0)

maxs = \_\_\_\_(audio\_rectified\_smooth, axis=0)

# Create the X and y arrays

X = np.column\_stack([\_\_\_\_, \_\_\_\_, \_\_\_\_])

y = labels.reshape(-1, 1)

# Fit the model and score on testing data

from sklearn.model\_selection import cross\_val\_score

percent\_score = \_\_\_\_(model, \_\_\_\_, \_\_\_\_, cv=5)

print(np.mean(percent\_score))

**# 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))**

**0.7166666666666667**

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

## Exercise

# 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.

## Instructions 1/2

Use librosa to calculate a tempogram of each heartbeat audio.

Calculate the mean, standard deviation, and maximum of each tempogram (this time using DataFrame methods)

# Calculate the tempo of the sounds

tempos = []

for col, i\_audio in audio.items():

    tempos.append(lr.beat.\_\_\_\_(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.\_\_\_\_(axis=-1)

tempos\_std = tempos.\_\_\_\_(axis=-1)

tempos\_max = tempos.\_\_\_\_(axis=-1)

# 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)

**# 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)**

## Exercise

# 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.

## Instructions 2/2

* Column stack these tempo features (mean, standard deviation, and maximum) in the same order.
* Score the classifier with cross-validation.

# 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

percent\_score = \_\_\_\_(model, X, y, cv=5)

print(np.mean(percent\_score)) 0.5

**# 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)) # 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))**

0.5

**Correct! 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!**

## 1. The spectrogram - spectral changes to sound over time

00:00 - 00:14

In this lesson, we'll discuss a special case of timeseries features: the spectrogram. Spectrograms are common in timeseries analysis, and we'll cover some basics to help you apply it to your machine learning problems.

## 2. Fourier transforms

00:14 - 00:34

To begin, we'll discuss a key part of the spectrogram: the Fourier Transform. This approach summarizes a time series as a collection of fast- and slow-moving waves. The Fourier Transform (or FFT) is a way to tell us how these waves can be combined in different amounts to create our time series.

## 3. A Fourier Transform (FFT)

00:34 - 01:01

On the left is a raw audio signal, and on the right is the Fourier Transform (or FFT) of the signal. This describes, for a window of time, the presence of fast- and slow-oscillations that are present in a timeseries. The slower oscillations are on the left (closer to 0) and the faster oscillations are on the right. This is a more rich representation of our audio signal.

## 4. Spectrograms: combinations of windows Fourier transforms

01:01 - 01:30

We can calculate multiple fourier transforms in a sliding window to see how it changes over time. For each timepoint, we take a window of time around it, calculate a fourier transform for the window, then slide to the next window (similar to calculating the rolling mean). The result is a description of the fourier transform as it changes throughout the timeseries. This is called a short-time fourier transform or STFT.

## 5. A Spectrogram Visualized

01:30 - 01:51

To calculate the spectrogram, we square each value of the STFT. An example is shown here. Note how the spectral content of the sound changes over time. Because this is speech, we see interesting patterns that correspond to spoken words (e.g. vowels or consonants).

## 6. Calculating the STFT

01:51 - 02:14

We'll use librosa's stft function to calculate a spectrogram. There are many parameters in this process, but we'll focus on the size of the window that is used. We'll calculate the STFT of our audio file, then convert the output to decibels to visualize it more cleanly with specshow (which results in the visualized spectrogram).

## 7. Calculating the STFT with code

02:14 - 02:54

Here's how to compute an STFT with librosa. We first define the size of the window used for the STFT. Next, we calculate the STFT, then convert it to decibels using the amplitude\_to\_db function, which ensures all values are positive, real numbers. Finally, we use the specshow function, which lets us quickly visualize a spectrogram. This code was used to produce the image shown in the previous slide. Note that we're glossing over some complex details for how spectrograms are calculated, but are focusing on the essentials for the purpose of fitting models.

## 8. Spectral feature engineering

02:54 - 03:13

Each timeseries has a unique spectral pattern to it. This means we can use patterns in the spectrogram to distinguish classes from one another. For example, we can calculate the spectral centroid and bandwidth over time. These describe where most of the spectral energy lies over time.

## 9. Calculating spectral features

03:13 - 03:40

To calculate the spectral centroid and bandwidth, we again turn to librosa. We'll use the spectral\_bandwidth and spectral\_centroid functions to calculate these values at each moment in time for the spectrogram we've computed. These functions could also accept a raw audio signal (in which case the STFT will be performed first). This visualization code is what produced the figure on the previous slide.

## 10. Combining spectral and temporal features in a classifier

03:40 - 04:08

In this chapter, we've calculated many different kinds of auditory features from our heartbeat sounds. As a final step, we can combine each of the features mentioned before into a single input matrix for our classifier. Here we calculate the mean value of the spectral centroid and bandwidth, and stack these into a single classifier input matrix. In general, as we include more complex features into our model, we'll improve model performance.

## 11. Let's practice!

04:08 - 04:13

Now let's try some examples.

## Exercise

# 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.

## Instructions 1/2

Import the short-time fourier transform (stft) function from librosa.core.

Calculate the spectral content (using the short-time fourier transform function) of audio.

# Import the stft function

\_\_\_\_

# Prepare the STFT

HOP\_LENGTH = 2\*\*4

spec = \_\_\_\_(audio, hop\_length=HOP\_LENGTH, n\_fft=2\*\*7)

# 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)

# 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)

* Convert the spectogram (spec) to decibels.
* Visualize the spectogram.

from librosa.core import amplitude\_to\_db

from librosa.display import specshow

# Convert into decibels

spec\_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)

\_\_\_\_(spec\_db, sr=sfreq, x\_axis='time', y\_axis='hz', hop\_length=HOP\_LENGTH, ax=axs[1])

plt.show()

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, ax=axs[1]) plt.show()

f**rom 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, ax=axs[1])**

**plt.show()**

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

## Exercise

## Exercise

# 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.

## Instructions 1/2

Calculate the **spectral bandwidth** as well as the **spectral centroid** of the spectrogram by using functions in librosa.feature.

import librosa as lr

# Calculate the spectral centroid and bandwidth for the spectrogram

bandwidths = lr.feature.\_\_\_\_(S=\_\_\_\_)[0]

centroids = lr.feature.\_\_\_\_(S=\_\_\_\_)[0]

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]

* Convert the spectrogram to decibels for visualization.
* Plot the spectrogram over time.

from librosa.core import amplitude\_to\_db

from librosa.display import specshow

# Convert spectrogram to decibels for visualization

spec\_db = \_\_\_\_(spec)

# Display these features on top of the spectrogram

fig, ax = plt.subplots(figsize=(10, 5))

**ax = \_\_\_\_(\_\_\_\_, x\_axis='time', y\_axis='hz', hop\_length=HOP\_LENGTH, ax=ax)**

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()

**should not have ax= leave it with specshow()**

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 =** not included

specshow(spec\_db, ax=ax, 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()

**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))**

**specshow(spec\_db, ax=ax, 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()**

Good job! 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.

