**Which of these is a classification problem?**

Once you decide to leverage supervised machine learning to solve a new problem, you need to identify whether your problem is better suited to classification or regression. This exercise will help you develop your intuition for distinguishing between the two.

Provided below are 4 example applications of machine learning. Which of them is a supervised classification problem?

**Answer the question**

**50 XP**

**Possible Answers**

Using labeled financial data to predict whether the value of a stock will go up or go down next week.

Using labeled housing price data to predict the price of a new house based on various features.

Using unlabeled data to cluster the students of an online education company into different categories based on their learning styles.

Using labeled financial data to predict what the value of a stock will be next week.

 +50 XP

Exactly! In this example, there are two discrete, qualitative outcomes: the stock market going up, and the stock market going down. This can be represented using a binary variable, and is an application perfectly suited for classification.

**Numerical EDA**

In this chapter, you'll be working with a dataset obtained from the [**UCI Machine Learning Repository**](https://archive.ics.uci.edu/ml/datasets/Congressional+Voting+Records) consisting of votes made by US House of Representatives Congressmen. Your goal will be to predict their party affiliation ('Democrat' or 'Republican') based on how they voted on certain key issues. Here, it's worth noting that we have preprocessed this dataset to deal with missing values. This is so that your focus can be directed towards understanding how to train and evaluate supervised learning models. Once you have mastered these fundamentals, you will be introduced to preprocessing techniques in Chapter 4 and have the chance to apply them there yourself - including on this very same dataset!

Before thinking about what supervised learning models you can apply to this, however, you need to perform Exploratory data analysis (EDA) in order to understand the structure of the data. For a refresher on the importance of EDA, check out the first two chapters of [**Statistical Thinking in Python (Part 1)**](https://www.datacamp.com/courses/statistical-thinking-in-python-part-1).

Get started with your EDA now by exploring this voting records dataset numerically. It has been pre-loaded for you into a DataFrame called df. Use pandas' .head(), .info(), and .describe() methods in the IPython Shell to explore the DataFrame, and select the statement below that is **not** true.

**Instructions**

**50 XP**

**Possible Answers**

* 

The DataFrame has a total of 435 rows and 17 columns.

* 

Except for 'party', all of the columns are of type int64.

* 

The first two rows of the DataFrame consist of votes made by Republicans and the next three rows consist of votes made by Democrats.

* 

There are 17 *predictor variables*, or *features*, in this DataFrame.

* 

The target variable in this DataFrame is 'party'.

Submit Answer

[**Take Hint (-15 XP)**](javascript:void(0))

+50 XP

Great work! The number of columns in the DataFrame is not equal to the number of features. One of the columns - 'party' is the target variable.

**Exercise**

**Exercise**

**Visual EDA**

The Numerical EDA you did in the previous exercise gave you some very important information, such as the names and data types of the columns, and the dimensions of the DataFrame. Following this with some visual EDA will give you an even better understanding of the data. In the video, Hugo used the scatter\_matrix() function on the Iris data for this purpose. However, you may have noticed in the previous exercise that all the features in this dataset are binary; that is, they are either 0 or 1. So a different type of plot would be more useful here, such as [**Seaborn's countplot**](http://seaborn.pydata.org/generated/seaborn.countplot.html).

Given on the right is a countplot of the 'education' bill, generated from the following code:

plt.figure()

sns.countplot(x='education', hue='party', data=df, palette='RdBu')

plt.xticks([0,1], ['No', 'Yes'])

plt.show()

In sns.countplot(), we specify the x-axis data to be 'education', and hue to be 'party'. Recall that 'party' is also our target variable. So the resulting plot shows the difference in voting behavior between the two parties for the 'education' bill, with each party colored differently. We manually specified the color to be 'RdBu', as the Republican party has been traditionally associated with red, and the Democratic party with blue.

It seems like Democrats voted resoundingly *against* this bill, compared to Republicans. This is the kind of information that our machine learning model will seek to learn when we try to predict party affiliation solely based on voting behavior. An expert in U.S politics may be able to predict this without machine learning, but probably not instantaneously - and certainly not if we are dealing with hundreds of samples!

In the IPython Shell, explore the voting behavior further by generating countplots for the 'satellite' and 'missile' bills, and answer the following question: Of these two bills, for which ones do Democrats vote resoundingly in *favor* of, compared to Republicans? Be sure to begin your plotting statements for each figure with plt.figure() so that a new figure will be set up. Otherwise, your plots will be overlayed onto the same figure.

**Instructions**

**50 XP**

**Possible Answers**

* 

'satellite'.

* 

'missile'.

* 

Both 'satellite' and 'missile'.

* 

Neither 'satellite' nor 'missile'.

Submit Answer

[**Take Hint (-15 XP)**](javascript:void(0))

+50 XP

Correct! Democrats voted in favor of both 'satellite' *and* 'missile'

**Incorrect Submission**

Democrats *did* vote in favor of 'satellite', but what about 'missile'?

**Hint**

* Use the code provided in the exercise description to create countplots for 'satellite' and 'missile'. You just have to replace x='education' with either 'satellite' or 'missile'. Observe the resulting plot. Does one party seem to be more in favor of a particular bill compared to the other?

**Exercise**

**Exercise**

**k-Nearest Neighbors: Fit**

Having explored the Congressional voting records dataset, it is time now to build your first classifier. In this exercise, you will fit a k-Nearest Neighbors classifier to the voting dataset, which has once again been pre-loaded for you into a DataFrame df.

In the video, Hugo discussed the importance of ensuring your data adheres to the format required by the scikit-learn API. The features need to be in an array where each column is a feature and each row a different observation or data point - in this case, a Congressman's voting record. The target needs to be a single column with the same number of observations as the feature data. We have done this for you in this exercise. Notice we named the feature array X and response variable y: This is in accordance with the common scikit-learn practice.

Your job is to create an instance of a k-NN classifier with 6 neighbors (by specifying the n\_neighbors parameter) and then fit it to the data. The data has been pre-loaded into a DataFrame called df.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import KNeighborsClassifier from sklearn.neighbors.
* Create arrays X and y for the features and the target variable. Here this has been done for you. Note the use of .drop() to drop the target variable 'party' from the feature array X as well as the use of the .values attribute to ensure X and y are NumPy arrays. Without using .values, X and y are a DataFrame and Series respectively; the scikit-learn API will accept them in this form also as long as they are of the right shape.
* Instantiate a KNeighborsClassifier called knn with 6 neighbors by specifying the n\_neighbors parameter.
* Fit the classifier to the data using the .fit() method.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import KNeighborsClassifier from sklearn.neighbors

from sklearn.neighbors import KNeighborsClassifier as knc

# Create arrays for the features and the response variable

y = df['party'].values

X = df.drop('party', axis=1).values

# Create a k-NN classifier with 6 neighbors

knn = knc(n\_neighbors=6)

# Fit the classifier to the data

knn.fit(X, y)

+100 XP

Excellent! Now that your k-NN classifier with 6 neighbors has been fit to the data, it can be used to predict the labels of new data points.

**Exercise**

**Exercise**

**k-Nearest Neighbors: Predict**

Having fit a k-NN classifier, you can now use it to predict the label of a new data point. However, there is no unlabeled data available since all of it was used to fit the model! You can still use the .predict() method on the X that was used to fit the model, but it is not a good indicator of the model's ability to generalize to new, unseen data.

In the next video, Hugo will discuss a solution to this problem. For now, a random unlabeled data point has been generated and is available to you as X\_new. You will use your classifier to predict the label for this new data point, as well as on the training data X that the model has already seen. Using .predict() on X\_new will generate 1 prediction, while using it on X will generate 435 predictions: 1 for each sample.

The DataFrame has been pre-loaded as df. This time, you will create the feature array X and target variable array y yourself.

**Instructions**

**100 XP**

* Create arrays for the features and the target variable from df. As a reminder, the target variable is 'party'.
* Instantiate a KNeighborsClassifier with 6 neighbors.
* Fit the classifier to the data.
* Predict the labels of the training data, X.
* Predict the label of the new data point X\_new.

[**Take Hint (-30 XP)**](javascript:void(0))

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* To create the target variable y, select the 'party' column of df and access its .values attribute. To create the feature array X, use the .drop() method on df with 'party' and axis=1 as arguments. Then access its .values attribute.
* To instantiate the classifier, use KNeighborsClassifier and specify the number of neighbors using the n\_neighbors parameter.
* Use the .fit() method with X and y as arguments to fit the classifier to the data.
* Use the .predict() method on X to predict the labels of the training data.
* Use the .predict() method on X\_new to predict the label of the new data point.

 Awesome, thanks for your feedback!

# Import KNeighborsClassifier from sklearn.neighbors

from sklearn.neighbors import KNeighborsClassifier as knc

# Create arrays for the features and the response variable

y = df.party.values

X = df.drop('party', axis=1).values

# Create a k-NN classifier with 6 neighbors: knn

knn = knc(6)

# Fit the classifier to the data

print(knn.fit(X, y))

# Predict the labels for the training data X

y\_pred = knn.predict(X)

# Predict and print the label for the new data point X\_new

new\_prediction = knn.predict(X\_new)

print("Prediction: {}".format(new\_prediction))

**Incorrect Submission**

Did you call KNeighborsClassifier()?

Did you call knn.fit()?

Did you call knn.predict()?

In [1]: X\_new

Out[1]:

0 1 2 3 4 5 6 \

0 0.696469 0.286139 0.226851 0.551315 0.719469 0.423106 0.980764

7 8 9 10 11 12 13 \

0 0.68483 0.480932 0.392118 0.343178 0.72905 0.438572 0.059678

14 15

0 0.398044 0.737995

Traceback (most recent call last):

File "script.py", line 9, in <module>

knn = knc(6)

NameError: name 'knc' is not defined

Traceback (most recent call last):

File "script.py", line 12, in <module>

print(knc.fit(X, y))

TypeError: fit() missing 1 required positional argument: 'y'

Traceback (most recent call last):

File "script.py", line 12, in <module>

(knc.fit(X, y))

TypeError: fit() missing 1 required positional argument: 'y'

<script.py> output:

Traceback (most recent call last):

File "script.py", line 13, in <module>

(knc.fit(X, y))

TypeError: fit() missing 1 required positional argument: 'y'

<script.py> output:

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=6, p=2,

weights='uniform')

Traceback (most recent call last):

File "script.py", line 15, in <module>

y\_pred = knc.predict(X)

TypeError: predict() missing 1 required positional argument: 'X'

<script.py> output:

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=6, p=2,

weights='uniform')

Traceback (most recent call last):

File "script.py", line 18, in <module>

new\_prediction = knc.predict(X\_new)

TypeError: predict() missing 1 required positional argument: 'X'

<script.py> output:

KNeighborsClassifier(algorithm='auto', leaf\_size=30, metric='minkowski',

metric\_params=None, n\_jobs=1, n\_neighbors=6, p=2,

weights='uniform')

Prediction: ['democrat']

In [2]:

+70 XP

Great work! Did your model predict 'democrat' or 'republican'? How sure can you be of its predictions? In other words, how can you measure its performance? This is what you will learn in the next video.