## Exercise

## Exercise

# 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.

## Instructions 1/2

Loop through each spectrogram, calculating the mean spectral bandwidth and centroid of each.

# Loop through each spectrogram

bandwidths = []

centroids = []

for spec in spectrograms:

    # Calculate the mean spectral bandwidth

    this\_mean\_bandwidth = np.\_\_\_\_(lr.feature.\_\_\_\_(S=spec))

    # Calculate the mean spectral centroid

    this\_mean\_centroid = np.\_\_\_\_(lr.feature.\_\_\_\_(S=spec))

    # Collect the values

    bandwidths.append(this\_mean\_bandwidth)

    centroids.append(this\_mean\_centroid)

# 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)

**# 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)**

* Column stack all the features to create the array X.
* Score the classifier with cross-validation.

# Create X and y arrays

X = \_\_\_\_([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 = \_\_\_\_(model, X, y, cv=5)

print(np.mean(percent\_score))

**# 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))**

**0.4833333333333333**

**# 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))**

**Good job! 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.**

## 1. Predicting data over time

00:00 - 00:24

In the third chapter we'll shift our focus from classification to regression. Regression has several features and caveats that are unique to timeseries data. We'll begin by visualizing and predicting timeseries data. Then, we'll cover the basics of cleaning the data, and finally, we'll begin extracting features that we can use in our models.

## 2. Classification vs. Regression

00:24 - 00:41

The biggest difference between regression and classification is that regression models predict continuous outputs whereas classification models predict categorical outputs. In the context of timeseries, this means we can have more fine-grained predictions over time.

## 3. Correlation and regression

00:41 - 01:10

Both Regression and correlation reflect the extent to which the values of two variables have a consistent relationship (either they both go down or up together, or they have an inverse relationship). However, regression results in a "model" of the data, while correlation is just a single statistic that describes the data. Regression models have more information about the data, while correlation is easier to calculate and interpret.

## 4. Correlation between variables often changes over time

01:10 - 01:25

When running regression models with timeseries data, it's important to visualize how the data changes over time. You can either do this by plotting the whole timeseries at once, or by directly comparing two segments of time.

## 5. Visualizing relationships between timeseries

01:25 - 01:40

Here we show two ways to compare timeseries data. On the left, we'll make two line plots with the x-axis encoding time. On the right, we'll make a single scatterplot, with color encoding time.

## 6. Visualizing two timeseries

01:40 - 02:04

Here is the visualization. In this case, it seems like these two timeseries are uncorrelated at first, but then move in sync with one another. We can confirm this by looking at the brighter colors on the right. We see that brighter datapoints fall on a line, meaning that for those moments in time, the two variables had a linear relationship.

## 7. Regression models with scikit-learn

02:04 - 02:32

Fitting regression models with scikit-learn works the same way as classifiers - the consistency in API is one of scikit-learn's greatest strengths. There are, however, a completely different subset of models that accomplish regression. We'll begin by focusing on LinearRegression, which is the simplest form of regression. Here we see how you can instantiate the model, fit, and predict on training data.

## 8. Visualize predictions with scikit-learn

02:32 - 02:59

Here we visualize the predictions from several different models fit on the same data. We'll use Ridge regression, which has a parameter called "alpha" that causes coefficients to be smoother and smaller, and is useful if you have noisy or correlated variables. We loop through a few values of alpha, initializing a model with each one and fitting it on the training data. We then plot the model's predictions on the test data,

## 9. Visualize predictions with scikit-learn

02:59 - 03:11

which lets us see what each model is getting right and wrong. For more information on Ridge regression, refer to DataCamp's introductory course on scikit-learn.

## 10. Scoring regression models

03:11 - 03:26

Visualizing is useful, but not quantifiable. There are several options for scoring a regression model. The simplest is the correlation coefficient, whereas the most common is the coefficient of determination, or R squared.

## 11. Coefficient of Determination ($R^2$)

03:26 - 03:57

The coefficient of determination can be summarized as the total amount of error in your model (the difference between predicted and actual values) divided by the total amount of error if you'd built a "dummy" model that simply predicted the output data's mean value at each timepoint. You subtract this ratio from "1", and the result is the coefficient of determination. It is bounded on top by "1", and can be infinitely low (since models can be infinitely bad).

## 12. $R^2$ in scikit-learn

03:57 - 04:12

In scikit-learn, we can import the r2\_score function which calculates the coefficient of determination. It takes the predicted output values first, and the "true" output values second, to calculate r-square.

## 13. Let's practice!

04:12 - 04:22

Now let's try some examples. We'll practice visualizing continuous timeseries data, as well as fitting some simple models.

## Exercise

# Introducing the dataset

As mentioned in the video, you'll 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.

## Instructions 1/3

Plot the data in prices. Pay attention to any irregularities you notice.

Generate a scatter plot with the values of Ebay on the x-axis, and Yahoo on the y-axis. Look up the symbols for both companies from the column names of the DataFrame.

Finally, encode time as the color of each datapoint in order to visualize how the relationship between these two variables changes.

# Plot the raw values over time

prices.\_\_\_\_()

plt.show()

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

**# Plot the raw values over time**

**prices.plot()**

**plt.show()**

# Scatterplot with one company per axis

prices.plot.\_\_\_\_(\_\_\_\_, \_\_\_\_)

plt.show()

# Scatterplot with one company per axis

prices.plot.scatter('EBAY', 'YHOO')

plt.show()

**In [1]:**

**print(prices.columns)**

**Index(['EBAY', 'YHOO'], dtype='object', name='symbol')**

**# 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=\_\_\_\_,

                    cmap=plt.cm.viridis, colorbar=False)

plt.show()

**# Scatterplot with color relating to time**

**prices.plot.scatter('EBAY', 'YHOO', c=prices.index,**

**cmap=plt.cm.viridis, colorbar=False)**

**plt.show()**

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

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

## Exercise

# 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.

## Instructions

100 XP

* Create the X and y arrays by using the column names provided.
* The input values should be from the companies "ebay", "nvidia", and "yahoo"
* The output values should be from the company "apple"
* Use the data to train and score the model with cross-validation.

from sklearn.linear\_model import Ridge

from sklearn.model\_selection import cross\_val\_score

# Use stock symbols to extract training data

X = all\_prices[[\_\_\_\_]]

y = all\_prices[[\_\_\_\_]]

# Fit and score the model with cross-validation

scores = cross\_val\_score(Ridge(), \_\_\_\_, \_\_\_\_, cv=3)

print(scores)

**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]

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)

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

## Exercise

# 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.

## Instructions 1/2

Split the data (X and y) into training and test sets.

Use the training data to train the regression model.

Then use the testing data to generate predictions for the model.

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(\_\_\_\_, \_\_\_\_,

                                                    train\_size=.8, shuffle=False)

# Fit our model and generate predictions

model = Ridge()

model.fit(\_\_\_\_, \_\_\_\_)

predictions = model.predict(\_\_\_\_)

score = r2\_score(y\_test, predictions)

print(score)

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)

# 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)

-5.709399019485158

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)

# 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)

Plot a time series of the predicted and "actual" values of the testing data.

# 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(\_\_\_\_, color='r', lw=2)

plt.show()

# 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()

# 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()

**Correct! Now you have an explanation for your poor score. The predictions clearly deviate from the true time series values.**

## 1. Cleaning and improving your data

00:00 - 00:09

Now that we've covered some simple visualizations and model fitting with continuous timeseries, let's see what happens when we look at more real-world data.

## 2. Data is messy

00:09 - 00:29

Real-world data is always messy, and requires preparing and cleaning the data before fitting models. In timeseries, messy data often happens due to failing sensors or human error in logging the data. Let's cover some specific ways to spot and fix messy data with timeseries.

## 3. What messy data looks like

00:29 - 01:04

First, let's look at some messy-looking data. Here, we're showing the value of the company AIG over the last several years. There seem to be two periods of time where no data was produced, as well as some periods of time where the data doesn't fluctuate at all. Both look like they're aberrations, so let's see how we can correct for them. Before moving forward, note that it is not always clear whether patterns in the data are "aberrations" or not. You should always investigate to understand the source of strange patterns in the data.

## 4. Interpolation: using time to fill in missing data

01:04 - 01:18

First, let's fill in the missing data using other datapoints we do have. We'll use a technique called interpolation, which uses the values on either end of a missing window of time to infer what's in-between.

## 5. Interpolation in Pandas

01:18 - 01:39

In this example, we'll first create a boolean mask that we'll use to mark where the missing values are. Next, we call the dot-interpolate method to fill in the missing values. We'll use the first argument to signal we want linear interpolation. Finally, we'll plot the interpolated values.

## 6. Visualizing the interpolated data

01:39 - 01:57

You can see the results of interpolation in red. In this case, we used the "linear" argument so the interpolated values are a line between the start and stop point of the missing window. Other arguments to the dot-interpolate method will result in different behavior.

## 7. Using a rolling window to transform data

01:57 - 02:09

Another common technique to clean data is transforming it so that it is more well-behaved. To do this, we'll use the same rolling window technique covered in Chapter 2.

## 8. Transforming data to standardize variance

02:09 - 02:22

Using a rolling window, we'll calculate each timepoint's percent change over the mean of a window of previous timepoints. This standardizes the variance of our data and reduces long-term drift.

## 9. Transforming to percent change with Pandas

02:22 - 02:41

In this function, we first separate out the final value of the input array. Then, we calculate the mean of all but the last datapoint. Finally, we subtract the mean from the final datapoint, and divide by the mean. The result is the percent change for the final value.

## 10. Applying this to our data

02:41 - 02:57

We can apply this to our data using the dot-aggregate method, passing our function as an input. On the right, the data is now roughly centered at zero, and periods of high and low changes are easier to spot.

## 11. Finding outliers in your data

02:57 - 03:13

We'll use this transformation to detect outliers. Outliers are datapoints that are statistically different from the dataset as a whole. A common definition is any datapoint that is more than three standard deviations away from the mean of the dataset.

## 12. Plotting a threshold on our data

03:13 - 03:30

Here we'll visualize our definition of an outlier. We calculate the mean and standard deviation of each dataset, then plot outlier "thresholds" (three times the standard deviation from the mean) on the raw and transformed data.

## 13. Visualizing outlier thresholds

03:30 - 03:49

Here is the result. Any datapoint outside these bounds could be an outlier. Note that the 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.

## 14. Replacing outliers using the threshold

03:49 - 04:18

Next, we replace outliers with the median of the remaining values. We first center the data by subtracting its mean, and calculate the standard deviation. Finally, we calculate the absolute value of each datapoint, and mark any that lie outside of three standard deviations from the mean. We then replace these using the nanmedian function, which calculates the median without being hindered by missing values.

## 15. Visualize the results

04:18 - 04:28

As you can see, once we've replaced the outliers, there don't seem to be as many extreme datapoints. This should help our model find the patterns we want.

## 16. Let's practice!

04:28 - 04:36

Now we'll practice some basics in cleaning up the data and identifying outliers.

## Exercise

# 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.

## Instructions

100 XP

* Visualize the time series data using Pandas.
* Calculate the number of missing values in each time series. Note any irregularities that you can see. What do you think they are?

# Visualize the dataset

prices.\_\_\_\_(legend=False)

plt.tight\_layout()

plt.show()

# Count the missing values of each time series

missing\_values = prices.\_\_\_\_.\_\_\_\_

print(missing\_values)

# 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)

symbol

EBAY 273

NVDA 502

YHOO 232

dtype: int64

# 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)

**Nice job. 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?**

## Exercise

# 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.

## Instructions 1/4

Create a boolean mask for missing values and interpolate the missing values using the interpolation argument of the function.