**Exercise**

**Exercise**

**The digits recognition dataset**

Up until now, you have been performing binary classification, since the target variable had two possible outcomes. Hugo, however, got to perform multi-class classification in the videos, where the target variable could take on three possible outcomes. Why does he get to have all the fun?! In the following exercises, you'll be working with the [**MNIST**](http://yann.lecun.com/exdb/mnist/) digits recognition dataset, which has 10 classes, the digits 0 through 9! A reduced version of the MNIST dataset is one of scikit-learn's included datasets, and that is the one we will use in this exercise.

Each sample in this scikit-learn dataset is an 8x8 image representing a handwritten digit. Each pixel is represented by an integer in the range 0 to 16, indicating varying levels of black. Recall that scikit-learn's built-in datasets are of type Bunch, which are dictionary-like objects. Helpfully for the MNIST dataset, scikit-learn provides an 'images' key in addition to the 'data' and 'target' keys that you have seen with the Iris data. Because it is a 2D array of the images corresponding to each sample, this 'images' key is useful for visualizing the images, as you'll see in this exercise (for more on plotting 2D arrays, see [**Chapter 2**](https://www.datacamp.com/courses/introduction-to-data-visualization-with-python) of DataCamp's course on Data Visualization with Python). On the other hand, the 'data' key contains the feature array - that is, the images as a flattened array of 64 pixels.

Notice that you can access the keys of these Bunch objects in two different ways: By using the . notation, as in digits.images, or the [] notation, as in digits['images'].

For more on the MNIST data, check out [**this exercise**](https://campus.datacamp.com/courses/importing-data-in-python-part-1/introduction-and-flat-files-1?ex=10) in Part 1 of DataCamp's Importing Data in Python course. There, the full version of the MNIST dataset is used, in which the images are 28x28. It is a famous dataset in machine learning and computer vision, and frequently used as a benchmark to evaluate the performance of a new model.

**Instructions**

**100 XP**

* Import datasets from sklearn and matplotlib.pyplot as plt.
* Load the digits dataset using the .load\_digits() method on datasets.
* Print the keys and DESCR of digits.
* Print the shape of images and data keys using the . notation.
* Display the 1011th image using plt.imshow(). This has been done for you, so hit 'Submit Answer' to see which handwritten digit this happens to be!

[**Take Hint (-30 XP)**](javascript:void(0))

# Import necessary modules

from sklearn import datasets

import matplotlib.pyplot as plt

# Load the digits dataset: digits

digits = datasets.load\_digits()

# Print the keys and DESCR of the dataset

print(digits.keys())

print(digits.DESCR)

# Print the shape of the images and data keys

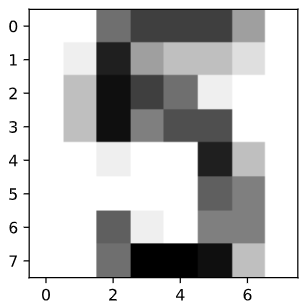
print(digits.images.shape)

print(digits.data.shape)

# Display digit 1010

plt.imshow(digits.images[1010], cmap=plt.cm.gray\_r, interpolation='nearest')

plt.show()



<script.py> output:

<built-in method keys of Bunch object at 0x7efd5e8485c8>

Optical Recognition of Handwritten Digits Data Set

===================================================

Notes

-----

Data Set Characteristics:

:Number of Instances: 5620

:Number of Attributes: 64

:Attribute Information: 8x8 image of integer pixels in the range 0..16.

:Missing Attribute Values: None

:Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)

:Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets

http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where

each class refers to a digit.

Preprocessing programs made available by NIST were used to extract

normalized bitmaps of handwritten digits from a preprinted form. From a

total of 43 people, 30 contributed to the training set and different 13

to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of

4x4 and the number of on pixels are counted in each block. This generates

an input matrix of 8x8 where each element is an integer in the range

0..16. This reduces dimensionality and gives invariance to small

distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.

T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.

L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,

1994.

References

----------

- C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their

Applications to Handwritten Digit Recognition, MSc Thesis, Institute of

Graduate Studies in Science and Engineering, Bogazici University.

- E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.

- Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.

Linear dimensionalityreduction using relevance weighted LDA. School of

Electrical and Electronic Engineering Nanyang Technological University.

2005.

- Claudio Gentile. A New Approximate Maximal Margin Classification

Algorithm. NIPS. 2000.

(1797, 8, 8)

(1797, 64)

<script.py> output:

dict\_keys(['data', 'target', 'target\_names', 'images', 'DESCR'])

Optical Recognition of Handwritten Digits Data Set

===================================================

Notes

-----

Data Set Characteristics:

:Number of Instances: 5620

:Number of Attributes: 64

:Attribute Information: 8x8 image of integer pixels in the range 0..16.

:Missing Attribute Values: None

:Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)

:Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets

http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where

each class refers to a digit.

Preprocessing programs made available by NIST were used to extract

normalized bitmaps of handwritten digits from a preprinted form. From a

total of 43 people, 30 contributed to the training set and different 13

to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of

4x4 and the number of on pixels are counted in each block. This generates

an input matrix of 8x8 where each element is an integer in the range

0..16. This reduces dimensionality and gives invariance to small

distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.

T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.

L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,

1994.

References

----------

- C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their

Applications to Handwritten Digit Recognition, MSc Thesis, Institute of

Graduate Studies in Science and Engineering, Bogazici University.

- E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.

- Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.

Linear dimensionalityreduction using relevance weighted LDA. School of

Electrical and Electronic Engineering Nanyang Technological University.

2005.

- Claudio Gentile. A New Approximate Maximal Margin Classification

Algorithm. NIPS. 2000.

(1797, 8, 8)

(1797, 64)

In [1]:

**Incorrect Submission**

Did you correctly print the keys of digits? using digits.keys()?

+100 XP

Good job! It looks like the image in question corresponds to the digit '5'. Now, can you build a classifier that can make this prediction not only for this image, but for all the other ones in the dataset? You'll do so in the next exercise!

**Exercise**

**Exercise**

**Train/Test Split + Fit/Predict/Accuracy**

Now that you have learned about the importance of splitting your data into training and test sets, it's time to practice doing this on the digits dataset! After creating arrays for the features and target variable, you will split them into training and test sets, fit a k-NN classifier to the training data, and then compute its accuracy using the .score() method.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import KNeighborsClassifier from sklearn.neighbors and train\_test\_split from sklearn.model\_selection.
* Create an array for the features using digits.data and an array for the target using digits.target.
* Create stratified training and test sets using 0.2 for the size of the test set. Use a random state of 42. Stratify the split according to the labels so that they are distributed in the training and test sets as they are in the original dataset.
* Create a k-NN classifier with 7 neighbors and fit it to the training data.
* Compute and print the accuracy of the classifier's predictions using the .score() method.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import necessary modules

from sklearn.neighbors import KNeighborsClassifier as knc

from sklearn.model\_selection import train\_test\_split as tts

# Create feature and target arrays

X, y = digits.data, digits.target

# Split into training and test set

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size = 0.2, random\_state=42, stratify=y)

# Create a k-NN classifier with 7 neighbors: knn

knn = knc(n\_neighbors=7)

# Fit the classifier to the training data

knn.fit(X\_train, y\_train)

# Print the accuracy

print(knn.score(X\_test, y\_test))

**Incorrect Submission**

Have you specified the arguments for knn.fit() using the right syntax?

Traceback (most recent call last):

File "script.py", line 15, in <module>

knn.fit(X\_train)

TypeError: fit() missing 1 required positional argument: 'y'

<script.py> output:

0.9833333333333333

In [1]:

+100 XP

Excellent work! Incredibly, this out of the box k-NN classifier with 7 neighbors has learned from the training data and predicted the labels of the images in the test set with 98% accuracy, and it did so in less than a second! This is one illustration of how incredibly useful machine learning techniques can be.

**Exercise**

**Exercise**

**Overfitting and underfitting**

Remember the model complexity curve that Hugo showed in the video? You will now construct such a curve for the digits dataset! In this exercise, you will compute and plot the training and testing accuracy scores for a variety of different neighbor values. By observing how the accuracy scores differ for the training and testing sets with different values of k, you will develop your intuition for overfitting and underfitting.

The training and testing sets are available to you in the workspace as X\_train, X\_test, y\_train, y\_test. In addition, KNeighborsClassifier has been imported from sklearn.neighbors.

**Instructions**

**100 XP**

* Inside the for loop:
  + Setup a k-NN classifier with the number of neighbors equal to k.
  + Fit the classifier with k neighbors to the training data.
  + Compute accuracy scores the training set and test set separately using the .score() method and assign the results to the train\_accuracy and test\_accuracy arrays respectively.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check the first for loop. Did you correctly specify the body? Did you call KNeighborsClassifier()?

knc = KNeighborsClassifier

# Setup arrays to store train and test accuracies

neighbors = np.arange(1, 9)

train\_accuracy = np.empty(len(neighbors))

test\_accuracy = np.empty(len(neighbors))

# Loop over different values of k

for i, k in enumerate(neighbors):

# Setup a k-NN Classifier with k neighbors: knn

knn = KNeighborsClassifier(k)

# Fit the classifier to the training data

knn.fit(X\_train, y\_train)

#Compute accuracy on the training set

train\_accuracy[i] = knn.score(X\_train, y\_train)

#Compute accuracy on the testing set

test\_accuracy[i] = knn.score(X\_test, y\_test)

# Generate plot

plt.title('k-NN: Varying Number of Neighbors')

plt.plot(neighbors, test\_accuracy, label = 'Testing Accuracy')

plt.plot(neighbors, train\_accuracy, label = 'Training Accuracy')

plt.legend()

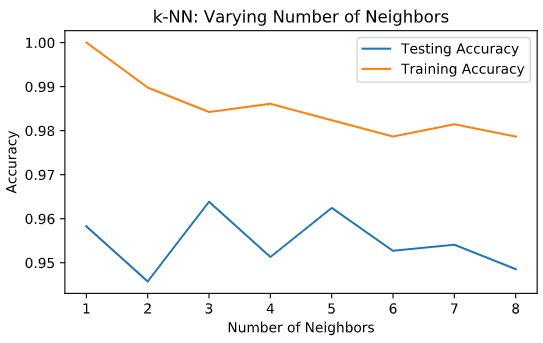
plt.xlabel('Number of Neighbors')

plt.ylabel('Accuracy')

plt.show()

+100 XP

Great work! It looks like the test accuracy is highest when using 3 and 5 neighbors. Using 8 neighbors or more seems to result in a simple model that underfits the data. Now that you've grasped the fundamentals of classification, you will learn about regression in the next chapter!



**Which of the following is a regression problem?**

Andy introduced regression to you using the Boston housing dataset. But regression models can be used in a variety of contexts to solve a variety of different problems.

Given below are four example applications of machine learning. Your job is to pick the one that is *best* framed as a **regression** problem.