Interpolate using the latest non-missing value and plot the results.

Recall that interpolate\_and\_plot's second input is a string specifying the kind of interpolation to use.

Interpolate linearly and plot the results.

Interpolate with a quadratic function and plot the results.

# 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.\_\_\_\_()

    # Interpolate the missing values

    prices\_interp = prices.\_\_\_\_(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()

# 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()

**# 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 = \_\_\_\_

interpolate\_and\_plot(prices, interpolation\_type)

# Interpolate using the latest non-missing value interpolation\_type = 'zero' interpolate\_and\_plot(prices, interpolation\_type)

**# Interpolate using the latest non-missing value**

**interpolation\_type = 'zero'**

**interpolate\_and\_plot(prices, interpolation\_type)**

# Interpolate linearly

interpolation\_type = \_\_\_\_

interpolate\_and\_plot(prices, interpolation\_type)

# Interpolate linearly

interpolation\_type = 'linear'

interpolate\_and\_plot(prices, interpolation\_type)

**# Interpolate linearly interpolation\_type = 'linear' interpolate\_and\_plot(prices, interpolation\_type**)

# Interpolate with a quadratic function

interpolation\_type = \_\_\_\_

interpolate\_and\_plot(prices, interpolation\_type)

# Interpolate with a quadratic function interpolation\_type = 'quadratic' interpolate\_and\_plot(prices, interpolation\_type)

# Interpolate with a quadratic function

interpolation\_type = 'quadratic'

interpolate\_and\_plot(prices, interpolation\_type)

**Correct! 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.**

## Exercise

# 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.

## Instructions

* Define a percent\_change function that takes an input time series and does the following:
  + Extract all but the last value of the input series (assigned to previous\_values) and the only the last value of the timeseries ( assigned to last\_value)
  + Calculate the percentage difference between the last value and the mean of earlier values.
* Using a rolling window of 20, apply this function to prices, and visualize it using the given code.

# Your custom function

def percent\_change(series):

    # Collect all \*but\* the last value of this window, then the final value

    previous\_values = series[:\_\_\_\_]

    last\_value = series[-1]

    # Calculate the % difference between the last value and the mean of earlier values

    percent\_change = (\_\_\_\_ - np.mean(previous\_values)) / np.mean(previous\_values)

    return percent\_change

# Apply your custom function and plot

prices\_perc = prices.rolling(20).\_\_\_\_

prices\_perc.loc["2014":"2015"].plot()

plt.show()

# 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()

# 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()

**Correct! You've converted the data so it's easier to compare one time point to another. This is a cleaner representation of the data**

## Exercise

# 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.

## Instructions

* Define a function that takes an input series and does the following:
  + Calculates the absolute value of each datapoint's distance from the series mean, then creates a boolean mask for datapoints that are three times the standard deviation from the mean.
  + Use this boolean mask to replace the outliers with the median of the entire series.
* Apply this function to your data and visualize the results using the given code.

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 zero

    this\_mask = absolute\_differences\_from\_mean > (np.\_\_\_\_(series) \* \_\_\_\_)

    # Replace these values with the median accross the data

    series[this\_mask] = np.\_\_\_\_(series)

    return series

# Apply your preprocessing function to the timeseries and plot the results

prices\_perc = prices\_perc.\_\_\_\_

prices\_perc.loc["2014":"2015"].plot()

plt.show()

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 zero 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()

**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 zero**

**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()**

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

## 1. Creating features over time

00:00 - 00:07

In the final lesson of this chapter, we'll cover some specific features that are useful in timeseries analysis.

## 2. Extracting features with windows

00:07 - 00:25

Remember the rolling window used earlier to smooth our data? We can use the same technique to extract features as they change over time. In this image, we can define multiple functions of each window to extract many features at once.

## 3. Using .aggregate for feature extraction

00:25 - 00:58

In pandas, the dot-aggregate method can be used to calculate many features of a window at once. By passing a list of functions to the method, each function will be called on the window, and collected in the output. Here's an example - we first use the dot-rolling method to define a rolling window, then pass a list of two functions (for the standard deviation, and maximum value). This extracts two features for each column over time.

## 4. Check the properties of your features!

00:58 - 01:17

You can extract many different kinds of features this way. Always plot the features you've extracted over time, as this can give you a clue for how they behave and help you spot noisy data and outliers. Here we can see that the maximum value is much jumpier than the mean.

## 5. Using partial() in Python

01:17 - 02:01

A useful tool when using the dot-aggregate method is the partial function. This is built-in to Python, and lets you create a \*new\* function from an old one, with some of the parameters pre-configured. Let's see how this works. In this example, we first import partial from functools, then use it to create a mean function that always operates on the first axis. The first argument is the function we want to modify, and subsequent key-value pairs will be pre-set in the output function. After this, we no longer need to configure those values when we call the new function.

## 6. Percentiles summarize your data

02:01 - 02:41

Now, back to feature extraction. A particularly useful tool for feature extraction is the percentile function. This is similar to calculating the mean or median of your data, but it gives you more fine-grained control over what is extracted. The percentile function takes an array as the first input, and an integer between 0 and 100 as the second input. It will return the value in the input array that matches the percentile you've chosen. Here it returns 40, which means that the value "40" is larger than 20% of the input array.

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

02:41 - 03:12

Here we'll combine the percentile function with partial functions in order to extract several percentiles with the dot-aggregate method. We use a list comprehension to create a list of functions (called percentile\_funcs). Then, we loop through the list, calling each function on our data, to return a different percentile of the data. We could pass this list of partial functions to our dot-aggregate method to extract several percentiles for each column.

## 8. Calculating "date-based" features

03:12 - 03:45

Another common feature to consider are "date-based" features. That is, features that take into consideration information like "what time of the year is it?" or "is it a holiday?". Since many datasets involve humans, these pieces of information are often important. For example, if you're trying to predict the number of customers that will visit your store each day, it's important to know if it's the weekend or not! Working with dates and times is straightforward in Pandas, which we'll cover next.

## 9. datetime features using Pandas

03:45 - 04:12

Datetime functionality is most commonly accessed with a DataFrame's index. As you saw in the first chapter, you can use the to\_datetime function to ensure dates are treated as datetime objects. You can also extract many date-specific pieces of information, such as the day of the week, or weekday name as shown here. These could then be treated as features in your model.