**Answer the question**

**50 XP**

**Possible Answers**

An e-commerce company using labeled customer data to predict whether or not a customer will purchase a particular item.

press1

A healthcare company using data about cancer tumors (such as their geometric measurements) to predict whether a new tumor is benign or malignant.

press2

A restaurant using review data to ascribe positive or negative sentiment to a given review.

press3

A bike share company using time and weather data to predict the number of bikes being rented at any given hour.

press4

Submit Answer

Take Hint (-15xp)

**Incorrect Submission**

Incorrect. There are only two outcomes here: Either the customer will purchase the item, or they will not. This is a classification task.

+50 XP

Great work! The target variable here - the number of bike rentals at any given hour - is quantitative, so this is best framed as a regression problem.

**Exercise**

**Exercise**

**Importing data for supervised learning**

In this chapter, you will work with [**Gapminder**](https://www.gapminder.org/data/) data that we have consolidated into one CSV file available in the workspace as 'gapminder.csv'. Specifically, your goal will be to use this data to predict the life expectancy in a given country based on features such as the country's GDP, fertility rate, and population. As in Chapter 1, the dataset has been preprocessed.

Since the target variable here is quantitative, this is a regression problem. To begin, you will fit a linear regression with just one feature: 'fertility', which is the average number of children a woman in a given country gives birth to. In later exercises, you will use all the features to build regression models.

Before that, however, you need to import the data and get it into the form needed by scikit-learn. This involves creating feature and target variable arrays. Furthermore, since you are going to use only one feature to begin with, you need to do some reshaping using NumPy's .reshape() method. Don't worry too much about this reshaping right now, but it is something you will have to do occasionally when working with scikit-learn so it is useful to practice.

**Instructions**

**100 XP**

* Import numpy and pandas as their standard aliases.
* Read the file 'gapminder.csv' into a DataFrame df using the read\_csv() function.
* Create array X for the 'fertility' feature and array y for the 'life' target variable.
* Reshape the arrays by using the .reshape() method and passing in -1 and 1.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import numpy and pandas

import numpy as np

import pandas as pd

# Read the CSV file into a DataFrame: df

df = pd.read\_csv('gapminder.csv')

# Create arrays for features and target variable

y, X = df.life, df.fertility

# Print the dimensions of X and y before reshaping

print("Dimensions of y before reshaping: {}".format(y.shape))

print("Dimensions of X before reshaping: {}".format(X.shape))

# Reshape X and y

y = \_\_\_\_

X = \_\_\_\_

# Print the dimensions of X and y after reshaping

print("Dimensions of y after reshaping: {}".format(y.shape))

print("Dimensions of X after reshaping: {}".format(X.shape))

**Incorrect Submission**

Did you correctly create the array y?

Did you define the pandas DataFrame df without errors?

+70 XP

Great work! Notice the differences in shape before and after applying the .reshape() method. Getting the feature and target variable arrays into the right format for scikit-learn is an important precursor to model building.

# Import numpy and pandas

import numpy as np

import pandas as pd

# Read the CSV file into a DataFrame: df

df = pd.read\_csv('gapminder.csv')

# Create arrays for features and target variable

y = df.life.values

X = df.fertility.values

# Print the dimensions of X and y before reshaping

print("Dimensions of y before reshaping: {}".format(y.shape))

print("Dimensions of X before reshaping: {}".format(X.shape))

# Reshape X and y

y.shape, X.shape = (-1, 1), (-1, 1)

# Print the dimensions of X and y after reshaping

print("Dimensions of y after reshaping: {}".format(y.shape))

print("Dimensions of X after reshaping: {}".format(X.shape))

<script.py> output:

Dimensions of y before reshaping: (139,)

Dimensions of X before reshaping: (139,)

Dimensions of y after reshaping: (139, 1)

Dimensions of X after reshaping: (139, 1)

In [1]: y.shape = (-1, 1)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

y.shape = (-1, 1)

NameError: name 'y' is not defined

In [2]: # Import numpy and pandas

import numpy as np

import pandas as pd

# Read the CSV file into a DataFrame: df

df = pd.read\_csv('gapminder.csv')

# Create arrays for features and target variable

y = df.life.values

X = df.fertility.values

# Print the dimensions of X and y before reshaping

print("Dimensions of y before reshaping: {}".format(y.shape))

print("Dimensions of X before reshaping: {}".format(X.shape))

# Reshape X and y

y = y.reshape(-1, 1)

X = X.reshape(-1, 1)

# Print the dimensions of X and y after reshaping

print("Dimensions of y after reshaping: {}".format(y.shape))

print("Dimensions of X after reshaping: {}".format(X.shape))

Dimensions of y before reshaping: (139,)

Dimensions of X before reshaping: (139,)

Dimensions of y after reshaping: (139, 1)

Dimensions of X after reshaping: (139, 1)

In [3]: y.shape = (-1, 1)

<script.py> output:

Dimensions of y before reshaping: (139,)

Dimensions of X before reshaping: (139,)

Dimensions of y after reshaping: (139, 1)

Dimensions of X after reshaping: (139, 1)

In [4]: y.shape

Out[4]: (139, 1)

In [5]:

**Exercise**

**Exercise**

**Exploring the Gapminder data**

As always, it is important to explore your data before building models. On the right, we have constructed a heatmap showing the correlation between the different features of the Gapminder dataset, which has been pre-loaded into a DataFrame as df and is available for exploration in the IPython Shell. Cells that are in green show positive correlation, while cells that are in red show negative correlation. Take a moment to explore this: Which features are positively correlated with life, and which ones are negatively correlated? Does this match your intuition?

Then, in the IPython Shell, explore the DataFrame using pandas methods such as .info(), .describe(), .head().

In case you are curious, the heatmap was generated using [**Seaborn's heatmap function**](http://seaborn.pydata.org/generated/seaborn.heatmap.html) and the following line of code, where df.corr() computes the pairwise correlation between columns:

sns.heatmap(df.corr(), square=True, cmap='RdYlGn')

Once you have a feel for the data, consider the statements below and select the one that is **not** true. After this, Hugo will explain the mechanics of linear regression in the next video and you will be on your way building regression models!

**Instructions**

**50 XP**

**Possible Answers**

* 

The DataFrame has 139 samples (or rows) and 9 columns.

* 

life and fertility are negatively correlated.

* 

The mean of life is 69.602878.

* 

fertility is of type int64.

* 

GDP and life are positively correlated.

Submit Answer

[**Take Hint (-15 XP)**](javascript:void(0))

In [1]: life

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

life

NameError: name 'life' is not defined

In [2]: life

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

life

NameError: name 'life' is not defined

In [3]: mean(life)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

mean(life)

NameError: name 'mean' is not defined

In [4]: np

Out[4]: <module 'numpy' from '/usr/local/lib/python3.6/dist-packages/numpy/\_\_init\_\_.py'>

In [5]: np.mean(df.life.values)

Out[5]: 69.60287769784172

In [6]: df.fertility.values.dtype

Out[6]: dtype('float64')

In [7]:

+50 XP

Good job! As seen by using df.info(), fertility, along with all the other columns, is of type float64, **not** int64.

**Exercise**

**Exercise**

**Fit & predict for regression**

Now, you will fit a linear regression and predict life expectancy using just one feature. You saw Andy do this earlier using the 'RM' feature of the Boston housing dataset. In this exercise, you will use the 'fertility' feature of the Gapminder dataset. Since the goal is to predict life expectancy, the target variable here is 'life'. The array for the target variable has been pre-loaded as y and the array for 'fertility' has been pre-loaded as X\_fertility.

A scatter plot with 'fertility' on the x-axis and 'life' on the y-axis has been generated. As you can see, there is a strongly negative correlation, so a linear regression should be able to capture this trend. Your job is to fit a linear regression and then predict the life expectancy, overlaying these predicted values on the plot to generate a regression line. You will also compute and print the R2R2 score using sckit-learn's .score() method.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import LinearRegression from sklearn.linear\_model.
* Create a LinearRegression regressor called reg.
* Set up the prediction space to range from the minimum to the maximum of X\_fertility. This has been done for you.
* Fit the regressor to the data (X\_fertility and y) and compute its predictions using the .predict() method and the prediction\_space array.
* Compute and print the R2R2 score using the .score() method.
* Overlay the plot with your linear regression line. This has been done for you, so hit 'Submit Answer' to see the result!

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you used print(reg.score(X\_fertility, y)) to do the appropriate printouts?

Traceback (most recent call last):

File "script.py", line 17, in <module>

print(reg.score(y, y\_pred))

File "script.py", line 387, in score

multioutput='variance\_weighted')

File "script.py", line 530, in r2\_score

y\_true, y\_pred, multioutput)

File "script.py", line 75, in \_check\_reg\_targets

check\_consistent\_length(y\_true, y\_pred)

File "script.py", line 204, in check\_consistent\_length

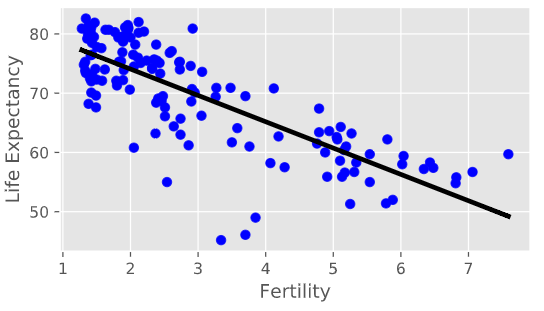
" samples: %r" % [int(l) for l in lengths])

ValueError: Found input variables with inconsistent numbers of samples: [50, 139]

<script.py> output:

0.6192442167740035

In [1]:



+100 XP

Fantastic! Notice how the line captures the underlying trend in the data. And the performance is quite decent for this basic regression model with only one feature!

**Exercise**

**Exercise**

**Train/test split for regression**

As you learned in Chapter 1, train and test sets are vital to ensure that your supervised learning model is able to generalize well to new data. This was true for classification models, and is equally true for linear regression models.

In this exercise, you will split the Gapminder dataset into training and testing sets, and then fit and predict a linear regression over **all** features. In addition to computing the R2R2 score, you will also compute the Root Mean Squared Error (RMSE), which is another commonly used metric to evaluate regression models. The feature array X and target variable array y have been pre-loaded for you from the DataFrame df.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import LinearRegression from sklearn.linear\_model, mean\_squared\_error from sklearn.metrics, and train\_test\_split from sklearn.model\_selection.
* Using X and y, create training and test sets such that 30% is used for testing and 70% for training. Use a random state of 42.
* Create a linear regression regressor called reg\_all, fit it to the training set, and evaluate it on the test set.
* Compute and print the R2R2 score using the .score() method on the test set.
* Compute and print the RMSE. To do this, first compute the Mean Squared Error using the mean\_squared\_error() function with the arguments y\_test and y\_pred, and then take its square root using np.sqrt().

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of train\_test\_split(). Did you correctly specify the argument random\_state? Expected 42, but got 0.7.

Check your call of reg\_all.fit(). Did you correctly specify the first argument? Expected something different.

# Import necessary modules

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

# Create training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = .3, random\_state=42)

# Create the regressor: reg\_all

reg\_all = LinearRegression()

# Fit the regressor to the training data

reg\_all.fit(X\_train, y\_train)

# Predict on the test data: y\_pred

y\_pred = reg\_all.predict(X\_test)

# Compute and print R^2 and RMSE

print("R^2: {}".format(reg\_all.score(X\_test, y\_test)))

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))

print("Root Mean Squared Error: {}".format(rmse))

Traceback (most recent call last):

File "script.py", line 7, in <module>

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = .3, random\_state=.7)

File "script.py", line 2056, in train\_test\_split

train, test = next(cv.split(X=arrays[0], y=stratify))

File "script.py", line 1204, in split

for train, test in self.\_iter\_indices(X, y, groups):

File "script.py", line 1305, in \_iter\_indices

rng = check\_random\_state(self.random\_state)

File "script.py", line 635, in check\_random\_state

' instance' % seed)

ValueError: 0.7 cannot be used to seed a numpy.random.RandomState instance

<script.py> output:

R^2: 0.8816244121829089

Root Mean Squared Error: 2.776508981964029

<script.py> output:

R^2: 0.838046873142936

Root Mean Squared Error: 3.2476010800377213

In [1]:

+100 XP

Excellent! Using all features has improved the model score. This makes sense, as the model has more information to learn from. However, there is one potential pitfall to this process. Can you spot it? You'll learn about this as well how to better validate your models in the next video!

**Exercise**

**Exercise**

**5-fold cross-validation**

Cross-validation is a vital step in evaluating a model. It maximizes the amount of data that is used to train the model, as during the course of training, the model is not only trained, but also tested on all of the available data.

In this exercise, you will practice 5-fold cross validation on the Gapminder data. By default, scikit-learn's cross\_val\_score() function uses R2R2 as the metric of choice for regression. Since you are performing 5-fold cross-validation, the function will return 5 scores. Your job is to compute these 5 scores and then take their average.

The DataFrame has been loaded as df and split into the feature/target variable arrays X and y. The modules pandas and numpy have been imported as pd and np, respectively.

**Instructions**

**100 XP**

* Import LinearRegression from sklearn.linear\_model and cross\_val\_score from sklearn.model\_selection.
* Create a linear regression regressor called reg.
* Use the cross\_val\_score() function to perform 5-fold cross-validation on X and y.
* Compute and print the average cross-validation score. You can use NumPy's mean() function to compute the average.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import the necessary modules

from sklearn.linear\_model import LinearRegression as lr

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

# Create a linear regression object: reg

reg = lr()

# Compute 5-fold cross-validation scores: cv\_scores

cv\_scores = cvs(reg, X, y, cv=5)

# Print the 5-fold cross-validation scores

print(cv\_scores)

print("Average 5-Fold CV Score: {}".format(np.mean(cv\_scores)))

**Incorrect Submission**

Your code can not be executed due to a syntax error:  
invalid syntax (script.py, line 3).

Check your call of cvs(). Did you correctly specify the first argument? Expected reg, but got X.

Have you specified the arguments for cvs() using the right syntax?

Did you define the variable cv\_scores without errors?

# Import the necessary modules

from sklearn.linear\_model import LinearRegression as lr

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

# Create a linear regression object: reg

reg = lr()

# Compute 5-fold cross-validation scores: cv\_scores

cv\_scores = cvs(reg, X, y, cv=5)

# Print the 5-fold cross-validation scores

print(cv\_scores)

print("Average 5-Fold CV Score: {}".format(np.mean(cv\_scores)))

**Hint**

* Use the command from y import x to import x from y.
* Use the LinearRegression() function to create the regressor.
* Inside cross\_val\_score(), specify the arguments reg, X, y, and cv=5.
* Pass cv\_scores to the first function, and np.mean(cv\_scores) to the format() function of the second print() function.

In [1]: np

Out[1]: <module 'numpy' from '/usr/local/lib/python3.6/dist-packages/numpy/\_\_init\_\_.py'>

File "script.py", line 3

\_from sklearn.model\_selection import cross\_val\_score as cvs

^

SyntaxError: invalid syntax

Traceback (most recent call last):

File "script.py", line 9, in <module>

cv\_scores = cvs(X, y)

File "script.py", line 335, in cross\_val\_score

scorer = check\_scoring(estimator, scoring=scoring)

File "script.py", line 274, in check\_scoring

"'fit' method, %r was passed" % estimator)

TypeError: estimator should be an estimator implementing 'fit' method, array([[3.4811059e+07, 2.7300000e+00, 1.0000000e-01, ..., 1.2314000e+04,

1.2990490e+02, 2.9500000e+01],

[1.9842251e+07, 6.4300000e+00, 2.0000000e+00, ..., 7.1030000e+03,

1.3012470e+02, 1.9200000e+02],

[4.0381860e+07, 2.2400000e+00, 5.0000000e-01, ..., 1.4646000e+04,

1.1889150e+02, 1.5400000e+01],

...,

[8.6589342e+07, 1.8600000e+00, 4.0000000e-01, ..., 4.0850000e+03,

1.2193670e+02, 2.6200000e+01],

[1.3114579e+07, 5.8800000e+00, 1.3600000e+01, ..., 3.0390000e+03,

1.3244930e+02, 9.4900000e+01],

[1.3495462e+07, 3.8500000e+00, 1.5100000e+01, ..., 1.2860000e+03,

1.3197450e+02, 9.8300000e+01]]) was passed

Traceback (most recent call last):

File "script.py", line 9, in <module>

cv\_scores = cvs(reg)

TypeError: cross\_val\_score() missing 1 required positional argument: 'X'

Traceback (most recent call last):

File "script.py", line 9, in <module>

cv\_scores = cvs(reg, X, y, cv=5)

File "script.py", line 342, in cross\_val\_score

pre\_dispatch=pre\_dispatch)

File "script.py", line 206, in cross\_validate

for train, test in cv.split(X, y, groups))

File "script.py", line 779, in \_\_call\_\_

while self.dispatch\_one\_batch(iterator):

File "script.py", line 625, in dispatch\_one\_batch

self.\_dispatch(tasks)

File "script.py", line 588, in \_dispatch

job = self.\_backend.apply\_async(batch, callback=cb)

File "script.py", line 111, in apply\_async

result = ImmediateResult(func)

File "script.py", line 332, in \_\_init\_\_

self.results = batch()

File "script.py", line 131, in \_\_call\_\_

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 131, in <listcomp>

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 458, in \_fit\_and\_score

estimator.fit(X\_train, y\_train, \*\*fit\_params)

File "script.py", line 489, in fit

copy=self.copy\_X, sample\_weight=sample\_weight)

File "script.py", line 171, in \_preprocess\_data

if fit\_intercept:

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

Traceback (most recent call last):

File "script.py", line 9, in <module>

cv\_scores = cvs(reg, X, y, cv=5)

File "script.py", line 342, in cross\_val\_score

pre\_dispatch=pre\_dispatch)

File "script.py", line 206, in cross\_validate

for train, test in cv.split(X, y, groups))

File "script.py", line 779, in \_\_call\_\_

while self.dispatch\_one\_batch(iterator):

File "script.py", line 625, in dispatch\_one\_batch

self.\_dispatch(tasks)

File "script.py", line 588, in \_dispatch

job = self.\_backend.apply\_async(batch, callback=cb)

File "script.py", line 111, in apply\_async

result = ImmediateResult(func)

File "script.py", line 332, in \_\_init\_\_

self.results = batch()

File "script.py", line 131, in \_\_call\_\_

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 131, in <listcomp>

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 458, in \_fit\_and\_score

estimator.fit(X\_train, y\_train, \*\*fit\_params)

File "script.py", line 489, in fit

copy=self.copy\_X, sample\_weight=sample\_weight)

File "script.py", line 171, in \_preprocess\_data

if fit\_intercept:

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

Traceback (most recent call last):

File "script.py", line 9, in <module>

cv\_scores = LinearRegression(reg, X, y, cv=5)

TypeError: \_\_init\_\_() got an unexpected keyword argument 'cv'

Traceback (most recent call last):

File "script.py", line 9, in <module>

cv\_scores = cross\_val\_score(reg, X, y, cv=5)

File "script.py", line 342, in cross\_val\_score

pre\_dispatch=pre\_dispatch)

File "script.py", line 206, in cross\_validate

for train, test in cv.split(X, y, groups))

File "script.py", line 779, in \_\_call\_\_

while self.dispatch\_one\_batch(iterator):

File "script.py", line 625, in dispatch\_one\_batch

self.\_dispatch(tasks)

File "script.py", line 588, in \_dispatch

job = self.\_backend.apply\_async(batch, callback=cb)

File "script.py", line 111, in apply\_async

result = ImmediateResult(func)

File "script.py", line 332, in \_\_init\_\_

self.results = batch()

File "script.py", line 131, in \_\_call\_\_

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 131, in <listcomp>

return [func(\*args, \*\*kwargs) for func, args, kwargs in self.items]

File "script.py", line 458, in \_fit\_and\_score

estimator.fit(X\_train, y\_train, \*\*fit\_params)

File "script.py", line 489, in fit

copy=self.copy\_X, sample\_weight=sample\_weight)

File "script.py", line 171, in \_preprocess\_data

if fit\_intercept:

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

Traceback (most recent call last):

File "script.py", line 9, in <module>

cv\_scores = cvs(reg, X, y, 5)

File "script.py", line 342, in cross\_val\_score

pre\_dispatch=pre\_dispatch)

File "script.py", line 192, in cross\_validate

X, y, groups = indexable(X, y, groups)

File "script.py", line 229, in indexable

check\_consistent\_length(\*result)

File "script.py", line 200, in check\_consistent\_length

lengths = [\_num\_samples(X) for X in arrays if X is not None]

File "script.py", line 200, in <listcomp>

lengths = [\_num\_samples(X) for X in arrays if X is not None]

File "script.py", line 119, in \_num\_samples

" a valid collection." % x)

TypeError: Singleton array array(5) cannot be considered a valid collection.

<script.py> output:

[0.81720569 0.82917058 0.90214134 0.80633989 0.94495637]

Average 5-Fold CV Score: 0.8599627722793232

In [2]:

+70 XP

Great work! Now that you have cross-validated your model, you can more confidently evaluate its predictions.

**Exercise**

**Exercise**

**K-Fold CV comparison**

Cross validation is essential but do not forget that the more folds you use, the more computationally expensive cross-validation becomes. In this exercise, you will explore this for yourself. Your job is to perform 3-fold cross-validation and then 10-fold cross-validation on the Gapminder dataset.

In the IPython Shell, you can use %timeit to see how long each 3-fold CV takes compared to 10-fold CV by executing the following cv=3 and cv=10:

%timeit cross\_val\_score(reg, X, y, cv = \_\_\_\_)

pandas and numpy are available in the workspace as pd and np. The DataFrame has been loaded as df and the feature/target variable arrays X and y have been created.