## 10. Let's practice!

04:12 - 04:21

Now that we've got a few more features to consider, let's practice extracting them and visualizing how they look on our data.

## Exercise

# 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.

## Instructions

* Define a list consisting of four features you will calculate: the minimum, maximum, mean, and standard deviation (in that order).
* Using the rolling window (prices\_perc\_rolling) we defined for you, calculate the features from features\_to\_calculate.
* Plot the results over time, along with the original time series using the given code.

# 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, \_\_\_\_, \_\_\_\_, \_\_\_\_]

# Calculate these features for your rolling window object

features = prices\_perc\_rolling.\_\_\_\_(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()

# 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()

**# 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()**

# 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()

**Well done! In the next exercise, you will calculate the percentiles.**

## Exercise

# 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.

## Instructions

* Import partial from functools.
* Use the partial() function to create several feature generators that calculate percentiles of your data using a list comprehension.
* Using the rolling window (prices\_perc\_rolling) we defined for you, calculate the quantiles using percentile\_functions.
* Visualize the results using the code given to you.

# 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()

**# 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()**

**Nice rolling! In the next exercise, you will extract the date components of the timestamps.**

## Exercise

# 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.

## Instructions

* Calculate the **day of the week**, **week number in a year**, and **month number in a year**.
* Add each one as a column to the prices\_perc DataFrame, under the names day\_of\_week, week\_of\_year and month\_of\_year, respectively.

# Extract date features from the data, add them as columns

prices\_perc['day\_of\_week'] = prices\_perc.\_\_\_\_.\_\_\_\_

prices\_perc['week\_of\_year'] = prices\_perc.\_\_\_\_.\_\_\_\_

prices\_perc['month\_of\_year'] = prices\_perc.\_\_\_\_.\_\_\_\_

# Print prices\_perc

print(prices\_perc)

# 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)

EBAY day\_of\_week week\_of\_year month\_of\_year

date

2014-01-02 0.018 3 1 1

2014-01-03 0.002 4 1 1

2014-01-06 -0.027 0 2 1

2014-01-07 -0.007 1 2 1

2014-01-08 -0.017 2 2 1

... ... ... ... ...

2015-12-24 -0.029 3 52 12

2015-12-28 -0.027 0 53 12

2015-12-29 -0.014 1 53 12

2015-12-30 -0.017 2 53 12

2015-12-31 -0.025 3 53 12

[504 rows x 4 columns]

**# 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)**

Good job! This concludes the third chapter. In the next chapter, you will learn advanced techniques to validate and inspect your time series models.

## 1. Time-delayed features and auto-regressive models

00:00 - 00:17

One of the most important steps in a machine learning pipeline is feature extraction. Defining high-quality and relevant features gives your model the best chance at finding useful patterns in the data. In this lesson, we'll cover some techniques for extracting features from data.

## 2. The past is useful

00:17 - 00:41

Perhaps the biggest difference between timeseries data and "non-timeseries" data is the relationship between data points. Because the data has a linear flow (matching the progression of time), patterns will persist over a span of datapoints. As a result, we can use information from the past in order to predict values in the future.

## 3. A note on smoothness and auto-correlation

00:41 - 01:07

It's important to consider how "smooth" your data is when fitting models with timeseries. The smoothness of your 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 timepoints is often described as "autocorrelation", and can have a big impact on the performance of your model.

## 4. Creating time-lagged features

01:07 - 01:26

Let's investigate this by creating a model in which previous timepoints are used as input features to the model. Remember that regression models will assign a "weight" to each input feature, and we can use these weights to determine how "smooth" or "autocorrelated" the signal is.

## 5. Time-shifting data with Pandas

01:26 - 01:52

First we'll create time-shifted versions of our data. This entails "rolling" your data either into the future or into the past, so that the same index of data now has an different timepoint in it. We can do this in Pandas by using the dot-shift method of a DataFrame. Positive values roll the data backward, while negative values roll the data forward.

## 6. Creating a time-shifted DataFrame

01:52 - 02:22

Here we use a dictionary comprehension that creates several time-lagged versions of the data. Each one shifts the data a different number of indices into the past. Since our data is recorded daily, this corresponds to shifting the data so that each index corresponds to the value of the data N days prior. We can then convert this into a DataFrame where dictionary keys become column names.

## 7. Fitting a model with time-shifted features

02:22 - 02:45

We will now fit a scikit-learn regression model. Note that in this case, "many\_shifts" is simply a time-shifted version of the timeseries contained in the "data" variable. We'll fit the model using Ridge regression, which spreads out weights across features (if applicable) rather than assign it all to a single feature.

## 8. Interpreting the auto-regressive model coefficients

02:45 - 03:06

Once we fit the model, we can investigate the coefficients it has found. Larger absolute values of coefficients mean that a given feature has a large impact on the output variable. We can use a bar plot in Matplotlib to visualize the model's coefficients that were created after fitting the model.

## 9. Visualizing coefficients for a rough signal

03:06 - 03:22

Here we see the coefficient values for a relatively non-smooth signal. On the left the signal is clearly jumping around, and on the right we see the model coefficients (one per time lag) drop to zero very quickly.

## 10. Visualizing coefficients for a smooth signal

03:22 - 03:35

And here is a timeseries that is more smooth. On the left we can see more "structure" to the data, and on the right we see that the coefficients for time lags drop off to zero smoothly.

## 11. Let's practice!

03:35 - 03:41

Now let's practice creating time-lagged features and using them to fit models.

## Exercise

# 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.

## Instructions

* Use a dictionary comprehension to create multiple time-shifted versions of prices\_perc using the lags specified in shifts.
* Convert the result into a DataFrame.
* Use the given code to visualize the results.

# 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.\_\_\_\_(\_\_\_\_) for day\_shift in shifts}

# Convert into a DataFrame for subsequent use

prices\_perc\_shifted = \_\_\_\_(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()

# 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()

**# 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()**

## Exercise

# 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.

## Instructions

* Replace missing values in prices\_perc\_shifted with the median of the DataFrame and assign it to X.
* Replace missing values in prices\_perc with the median of the series and assign it to y.
* Fit a regression model using the X and y arrays.