**Instructions**

**100 XP**

* Import LinearRegression from sklearn.linear\_model and cross\_val\_score from sklearn.model\_selection.
* Create a linear regression regressor called reg.
* Perform 3-fold CV and then 10-fold CV. Compare the resulting mean scores.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import necessary modules

from sklearn.linear\_model import LinearRegression as lr

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

# Create a linear regression object: reg

reg = lr()

# Perform 3-fold CV

cvscores\_3 = cvs(reg, X, y, cv=3)

print(np.mean(cvscores\_3))

# Perform 10-fold CV

cvscores\_10 = cvs(reg, X, y, cv=10)

print(np.mean(cvscores\_10))

<script.py> output:

0.8718712782622108

0.8436128620131201

In [1]:

+100 XP

Excellent! Did you use %timeit in the IPython Shell to see how much longer it takes 10-fold cross-validation to run compared to 3-fold cross-validation?

**Exercise**

**Exercise**

**Regularization I: Lasso**

In the video, you saw how Lasso selected out the 'RM' feature as being the most important for predicting Boston house prices, while shrinking the coefficients of certain other features to 0. Its ability to perform feature selection in this way becomes even more useful when you are dealing with data involving thousands of features.

In this exercise, you will fit a lasso regression to the Gapminder data you have been working with and plot the coefficients. Just as with the Boston data, you will find that the coefficients of some features are shrunk to 0, with only the most important ones remaining.

The feature and target variable arrays have been pre-loaded as X and y.

**Instructions**

**100 XP**

* Import Lasso from sklearn.linear\_model.
* Instantiate a Lasso regressor with an alpha of 0.4 and specify normalize=True.
* Fit the regressor to the data and compute the coefficients using the coef\_ attribute.
* Plot the coefficients on the y-axis and column names on the x-axis. This has been done for you, so hit 'Submit Answer' to view the plot!

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you define the variable lasso\_coef without errors?

# Import Lasso

from sklearn.linear\_model import Lasso

# Instantiate a lasso regressor: lasso

lasso = Lasso(alpha=0.4, normalize=1)

# Fit the regressor to the data

lasso.fit(X, y)

# Compute and print the coefficients

lasso\_coef = lasso.coef\_

print(lasso\_coef)

# Plot the coefficients

plt.plot(range(len(df\_columns)), lasso\_coef)

plt.xticks(range(len(df\_columns)), df\_columns.values, rotation=60)

plt.margins(0.02)

plt.show()

Traceback (most recent call last):

File "script.py", line 11, in <module>

lasso\_coef = \_\_\_\_

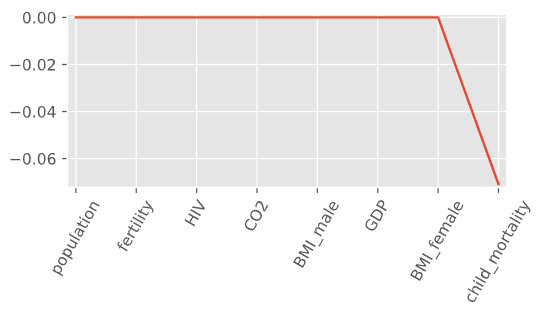
NameError: name '\_\_\_\_' is not defined

<script.py> output:

[-0. -0. -0. 0. 0. 0.

-0. -0.07087587]

In [1]:



+100 XP

Great work! According to the lasso algorithm, it seems like 'child\_mortality' is the most important feature when predicting life expectancy.

**Exercise**

**Exercise**

**Regularization II: Ridge**

Lasso is great for feature selection, but when building regression models, Ridge regression should be your first choice.

Recall that lasso performs regularization by adding to the loss function a penalty term of the *absolute* value of each coefficient multiplied by some alpha. This is also known as L1L1 regularization because the regularization term is the L1L1 norm of the coefficients. This is not the only way to regularize, however.

If instead you took the sum of the *squared* values of the coefficients multiplied by some alpha - like in Ridge regression - you would be computing the L2L2 norm. In this exercise, you will practice fitting ridge regression models over a range of different alphas, and plot cross-validated R2R2 scores for each, using this function that we have defined for you, which plots the R2R2 score as well as standard error for each alpha:

def display\_plot(cv\_scores, cv\_scores\_std):

fig = plt.figure()

ax = fig.add\_subplot(1,1,1)

ax.plot(alpha\_space, cv\_scores)

std\_error = cv\_scores\_std / np.sqrt(10)

ax.fill\_between(alpha\_space, cv\_scores + std\_error, cv\_scores - std\_error, alpha=0.2)

ax.set\_ylabel('CV Score +/- Std Error')

ax.set\_xlabel('Alpha')

ax.axhline(np.max(cv\_scores), linestyle='--', color='.5')

ax.set\_xlim([alpha\_space[0], alpha\_space[-1]])

ax.set\_xscale('log')

plt.show()

Don't worry about the specifics of the above function works. The motivation behind this exercise is for you to see how the R2R2 score varies with different alphas, and to understand the importance of selecting the right value for alpha. You'll learn how to tune alpha in the next chapter.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Instantiate a Ridge regressor and specify normalize=True.
* Inside the for loop:
  + Specify the alpha value for the regressor to use.
  + Perform 10-fold cross-validation on the regressor with the specified alpha. The data is available in the arrays X and y.
  + Append the average and the standard deviation of the computed cross-validated scores. NumPy has been pre-imported for you as np.
* Use the display\_plot() function to visualize the scores and standard deviations.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of cross\_val\_score(). Did you correctly specify the first argument? Expected ridge, but got reg.

Traceback (most recent call last):

File "script.py", line 20, in <module>

ridge\_cv\_scores = cross\_val\_score(reg, X, y, cv=10)

NameError: name 'reg' is not defined

In [1]:

+100 XP

Great work! Notice how the cross-validation scores change with different alphas. Which alpha should you pick? How can you fine-tune your model? You'll learn all about this in the next chapter!

**Exercise**

**Exercise**

**Metrics for classification**

In Chapter 1, you evaluated the performance of your k-NN classifier based on its accuracy. However, as Andy discussed, accuracy is not always an informative metric. In this exercise, you will dive more deeply into evaluating the performance of binary classifiers by computing a confusion matrix and generating a classification report.

You may have noticed in the video that the classification report consisted of three rows, and an additional *support* column. The *support* gives the number of samples of the true response that lie in that class - so in the video example, the support was the number of Republicans or Democrats in the test set on which the classification report was computed. The *precision*, *recall*, and *f1-score* columns, then, gave the respective metrics for that particular class.

Here, you'll work with the [**PIMA Indians**](https://www.kaggle.com/uciml/pima-indians-diabetes-database) dataset obtained from the UCI Machine Learning Repository. The goal is to predict whether or not a given female patient will contract diabetes based on features such as BMI, age, and number of pregnancies. Therefore, it is a binary classification problem. A target value of 0 indicates that the patient does *not* have diabetes, while a value of 1 indicates that the patient *does* have diabetes. As in Chapters 1 and 2, the dataset has been preprocessed to deal with missing values.

The dataset has been loaded into a DataFrame df and the feature and target variable arrays X and y have been created for you. In addition, sklearn.model\_selection.train\_test\_split and sklearn.neighbors.KNeighborsClassifier have already been imported.

Your job is to train a k-NN classifier to the data and evaluate its performance by generating a confusion matrix and classification report.

**Instructions**

**100 XP**

* Import classification\_report and confusion\_matrix from sklearn.metrics.
* Create training and testing sets with 40% of the data used for testing. Use a random state of 42.
* Instantiate a k-NN classifier with 6 neighbors, fit it to the training data, and predict the labels of the test set.
* Compute and print the confusion matrix and classification report using the confusion\_matrix() and classification\_report() functions.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import necessary modules

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

# Create training and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=42)

# Instantiate a k-NN classifier: knn

knn = KNeighborsClassifier(6)

# Fit the classifier to the training data

knn.fit(X\_train, y\_train)

# Predict the labels of the test data: y\_pred

y\_pred = knn.fit(X\_test)

# Generate the confusion matrix and classification report

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

**Incorrect Submission**

Did you call knn.predict()?

Traceback (most recent call last):

File "script.py", line 15, in <module>

y\_pred = knn.fit(X\_test)

TypeError: fit() missing 1 required positional argument: 'y'

<script.py> output:

precision recall f1-score support

0 0.77 0.85 0.81 206

1 0.62 0.49 0.55 102

avg / total 0.72 0.73 0.72 308

[[176 30]

[ 52 50]]

In [1]:

+100 XP

Excellent work! By analyzing the confusion matrix and classification report, you can get a much better understanding of your classifier's performance.

classification\_report(y\_test, y\_pred)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

classification\_report(y\_test, y\_pred)

NameError: name 'classification\_report' is not defined

In [2]: # Import necessary modules

from sklearn.metrics import classification\_report, confusion\_matrix

# Create training and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=42)

# Instantiate a k-NN classifier: knn

knn = KNeighborsClassifier(6)

# Fit the classifier to the training data

knn.fit(X\_train, y\_train)

# Predict the labels of the test data: y\_pred

y\_pred = knn.predict(X\_test)

# Generate the confusion matrix and classification report

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

precision recall f1-score support

0 0.77 0.85 0.81 206

1 0.62 0.49 0.55 102

avg / total 0.72 0.73 0.72 308

[[176 30]

[ 52 50]]

In [3]: classification\_report(y\_test, y\_pred)

Out[3]: ' precision recall f1-score support\n\n 0 0.77 0.85 0.81 206\n 1 0.62 0.49 0.55 102\n\navg / total 0.72 0.73 0.72 308\n'

In [4]: classification\_report(y\_pred, y\_test)

Out[4]: ' precision recall f1-score support\n\n 0 0.85 0.77 0.81 228\n 1 0.49 0.62 0.55 80\n\navg / total 0.76 0.73 0.74 308\n'

In [5]:

**Exercise**

**Exercise**

**Building a logistic regression model**

Time to build your first logistic regression model! As Hugo showed in the video, scikit-learn makes it very easy to try different models, since the Train-Test-Split/Instantiate/Fit/Predict paradigm applies to all classifiers and regressors - which are known in scikit-learn as 'estimators'. You'll see this now for yourself as you train a logistic regression model on exactly the same data as in the previous exercise. Will it outperform k-NN? There's only one way to find out!

The feature and target variable arrays X and y have been pre-loaded, and train\_test\_split has been imported for you from sklearn.model\_selection.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import:
  + LogisticRegression from sklearn.linear\_model.
  + confusion\_matrix and classification\_report from sklearn.metrics.
* Create training and test sets with 40% (or 0.4) of the data used for testing. Use a random state of 42. This has been done for you.
* Instantiate a LogisticRegression classifier called logreg.
* Fit the classifier to the training data and predict the labels of the test set.
* Compute and print the confusion matrix and classification report. This has been done for you, so hit 'Submit Answer' to see how logistic regression compares to k-NN!

[**Take Hint (-30 XP)**](javascript:void(0))

# Import the necessary modules

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix, classification\_report

# Create training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.4, random\_state=42)

# Create the classifier: logreg

logreg = LogisticRegression()

# Fit the classifier to the training data

logreg.fit(X\_train, y\_train)

# Predict the labels of the test set: y\_pred

y\_pred = logreg.predict(X\_test)

# Compute and print the confusion matrix and classification report

print(confusion\_matrix(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred))

<script.py> output:

[[176 30]

[ 35 67]]

precision recall f1-score support

0 0.83 0.85 0.84 206

1 0.69 0.66 0.67 102

avg / total 0.79 0.79 0.79 308

In [1]:

+100 XP

You now know how to use logistic regression for binary classification - great work! Logistic regression is used in a variety of machine learning applications and will become a vital part of your data science toolbox.

**Plotting an ROC curve**

Great job in the previous exercise - you now have a new addition to your toolbox of classifiers!

Classification reports and confusion matrices are great methods to quantitatively evaluate model performance, while ROC curves provide a way to visually evaluate models. As Hugo demonstrated in the video, most classifiers in scikit-learn have a .predict\_proba() method which returns the probability of a given sample being in a particular class. Having built a logistic regression model, you'll now evaluate its performance by plotting an ROC curve. In doing so, you'll make use of the .predict\_proba() method and become familiar with its functionality.

Here, you'll continue working with the PIMA Indians diabetes dataset. The classifier has already been fit to the training data and is available as logreg.

**Instructions**

**100 XP**

* Import roc\_curve from sklearn.metrics.
* Using the logreg classifier, which has been fit to the training data, compute the predicted probabilities of the labels of the test set X\_test. Save the result as y\_pred\_prob.
* Use the roc\_curve() function with y\_test and y\_pred\_prob and unpack the result into the variables fpr, tpr, and thresholds.
* Plot the ROC curve with fpr on the x-axis and tpr on the y-axis.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import necessary modules

from sklearn.metrics import roc\_curve

# Compute predicted probabilities: y\_pred\_prob

y\_pred\_prob = logreg.predict\_proba(X\_test)[:,1]

# Generate ROC curve values: fpr, tpr, thresholds

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred\_prob)

# Plot ROC curve

plt.plot([0, 1], [0, 1], 'k--')

plt.plot(fpr, tpr)

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve')

plt.show()

**Incorrect Submission**

Did you correctly compute the predicted probabilities using .predict\_proba()?

Traceback (most recent call last):

File "script.py", line 5, in <module>

y\_pred\_prob = logreg.predict\_proba[:,1]

TypeError: 'method' object is not subscriptable

In [1]:

+100 XP

Excellent! This ROC curve provides a nice visual way to assess your classifier's performance.

**Exercise**

**Exercise**

**Precision-recall Curve**

When looking at your ROC curve, you may have noticed that the y-axis (True positive rate) is also known as recall. Indeed, in addition to the ROC curve, there are other ways to visually evaluate model performance. One such way is the precision-recall curve, which is generated by plotting the precision and recall for different thresholds. As a reminder, precision and recall are defined as:

Precision=TPTP+FPPrecision=TPTP+FP

Recall=TPTP+FNRecall=TPTP+FN

On the right, a precision-recall curve has been generated for the diabetes dataset. The classification report and confusion matrix are displayed in the IPython Shell.

Study the precision-recall curve and then consider the statements given below. Choose the one statement that is **not** true. Note that here, the class is positive (1) if the individual *has* diabetes.

**Instructions**

**50 XP**

**Possible Answers**

* 

A recall of 1 corresponds to a classifier with a low threshold in which *all* females who contract diabetes were correctly classified as such, at the expense of many misclassifications of those who did *not* have diabetes.

* 

Precision is undefined for a classifier which makes *no* positive predictions, that is, classifies *everyone* as *not* having diabetes.

* 

When the threshold is very close to 1, precision is also 1, because the classifier is absolutely certain about its predictions.

* 

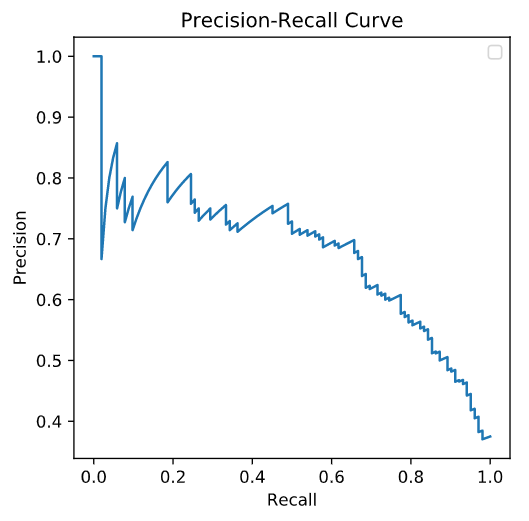
Precision and recall take *true negatives* into consideration.

**Submit Answer**

[**Take Hint (-15 XP)**](javascript:void(0))

+50 XP

Great work! True negatives do not appear at all in the definitions of precision and recall.



precision recall f1-score support

0 0.83 0.85 0.84 206

1 0.69 0.66 0.67 102

avg / total 0.79 0.79 0.79 308

[[176 30]

[ 35 67]]

In [1]:

**Exercise**

**Exercise**

**AUC computation**

Say you have a binary classifier that in fact is just randomly making guesses. It would be correct approximately 50% of the time, and the resulting ROC curve would be a diagonal line in which the True Positive Rate and False Positive Rate are always equal. The Area under this ROC curve would be 0.5. This is one way in which the AUC, which Hugo discussed in the video, is an informative metric to evaluate a model. If the AUC is greater than 0.5, the model is better than random guessing. Always a good sign!

In this exercise, you'll calculate AUC scores using the roc\_auc\_score() function from sklearn.metrics as well as by performing cross-validation on the diabetes dataset.

X and y, along with training and test sets X\_train, X\_test, y\_train, y\_test, have been pre-loaded for you, and a logistic regression classifier logreg has been fit to the training data.

**Instructions**

**100 XP**

* Import roc\_auc\_score from sklearn.metrics and cross\_val\_score from sklearn.model\_selection.
* Using the logreg classifier, which has been fit to the training data, compute the predicted probabilities of the labels of the test set X\_test. Save the result as y\_pred\_prob.
* Compute the AUC score using the roc\_auc\_score() function, the test set labels y\_test, and the predicted probabilities y\_pred\_prob.
* Compute the AUC scores by performing 5-fold cross-validation. Use the cross\_val\_score() function and specify the scoring parameter to be 'roc\_auc'.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you specified the arguments for cross\_val\_score() using the right syntax?

# Import necessary modules

from sklearn.metrics import roc\_auc\_score

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

# Compute predicted probabilities: y\_pred\_prob

y\_pred\_prob = logreg.predict\_proba(X\_test)[:,1]

# Compute and print AUC score

print("AUC: {}".format(roc\_auc\_score(y\_test, y\_pred\_prob)))

# Compute cross-validated AUC scores: cv\_auc

cv\_auc = cross\_val\_score(logreg, X, y, cv=5,

scoring='roc\_auc')

# Print list of AUC scores

print("AUC scores computed using 5-fold cross-validation: {}".format(cv\_auc))

In [1]: logreg

Out[1]:

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, max\_iter=100, multi\_class='ovr', n\_jobs=1,

penalty='l2', random\_state=None, solver='liblinear', tol=0.0001,

verbose=0, warm\_start=False)

<script.py> output:

AUC: 0.8254806777079764

Traceback (most recent call last):

File "script.py", line 12, in <module>

cv\_auc = cross\_val\_score()

TypeError: cross\_val\_score() missing 2 required positional arguments: 'estimator' and 'X'

In [2]:

<script.py> output:

AUC: 0.8254806777079764

Traceback (most recent call last):

File "script.py", line 12, in <module>

cv\_auc = cross\_val\_score()

TypeError: cross\_val\_score() missing 2 required positional arguments: 'estimator' and 'X'

In [2]:

In

Out[2]: ['', 'logreg', 'In']

In [3]: C

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

C

NameError: name 'C' is not defined

In [4]: X

Out[4]:

pregnancies glucose diastolic triceps insulin bmi dpf \

0 6 148 72 35.00000 155.548223 33.600000 0.627

1 1 85 66 29.00000 155.548223 26.600000 0.351

2 8 183 64 29.15342 155.548223 23.300000 0.672

3 1 89 66 23.00000 94.000000 28.100000 0.167

4 0 137 40 35.00000 168.000000 43.100000 2.288

5 5 116 74 29.15342 155.548223 25.600000 0.201

6 3 78 50 32.00000 88.000000 31.000000 0.248

7 10 115 0 29.15342 155.548223 35.300000 0.134

8 2 197 70 45.00000 543.000000 30.500000 0.158

9 8 125 96 29.15342 155.548223 32.457464 0.232

10 4 110 92 29.15342 155.548223 37.600000 0.191

11 10 168 74 29.15342 155.548223 38.000000 0.537

12 10 139 80 29.15342 155.548223 27.100000 1.441

13 1 189 60 23.00000 846.000000 30.100000 0.398

14 5 166 72 19.00000 175.000000 25.800000 0.587

15 7 100 0 29.15342 155.548223 30.000000 0.484

16 0 118 84 47.00000 230.000000 45.800000 0.551

17 7 107 74 29.15342 155.548223 29.600000 0.254

18 1 103 30 38.00000 83.000000 43.300000 0.183

19 1 115 70 30.00000 96.000000 34.600000 0.529

20 3 126 88 41.00000 235.000000 39.300000 0.704

21 8 99 84 29.15342 155.548223 35.400000 0.388

22 7 196 90 29.15342 155.548223 39.800000 0.451

23 9 119 80 35.00000 155.548223 29.000000 0.263

24 11 143 94 33.00000 146.000000 36.600000 0.254

25 10 125 70 26.00000 115.000000 31.100000 0.205

26 7 147 76 29.15342 155.548223 39.400000 0.257

27 1 97 66 15.00000 140.000000 23.200000 0.487

28 13 145 82 19.00000 110.000000 22.200000 0.245

29 5 117 92 29.15342 155.548223 34.100000 0.337

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

738 2 99 60 17.00000 160.000000 36.600000 0.453

739 1 102 74 29.15342 155.548223 39.500000 0.293

740 11 120 80 37.00000 150.000000 42.300000 0.785

741 3 102 44 20.00000 94.000000 30.800000 0.400

742 1 109 58 18.00000 116.000000 28.500000 0.219

743 9 140 94 29.15342 155.548223 32.700000 0.734

744 13 153 88 37.00000 140.000000 40.600000 1.174

745 12 100 84 33.00000 105.000000 30.000000 0.488

746 1 147 94 41.00000 155.548223 49.300000 0.358

747 1 81 74 41.00000 57.000000 46.300000 1.096

748 3 187 70 22.00000 200.000000 36.400000 0.408

749 6 162 62 29.15342 155.548223 24.300000 0.178

750 4 136 70 29.15342 155.548223 31.200000 1.182

751 1 121 78 39.00000 74.000000 39.000000 0.261

752 3 108 62 24.00000 155.548223 26.000000 0.223

753 0 181 88 44.00000 510.000000 43.300000 0.222

754 8 154 78 32.00000 155.548223 32.400000 0.443

755 1 128 88 39.00000 110.000000 36.500000 1.057

756 7 137 90 41.00000 155.548223 32.000000 0.391

757 0 123 72 29.15342 155.548223 36.300000 0.258

758 1 106 76 29.15342 155.548223 37.500000 0.197

759 6 190 92 29.15342 155.548223 35.500000 0.278

760 2 88 58 26.00000 16.000000 28.400000 0.766

761 9 170 74 31.00000 155.548223 44.000000 0.403

762 9 89 62 29.15342 155.548223 22.500000 0.142

763 10 101 76 48.00000 180.000000 32.900000 0.171

764 2 122 70 27.00000 155.548223 36.800000 0.340

765 5 121 72 23.00000 112.000000 26.200000 0.245

766 1 126 60 29.15342 155.548223 30.100000 0.349

767 1 93 70 31.00000 155.548223 30.400000 0.315

age

0 50

1 31

2 32

3 21

4 33

5 30

6 26

7 29

8 53

9 54

10 30

11 34

12 57

13 59

14 51

15 32

16 31

17 31

18 33

19 32

20 27

21 50

22 41

23 29

24 51

25 41

26 43

27 22

28 57

29 38

.. ...