# Replace missing values with the median for each column

X = prices\_perc\_shifted.\_\_\_\_(np.\_\_\_\_(prices\_perc\_shifted))

y = prices\_perc.\_\_\_\_(np.\_\_\_(prices\_perc))

# Fit the model

model = Ridge()

model.fit(\_\_\_\_, \_\_\_\_)

# 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)

Ridge()

# 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)

**Correct! You've filled in the missing values with the median so that it behaves well with scikit-learn. Now let's take a look at what your model found.**

## Exercise

# 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.

## Instructions 1/2

Define a function (called visualize\_coefficients) that takes as input an array of coefficients, an array of each coefficient's name, and an instance of a Matplotlib axis object. It should then generate a bar plot for the input coefficients, with their names on the x-axis.

def visualize\_coefficients(coefs, names, ax):

    # Make a bar plot for the coefficients, including their names on the x-axis

    ax.bar(\_\_\_\_, \_\_\_\_)

    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

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

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

## Instructions 2/2

Use this function (visualize\_coefficients()) with the coefficients contained in the model variable and column names of prices\_perc\_shifted.

# 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(\_\_\_\_, \_\_\_\_, ax=axs[1])

plt.show()

# 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()

# 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.**

## Exercise

# 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.

## Instructions

Using the function (visualize\_coefficients()) you created in the last exercise, generate a plot with coefficients of model and column names of prices\_perc\_shifted.

# 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(\_\_\_\_, \_\_\_\_, ax=axs[1])

plt.show()

# 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()

# 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()

**Correct! 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.**

## 1. Cross-validating timeseries data

00:00 - 00:08

We'll now discuss some of the basics of cross-validation, as well as how they relate to fitting timeseries data.

## 2. Cross validation with scikit-learn

00:08 - 00:17

Scikit-learn has many classes for cross-validation. For reference, here is the standard way to use each one of them.

## 3. Cross validation types: KFold

00:17 - 00:39

The most common form of cross-validation is k-fold cross-validation. In this case, data is split into K subsets of equal size. In each iteration, a single subset is left out as the validation set. Here we show how to initialize a k-fold iterator with scikit-learn.

## 4. Visualizing model predictions

00:39 - 00:58

Always visualize your model's behavior during cross-validation. Here, we first show how to plot the indices of the validation set for each iteration. Next, we visualize the predictions generated in this loop. This helps us perform sanity checks on the process.

## 5. Visualizing KFold CV behavior

00:58 - 01:21

Looking at the generated plot, we first see that the validation indices are chunked in "blocks" or "folds". This is the default cross-validation behavior. On the bottom we see three timeseries that were predicted, one for each iteration. They have the same general structure as time series data, which is a good sanity check.

## 6. A note on shuffling your data

01:21 - 01:49

Many cross-validation iterators let you randomly shuffle the data. This may be appropriate if your data is independent and identically distributed, but timeseries data is usually not i.i.d. Here we use the "ShuffleSplit" cross-validation iterator, which randomly permutes the data labels in each iteration. Let's see what our visualization looks like when using shuffled data.

## 7. Visualizing shuffled CV behavior

01:49 - 02:22

Here we can see from the top plot that the validation indices are all over the place, not in chunks like before. That's because we used a cross-validation object that shuffles the data. Below we see that the output data no longer "looks" like timeseries, because the temporal structure of the data has been destroyed. If the data is shuffled, it means that some information about the training set now exists in the validation set, and you can no longer trust the score of your model.

## 8. Using the time series CV iterator

02:22 - 02:59

Finally, we'll cover another cross-validation iterator that can help deal with these problems. There is one cross-validation technique that is meant particularly for time series data. This approach \*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. This more closely mimics the data collection and prediction process in the real world.

## 9. Visualizing time series cross validation iterators

02:59 - 03:15

We can use this cross-validation approach with the TimeSeriesSplit object in scikit-learn. Here we iterate through the object and plot the training data in blue and the validation data in red. Let's visualize this behavior.

## 10. Visualizing the TimeSeriesSplit cross validation iterator

03:15 - 03:39

Here we see how the TimeSeriesSplit CV object behaves. Each row is an iteration of the cross validation. In red we see the validation indices for that iteration. As you can see, the training data always comes before the validation data. This ensures that only the past is always used to predict the future.

## 11. Custom scoring functions in scikit-learn

03:39 - 03:59

You can also create custom scorers in scikit-learn. They must all take an estimator object, and X and y arrays as inputs, and output a single number, representing the score calculated after generating model predictions. You can use whatever function you like to create this score.

## 12. A custom correlation function for scikit-learn

03:59 - 04:19

Here we define our custom correlation function for scikit-learn. It generates model predictions, then uses numpy's corrcoef function to output a correlation matrix. We take a single value from this matrix, since there are only two variables in it, and return the value.

## 13. Let's practice!

04:19 - 04:26

We'll now use this custom function in the exercises to assess the scores of our time series models.

## Exercise

# 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.

## NormalInstructionHeader.heading

* Initialize a ShuffleSplit cross-validation object with 10 splits.
* Iterate through CV splits using this object. On each iteration:
  + Fit a model using the training indices.
  + Generate predictions using the test indices, score the model (

) using the predictions, and collect the results.

# Import ShuffleSplit and create the cross-validation object

from sklearn.model\_selection import ShuffleSplit

cv = \_\_\_\_(\_\_\_\_, random\_state=1)

# Iterate through CV splits

results = []

for tr, tt in cv.\_\_\_\_(X, y):

    # Fit the model on training data

    \_\_\_\_(X[tr], y[tr])

    # Generate predictions on the test data, score the predictions, and collect

    prediction = \_\_\_\_(X[tt])

    score = r2\_score(\_\_\_\_, \_\_\_\_)

    results.append((prediction, score, tt))

# Custom function to quickly visualize predictions

visualize\_predictions(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)

**# 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 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.**

## Exercise

# 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.

## NormalInstructionHeader.heading

* Instantiate another cross-validation object, this time using KFold cross-validation with 10 splits and no shuffling.
* Iterate through this object to fit a model using the training indices and generate predictions using the test indices.
* Visualize the predictions across CV splits using the helper function (visualize\_predictions()) we've provided.

# Create KFold cross-validation object

from sklearn.model\_selection import KFold

cv = \_\_\_\_(n\_splits=\_\_\_\_, shuffle=\_\_\_\_)

# Iterate through CV splits

results = []

for tr, tt in cv.split(X, y):

    # Fit the model on training data

    model.fit(\_\_\_\_)

    # Generate predictions on the test data and collect

    prediction = model.predict(\_\_\_\_)

    results.append((prediction, tt))

# Custom function to quickly visualize predictions

visualize\_predictions(results)

# Create KFold cross-validation object from sklearn.model\_selection import KFold cv = KFold(n\_splits=10, shuffle=False) # 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)

**# Create KFold cross-validation object**

**from sklearn.model\_selection import KFold**

**cv = KFold(n\_splits=10, shuffle=False)**

**# 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)**

**Good job! 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.**

## Exercise

# 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.

## NormalInstructionHeader.heading

* Import TimeSeriesSplit from sklearn.model\_selection.
* Instantiate a time series cross-validation iterator with 10 splits.
* Iterate through CV splits. On each iteration, visualize the values of the input data that would be used to train the model for that iteration.

# Import TimeSeriesSplit

\_\_\_\_

# Create time-series cross-validation object

cv = \_\_\_\_

# 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()

# 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()

**# 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()**

**Good job! 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.**

## 1. Stationarity and stability

00:00 - 00:08

In this lesson, we'll cover how to quantify variability in our models and how this relates to time series data.

## 2. Stationarity

00:08 - 00:29

A stationary signal is one that does not change its statistical properties over time. It has the same mean, standard deviation, and general trends. A non-stationary signal does change its properties over time. Each of these has important implications for how to fit your model.

## 3. Examples of stationary and non-stationary data

00:29 - 01:01

Here's an example of a stationary and a non-stationary signal. On the top, we can see that the signal generally does not change its structure. Its variability is constant throughout time. On the bottom, we see a signal that is highly non-stationary. Its variance and trends change over time. Almost all real world data are non-stationary. In fact, these two plots are of the same data, but in different ranges of time.

## 4. Model stability

01:01 - 01:22

Most models have an implicit assumption that the relationship between inputs and outputs is static. If this relationship changes (because the data is not stationary), then the model will generate predictions using an outdated relationship between inputs and outputs. How can we quantify and correct for this?

## 5. Cross validation to quantify parameter stability

01:22 - 01:43

One approach is to use cross-validation, which yields a set of model coefficients per iteration. We can quantify the variability of these coefficients across iterations. If a model's coefficients vary widely between cross-validation splits, there's a good chance the data is non-stationary (or noisy).

## 6. Bootstrapping the mean

01:43 - 02:09

Bootstrapping is a way to estimate the confidence in the mean of a collection of numbers. To perform a bootstrap for the mean, take many random samples (with replacement) from your collection of numbers and calculate the mean of each. Now, calculate lower/upper percentiles for this list. The lower and upper percentiles represent the variability of the mean.

## 7. Bootstrapping the mean

02:09 - 02:42

Here's an example using scikit-learn and numpy. Use the resample function in scikit-learn to take a random sample of coefficients, then use numpy to calculate the mean for each coefficient in the sample and store it in an array. Then, we calculate the 2-point-5 and 97-point-5 percentile of the results to calculate lower and upper bounds for each coefficient. This is called a 95% confidence interval.

## 8. Plotting the bootstrapped coefficients

02:42 - 02:55

Here we plot the lower and upper bounds of the 95% confidence intervals we calculated. This gives us an idea for the variability of the mean across all cross-validation iterations.

## 9. Assessing model performance stability

02:55 - 03:10

It's also common to quantify the stability of a model's predictive power across cross-validation folds. If you're using the TimeSeriesSplit object mentioned before, then you can visualize this as a timeseries.

## 10. Model performance over time

03:10 - 03:42

In this example, we'll use the cross\_val\_score function, along with the TimeSeriesSplit iterator, to calculate the predictive power of the model over cross-validation splits. We first create a small scoring function that can be passed to cross\_val\_score. Next we use a list comprehension to find the date of the beginning of each validation block. Finally, we collect the scores and convert them into a Pandas Series.

## 11. Visualizing model scores as a timeseries

03:42 - 04:06

Because the cross-validation splits happen linearly over time, we can visualize the results as a timeseries. If we see large changes in the predictive power of a model at one moment in time, it could be because the statistics of the data have changed. Here we create a rolling mean of our cross-validation scores and plot it with matplotlib.

## 12. Visualizing model scores

04:06 - 04:20

We can see the scores of our model across validation sets, which means over time. There is a clear dip in the middle, probably because the statistics of the data changed. What can we do about this?

## 13. Fixed windows with time series cross-validation

04:20 - 04:34

One option is to restrict the size of the training window. This ensures that only the latest datapoints are used in training. We can control this with the max\_train\_size parameter.

## 14. Non-stationary signals

04:34 - 04:44

Re-visiting our visualization from before, we see that restricting the training window slightly improves the dip in performance in the middle of our validation data.

## 15. Let's practice!

04:44 - 04:53

Now let's prectice using cross-validation to assess the stability of our regression models.

# Stationarity

First, let's confirm what we know about stationarity. Take a look at these time series.

A group of blue lines

Description automatically generated

Which of the following time series do you think are not stationary?

##### Answer the question



C only



D and C



B only

**B and C**

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

## Exercise

# 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.

## NormalInstructionHeader.heading

* The function should loop over the number of bootstraps (given by the parameter n\_boots) and:
  + Take a random sample of the data, with replacement, and calculate the mean of this random sample
  + Compute the percentiles of bootstrap\_means and return it

from sklearn.utils import \_\_\_\_

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 our empty array to fill the results

    bootstrap\_means = np.zeros([n\_boots, data.shape[-1]])

    for ii in range(\_\_\_\_):

        # Generate random indices for our data \*with\* replacement, then take the sample mean

        random\_sample = \_\_\_\_

        bootstrap\_means[ii] = random\_sample.mean(axis=0)

    # Compute the percentiles of choice for the bootstrapped means

    percentiles = \_\_\_\_(bootstrap\_means, percentiles, axis=0)

    return percentiles

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 our 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 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, percentiles, axis=0) return percentiles

**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 our 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 our 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**