738 21

739 42

740 48

741 26

742 22

743 45

744 39

745 46

746 27

747 32

748 36

749 50

750 22

751 28

752 25

753 26

754 45

755 37

756 39

757 52

758 26

759 66

760 22

761 43

762 33

763 63

764 27

765 30

766 47

767 23

[768 rows x 8 columns]

In [5]: y

Out[5]:

0 1

1 0

2 1

3 0

4 1

5 0

6 1

7 0

8 1

9 1

10 0

11 1

12 0

13 1

14 1

15 1

16 1

17 1

18 0

19 1

20 0

21 0

22 1

23 1

24 1

25 1

26 1

27 0

28 0

29 0

..

738 0

739 1

740 1

741 0

742 0

743 1

744 0

745 0

746 1

747 0

748 1

749 1

750 1

751 0

752 0

753 1

754 1

755 1

756 0

757 1

758 0

759 1

760 0

761 1

762 0

763 0

764 0

765 0

766 1

767 0

Name: diabetes, dtype: int64

<script.py> output:

AUC: 0.8254806777079764

AUC scores computed using 5-fold cross-validation: [0.80148148 0.8062963 0.81481481 0.86245283 0.8554717 ]

In [6]:

+100 XP

Great work! You now have a number of different methods you can use to evaluate your model's performance.

**Exercise**

**Exercise**

**Hyperparameter tuning with GridSearchCV**

Hugo demonstrated how to tune the n\_neighbors parameter of the KNeighborsClassifier() using GridSearchCV on the voting dataset. You will now practice this yourself, but by using logistic regression on the diabetes dataset instead!

Like the alpha parameter of lasso and ridge regularization that you saw earlier, logistic regression also has a regularization parameter: CC. CC controls the *inverse* of the regularization strength, and this is what you will tune in this exercise. A large CC can lead to an *overfit* model, while a small CC can lead to an *underfit* model.

The hyperparameter space for CC has been setup for you. Your job is to use GridSearchCV and logistic regression to find the optimal CC in this hyperparameter space. The feature array is available as X and target variable array is available as y.

You may be wondering why you aren't asked to split the data into training and test sets. Good observation! Here, we want you to focus on the process of setting up the hyperparameter grid and performing grid-search cross-validation. In practice, you will indeed want to hold out a portion of your data for evaluation purposes, and you will learn all about this in the next video!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import LogisticRegression from sklearn.linear\_model and GridSearchCV from sklearn.model\_selection.
* Setup the hyperparameter grid by using c\_space as the grid of values to tune CC over.
* Instantiate a logistic regression classifier called logreg.
* Use GridSearchCV with 5-fold cross-validation to tune CC:
  + Inside GridSearchCV(), specify the classifier, parameter grid, and number of folds to use.
  + Use the .fit() method on the GridSearchCV object to fit it to the data X and y.
* Print the best parameter and best score obtained from GridSearchCV by accessing the best\_params\_ and best\_score\_ attributes of logreg\_cv.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import necessary modules

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import GridSearchCV

# Setup the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

param\_grid = {'C': \_\_\_\_}

# Instantiate a logistic regression classifier: logreg

logreg = LogisticRegression()

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(X, y, cv=5)

# Fit it to the data

logreg\_cv.fit(X, y)

# Print the tuned parameters and score

print("Tuned Logistic Regression Parameters: {}".format(logreg\_cv.best\_params\_))

print("Best score is {}".format(logreg\_cv.best\_score\_))

**Incorrect Submission**

Your code could not be parsed due to an error in the indentation:  
unexpected indent (script.py, line 2).

Check your call of GridSearchCV(). Did you correctly specify the first argument? Expected logreg, but got X.

# Import necessary modules

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import GridSearchCV

# Setup the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

param\_grid = {'C': c\_space}

# Instantiate a logistic regression classifier: logreg

logreg = LogisticRegression()

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(logreg, param\_grid, cv=5)

# Fit it to the data

logreg\_cv.fit(X, y)

# Print the tuned parameters and score

print("Tuned Logistic Regression Parameters: {}".format(logreg\_cv.best\_params\_))

print("Best score is {}".format(logreg\_cv.best\_score\_))

In [1]: best\_params\_

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

best\_params\_

NameError: name 'best\_params\_' is not defined

In [2]: # Import necessary modules

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import LogisticRegression

# Setup the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

param\_grid = {'C': \_\_\_\_}

# Instantiate a logistic regression classifier: logreg

logreg = \_\_\_\_

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(X, y, cv=5)

logreg\_cv.best\_params\_

# Fit it to the data

logreg\_cv.fit(X, y)

# Print the tuned parameters and score

print("Tuned Logistic Regression Parameters: {}".format(\_\_\_\_))

print("Best score is {}".format(\_\_\_\_))

Traceback (most recent call last):

File "<stdin>", line 3, in <module>

from sklearn.model\_selection import LogisticRegression

ImportError: cannot import name 'LogisticRegression'

In [3]: # Import necessary modules

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import GridSearchCV

# Setup the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

param\_grid = {'C': \_\_\_\_}

# Instantiate a logistic regression classifier: logreg

logreg = \_\_\_\_

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(X, y, cv=5)

logreg\_cv.best\_params\_

# Fit it to the data

logreg\_cv.fit(X, y)

# Print the tuned parameters and score

print("Tuned Logistic Regression Parameters: {}".format(\_\_\_\_))

print("Best score is {}".format(\_\_\_\_))

Traceback (most recent call last):

File "<stdin>", line 7, in <module>

param\_grid = {'C': \_\_\_\_}

NameError: name '\_\_\_\_' is not defined

In [4]: # Import necessary modules

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import GridSearchCV

# Setup the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

#param\_grid = {'C': \_\_\_\_}

# Instantiate a logistic regression classifier: logreg

#logreg = \_\_\_\_

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(X, y, cv=5)

logreg\_cv.best\_params\_

# Fit it to the data

logreg\_cv.fit(X, y)

# Print the tuned parameters and score

print("Tuned Logistic Regression Parameters: {}".format(\_\_\_\_))

print("Best score is {}".format(\_\_\_\_))

Traceback (most recent call last):

File "<stdin>", line 13, in <module>

logreg\_cv = GridSearchCV(X, y, cv=5)

File "<stdin>", line 1078, in \_\_init\_\_

\_check\_param\_grid(param\_grid)

File "<stdin>", line 357, in \_check\_param\_grid

" np.ndarray.".format(name))

ValueError: Parameter values for parameter (0) need to be a sequence(but not a string) or np.ndarray.

Traceback (most recent call last):

File "script.py", line 7, in <module>

param\_grid = {'C': \_\_\_\_}

NameError: name '\_\_\_\_' is not defined

Traceback (most recent call last):

File "script.py", line 13, in <module>

logreg\_cv = GridSearchCV(X, y, cv=5)

File "script.py", line 1078, in \_\_init\_\_

\_check\_param\_grid(param\_grid)

File "script.py", line 357, in \_check\_param\_grid

" np.ndarray.".format(name))

ValueError: Parameter values for parameter (0) need to be a sequence(but not a string) or np.ndarray.

<script.py> output:

Tuned Logistic Regression Parameters: {'C': 3.727593720314938}

Best score is 0.7708333333333334

In [5]:

+100 XP

Good job! It looks like a 'C' of 3.727 results in the best performance.

**Exercise**

**Exercise**

**Hyperparameter tuning with RandomizedSearchCV**

GridSearchCV can be computationally expensive, especially if you are searching over a large hyperparameter space and dealing with multiple hyperparameters. A solution to this is to use RandomizedSearchCV, in which not all hyperparameter values are tried out. Instead, a fixed number of hyperparameter settings is sampled from specified probability distributions. You'll practice using RandomizedSearchCV in this exercise and see how this works.

Here, you'll also be introduced to a new model: the Decision Tree. Don't worry about the specifics of how this model works. Just like k-NN, linear regression, and logistic regression, decision trees in scikit-learn have .fit() and .predict() methods that you can use in exactly the same way as before. Decision trees have many parameters that can be tuned, such as max\_features, max\_depth, and min\_samples\_leaf: This makes it an ideal use case for RandomizedSearchCV.

As before, the feature array X and target variable array y of the diabetes dataset have been pre-loaded. The hyperparameter settings have been specified for you. Your goal is to use RandomizedSearchCV to find the optimal hyperparameters. Go for it!

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import DecisionTreeClassifier from sklearn.tree and RandomizedSearchCV from sklearn.model\_selection.
* Specify the parameters and distributions to sample from. This has been done for you.
* Instantiate a DecisionTreeClassifier.
* Use RandomizedSearchCV with 5-fold cross-validation to tune the hyperparameters:
  + Inside RandomizedSearchCV(), specify the classifier, parameter distribution, and number of folds to use.
  + Use the .fit() method on the RandomizedSearchCV object to fit it to the data X and y.
* Print the best parameter and best score obtained from RandomizedSearchCV by accessing the best\_params\_ and best\_score\_ attributes of tree\_cv.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import necessary modules

from scipy.stats import randint

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import RandomizedSearchCV

# Setup the parameters and distributions to sample from: param\_dist

param\_dist = {"max\_depth": [3, None],

"max\_features": randint(1, 9),

"min\_samples\_leaf": randint(1, 9),

"criterion": ["gini", "entropy"]}

# Instantiate a Decision Tree classifier: tree

tree = DecisionTreeClassifier()

# Instantiate the RandomizedSearchCV object: tree\_cv

tree\_cv = RandomizedSearchCV(X, y, cv=5)

# Fit it to the data

tree\_cv.fit()

# Print the tuned parameters and score

print("Tuned Decision Tree Parameters: {}".format(tree\_cv.best\_params\_))

print("Best score is {}".format(tree\_cv.best\_score\_))

**Incorrect Submission**

In RandomizedSearchCV(), did you correctly specify the classifier?

Have you specified the arguments for RandomizedSearchCV() using the right syntax?