**Good job! You can use this function to assess the variability of your model coefficients.**

## Exercise

# 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.

## TabInstructionHeader.heading

* Initialize a TimeSeriesSplit cross-validation object
* Create an array of all zeros to collect the coefficients.
* Iterate through splits of the cross-validation object. On each iteration:
* Fit the model on training data
* Collect the model's coefficients for analysis later

# Iterate through CV splits

n\_splits = 100

cv = TimeSeriesSplit(n\_splits=\_\_\_\_)

# Create empty array to collect coefficients

coefficients = np.\_\_\_\_([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] = \_\_\_\_

# Iterate through CV splits n\_splits = 100 cv = TimeSeriesSplit(n\_splits=100) # 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\_

**# Iterate through CV splits**

**n\_splits = 100**

**cv = TimeSeriesSplit(n\_splits=100)**

**# 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\_**

## Exercise

# 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.

## TabInstructionHeader.heading

Finally, calculate the 95% confidence interval for each coefficient in coefficients using the bootstrap\_interval() function you defined in the previous exercise. You can run bootstrap\_interval? if you want a refresher on the parameters that this function takes.

# Calculate a confidence interval around each coefficient

bootstrapped\_interval = \_\_\_\_

# 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()

# 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()

**# 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()**

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

## Exercise

# 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.

## TabInstructionHeader.heading

* Calculate the cross-validated scores of the model on the data (using a custom scorer we defined for you, my\_pearsonr along with cross\_val\_score).
* Convert the output scores into a pandas Series so that you can treat it as a time series.
* Bootstrap a rolling confidence interval for the mean score using bootstrap\_interval().

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.\_\_\_\_(20).aggregate(partial(\_\_\_\_, percentiles=2.5))

scores\_hi = scores\_series.\_\_\_\_(20).aggregate(partial(\_\_\_\_, percentiles=97.5))

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))

**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))**

## Exercise

# 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.

## TabInstructionHeader.heading

Run the given code to plot the results.

# 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()

**# 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()**

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

## Exercise

# 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.

## TabInstructionHeader.heading

* Create an empty DataFrame to collect the results.
* Iterate through multiple window sizes, each time creating a new TimeSeriesSplit object.

Calculate the cross-validated scores (using a custom scorer we defined for you, my\_pearsonr) # Pre-initialize window sizes

window\_sizes = [25, 50, 75, 100]

# Create an empty DataFrame to collect the stores

all\_scores = \_\_\_\_(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 = \_\_\_\_(n\_splits=100, max\_train\_size=window)

    # Calculate scores across all CV splits and collect them in a DataFrame

    this\_scores = \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_, cv=cv, scoring=my\_pearsonr)

    all\_scores['Length {}'.format(window)] = this\_scores

# 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

# 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

## Exercise

# 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.

## TabInstructionHeader.heading

Run the given code to plot the results.

# 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()

# 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.**

## 1. Wrapping-up

00:00 - 00:10

Now that you've gotten a taste of how time series and machine learning interact with each other, let's quickly recap what we covered in this course, and discuss where you can go next.

## 2. Timeseries and machine learning

00:10 - 00:32

The course started with a discussion of why it is useful to apply machine learning concepts to time series data, with a focus on classifying heartbeat sounds as normal and abnormal, as well as building predictive models of a company's value over time. We also emphasized the importance of visualizing and understanding your raw data.

## 3. Feature extraction and classification

00:32 - 00:54

We then turned to feature extraction and discussed how to collapse a time series array into a single summary statistic and combining many statistics of a time series into a single feature matrix that can be used to classify heartbeat sounds. We also discussed some advanced features that can only be calculated with time series data, such as the spectrogram.

## 4. Model fitting and improving data quality

00:54 - 01:12

We then covered generating model predictions that change over time by discussing a few regression features that are useful for this, such as creating time-shifted versions of an input. We used this in combination with techniques to clean and prepare our data in order to fit the best models.

## 5. Validating and assessing our model performance

01:12 - 01:37

Finally, we covered how to validate machine learning models with time series data. We covered some tips for doing cross-validation with time series data, and introduced a cross-validation iterator that is unique to time series data. We also discussed the importance of assessing if your time series is stationary, and showed a few ways to calculate and visualize your model's stability over subsets of the data.

## 6. Advanced concepts in time series

01:37 - 02:05

If you'd like to learn more about time series data in general, here are some topics worth investigating further. Advanced window functions can improve the "rolling window" statistics we calculated in this course. They're related to advanced signal processing techniques which will help you manipulate your time series data more effectively. Finally, we only barely touched the surface of spectral analysis, which is a complicated and quite interesting field.

## 7. Advanced machine learning

02:05 - 02:44

There are also many more topics to investigate in machine learning for time series. There are many more features that can be extracted for modeling time series. One useful Python package is tsfresh, which can calculate dozens of features from time series. We also only discussed a few specific models in classification and regression, but there is a wide range of other models that are suited for time series predictions. Finally, we only worked with pre-collected data that all fits into the computer's memory, but "real world" applications usually involve more complex processing and machine learning pipelines.

## 8. Ways to practice

02:44 - 03:15

If you'd like to practice your skills in machine learning with time series, there are many options out there for you to experiment. For example, Kaggle has many curated time series datasets with information about how they were collected, and community code snippets for models that have been fit. In addition, Quantopian runs a public service that lets users play around with financial data and build models that attempt to predict future values. It also has a large community of enthusiasts who learn and share their machine learning skills together.

## 9. Let's practice!

03:15 - 03:20

Good luck analyzing time series data

#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()

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.

**Wrap-up** \_\_\_

* Timeseries and machine learning
  + the many applications of time series + machine learning
  + always visualize the data first
  + the scikit-learn API standardizes this process
* Feature extraction and classification
  + summary statistics for time series classification
  + combining multiple features into a single input matrix
  + feature extraction for time series data
* Model fitting and improving data quality
  + time series features for regression
  + generating predictions over time
  + cleaning and improving time series data
* Validating and assessing our model performance
  + cross-validation with time series data (don't shuffle the data!)
  + time series stationarity
  + assessing model coefficient and score stability
* Advanced concepts in time series
  + advanced window functions
  + signal processing and filtering details
  + spectral analysis
* Advanced machine learning
  + advanced time series feature extraction
    - e.g., tsfresh
  + more complex model architectures for regression and classification
  + production-ready pipelines for time series analysis
* Ways to practice
  + Kaggle
  + Quantopian