# Import necessary modules

from scipy.stats import randint

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import RandomizedSearchCV

# Setup the parameters and distributions to sample from: param\_dist

param\_dist = {"max\_depth": [3, None],

"max\_features": randint(1, 9),

"min\_samples\_leaf": randint(1, 9),

"criterion": ["gini", "entropy"]}

# Instantiate a Decision Tree classifier: tree

tree = DecisionTreeClassifier()

# Instantiate the RandomizedSearchCV object: tree\_cv

tree\_cv = RandomizedSearchCV(tree, param\_dist, cv=5)

# Fit it to the data

tree\_cv.fit(X, y)

# Print the tuned parameters and score

print("Tuned Decision Tree Parameters: {}".format(tree\_cv.best\_params\_))

print("Best score is {}".format(tree\_cv.best\_score\_))

Traceback (most recent call last):

File "script.py", line 19, in <module>

tree\_cv.fit()

TypeError: fit() missing 1 required positional argument: 'X'

In [1]: tree\_cv.fit(X, y)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

tree\_cv.fit(X, y)

NameError: name 'tree\_cv' is not defined

Traceback (most recent call last):

File "script.py", line 16, in <module>

tree\_cv = RandomizedSearchCV(cv=5)

TypeError: \_\_init\_\_() missing 2 required positional arguments: 'estimator' and 'param\_distributions'

<script.py> output:

Tuned Decision Tree Parameters: {'criterion': 'gini', 'max\_depth': 3, 'max\_features': 5, 'min\_samples\_leaf': 2}

Best score is 0.7395833333333334

In [2]:

+100 XP

Great work! You'll see a lot more of decision trees and RandomizedSearchCV as you continue your machine learning journey. Note that RandomizedSearchCV will never outperform GridSearchCV. Instead, it is valuable because it saves on computation time.

**Hold-out set reasoning**

For which of the following reasons would you want to use a hold-out set for the very end?

**Answer the question**

**50 XP**

**Possible Answers**

You want to maximize the amount of training data used.

press

1

You want to be absolutely certain about your model's ability to generalize to unseen data.

press

2

You want to tune the hyperparameters of your model.

press

3

**Incorrect Submission**

Incorrect. You can tune the hyperparameters of your model even without using a hold-out set.

+50 XP

Correct! The idea is to tune the model's hyperparameters on the training set, and then evaluate its performance on the hold-out set which it has never seen before.

**Exercise**

**Exercise**

**Hold-out set in practice I: Classification**

You will now practice evaluating a model with tuned hyperparameters on a hold-out set. The feature array and target variable array from the diabetes dataset have been pre-loaded as X and y.

In addition to CC, logistic regression has a 'penalty' hyperparameter which specifies whether to use 'l1' or 'l2' regularization. Your job in this exercise is to create a hold-out set, tune the 'C' and 'penalty' hyperparameters of a logistic regression classifier using GridSearchCV on the training set.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Create the hyperparameter grid:
  + Use the array c\_space as the grid of values for 'C'.
  + For 'penalty', specify a list consisting of 'l1' and 'l2'.
* Instantiate a logistic regression classifier.
* Create training and test sets. Use a test\_size of 0.4 and random\_state of 42. In practice, the test set here will function as the hold-out set.
* Tune the hyperparameters on the training set using GridSearchCV with 5-folds. This involves first instantiating the GridSearchCV object with the correct parameters and then fitting it to the training data.
* Print the best parameter and best score obtained from GridSearchCV by accessing the best\_params\_ and best\_score\_ attributes of logreg\_cv.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you define the variable param\_grid without errors?

Check your call of train\_test\_split(). Did you specify the argument 0?

Did you correctly pass in the feature variable to train\_test\_split()?

Check your call of train\_test\_split(). Did you specify the argument test\_size?

Check your call of GridSearchCV(). Did you correctly specify the first argument? Expected logreg, but got knn.

Check your call of logreg\_cv.fit(). Did you correctly specify the first argument? Expected something different.

Did you print the best parameter by accessing the best\_params\_ attribute of logreg\_cv?

# Import necessary modules

from sklearn.model\_selection import train\_test\_split as tts

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import GridSearchCV

# Create the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

param\_grid = {'C': c\_space, 'penalty': ['l1', 'l2']}

# Instantiate the logistic regression classifier: logreg

logreg = LogisticRegression()

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size=0.4, random\_state=42)

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(logreg, param\_grid, cv=5)

# Fit it to the training data

logreg\_cv.fit(X\_train, y\_train)

# Print the optimal parameters and best score

print("Tuned Logistic Regression Parameter: {}".format(logreg\_cv.best\_params\_))

print("Tuned Logistic Regression Accuracy: {}".format(logreg\_cv.best\_score\_))

In [1]: best\_params\_

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

best\_params\_

NameError: name 'best\_params\_' is not defined

In [2]: X, y

Out[2]:

( pregnancies glucose diastolic triceps insulin bmi dpf \

0 6 148 72 35.00000 155.548223 33.600000 0.627

1 1 85 66 29.00000 155.548223 26.600000 0.351

2 8 183 64 29.15342 155.548223 23.300000 0.672

3 1 89 66 23.00000 94.000000 28.100000 0.167

4 0 137 40 35.00000 168.000000 43.100000 2.288

5 5 116 74 29.15342 155.548223 25.600000 0.201

6 3 78 50 32.00000 88.000000 31.000000 0.248

7 10 115 0 29.15342 155.548223 35.300000 0.134

8 2 197 70 45.00000 543.000000 30.500000 0.158

9 8 125 96 29.15342 155.548223 32.457464 0.232

10 4 110 92 29.15342 155.548223 37.600000 0.191

11 10 168 74 29.15342 155.548223 38.000000 0.537

12 10 139 80 29.15342 155.548223 27.100000 1.441

13 1 189 60 23.00000 846.000000 30.100000 0.398

14 5 166 72 19.00000 175.000000 25.800000 0.587

15 7 100 0 29.15342 155.548223 30.000000 0.484

16 0 118 84 47.00000 230.000000 45.800000 0.551

17 7 107 74 29.15342 155.548223 29.600000 0.254

18 1 103 30 38.00000 83.000000 43.300000 0.183

19 1 115 70 30.00000 96.000000 34.600000 0.529

20 3 126 88 41.00000 235.000000 39.300000 0.704

21 8 99 84 29.15342 155.548223 35.400000 0.388

22 7 196 90 29.15342 155.548223 39.800000 0.451

23 9 119 80 35.00000 155.548223 29.000000 0.263

24 11 143 94 33.00000 146.000000 36.600000 0.254

25 10 125 70 26.00000 115.000000 31.100000 0.205

26 7 147 76 29.15342 155.548223 39.400000 0.257

27 1 97 66 15.00000 140.000000 23.200000 0.487

28 13 145 82 19.00000 110.000000 22.200000 0.245

29 5 117 92 29.15342 155.548223 34.100000 0.337

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

738 2 99 60 17.00000 160.000000 36.600000 0.453

739 1 102 74 29.15342 155.548223 39.500000 0.293

740 11 120 80 37.00000 150.000000 42.300000 0.785

741 3 102 44 20.00000 94.000000 30.800000 0.400

742 1 109 58 18.00000 116.000000 28.500000 0.219

743 9 140 94 29.15342 155.548223 32.700000 0.734

744 13 153 88 37.00000 140.000000 40.600000 1.174

745 12 100 84 33.00000 105.000000 30.000000 0.488

746 1 147 94 41.00000 155.548223 49.300000 0.358

747 1 81 74 41.00000 57.000000 46.300000 1.096

748 3 187 70 22.00000 200.000000 36.400000 0.408

749 6 162 62 29.15342 155.548223 24.300000 0.178

750 4 136 70 29.15342 155.548223 31.200000 1.182

751 1 121 78 39.00000 74.000000 39.000000 0.261

752 3 108 62 24.00000 155.548223 26.000000 0.223

753 0 181 88 44.00000 510.000000 43.300000 0.222

754 8 154 78 32.00000 155.548223 32.400000 0.443

755 1 128 88 39.00000 110.000000 36.500000 1.057

756 7 137 90 41.00000 155.548223 32.000000 0.391

757 0 123 72 29.15342 155.548223 36.300000 0.258

758 1 106 76 29.15342 155.548223 37.500000 0.197

759 6 190 92 29.15342 155.548223 35.500000 0.278

760 2 88 58 26.00000 16.000000 28.400000 0.766

761 9 170 74 31.00000 155.548223 44.000000 0.403

762 9 89 62 29.15342 155.548223 22.500000 0.142

763 10 101 76 48.00000 180.000000 32.900000 0.171

764 2 122 70 27.00000 155.548223 36.800000 0.340

765 5 121 72 23.00000 112.000000 26.200000 0.245

766 1 126 60 29.15342 155.548223 30.100000 0.349

767 1 93 70 31.00000 155.548223 30.400000 0.315

age

0 50

1 31

2 32

3 21

4 33

5 30

6 26

7 29

8 53

9 54

10 30

11 34

12 57

13 59

14 51

15 32

16 31

17 31

18 33

19 32

20 27

21 50

22 41

23 29

24 51

25 41

26 43

27 22

28 57

29 38

.. ...

738 21

739 42

740 48

741 26

742 22

743 45

744 39

745 46

746 27

747 32

748 36

749 50

750 22

751 28

752 25

753 26

754 45

755 37

756 39

757 52

758 26

759 66

760 22

761 43

762 33

763 63

764 27

765 30

766 47

767 23

[768 rows x 8 columns], 0 1

1 0

2 1

3 0

4 1

5 0

6 1

7 0

8 1

9 1

10 0

11 1

12 0

13 1

14 1

15 1

16 1

17 1

18 0

19 1

20 0

21 0

22 1

23 1

24 1

25 1

26 1

27 0

28 0

29 0

..

738 0

739 1

740 1

741 0

742 0

743 1

744 0

745 0

746 1

747 0

748 1

749 1

750 1

751 0

752 0

753 1

754 1

755 1

756 0

757 1

758 0

759 1

760 0

761 1

762 0

763 0

764 0

765 0

766 1

767 0

Name: diabetes, dtype: int64)

Traceback (most recent call last):

File "script.py", line 8, in <module>

param\_grid = {\_\_\_\_: \_\_\_\_, \_\_\_\_: ['l1', 'l2']}

NameError: name '\_\_\_\_' is not defined

In [3]: # Import necessary modules

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import GridSearchCV

# Create the hyperparameter grid

c\_space = np.logspace(-5, 8, 15)

param\_grid = {c\_space: 'C', 'penalty': ['l1', 'l2']}

# Instantiate the logistic regression classifier: logreg

logreg = \_\_\_\_

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = \_\_\_\_

# Instantiate the GridSearchCV object: logreg\_cv

logreg\_cv = GridSearchCV(test\_size=0.4, random\_state=42, cv=5)

# Fit it to the training data

logreg\_cv.fit(X, y)

# Print the optimal parameters and best score

print("Tuned Logistic Regression Parameter: {}".format(best\_params\_))

print("Tuned Logistic Regression Accuracy: {}".format(best\_score\_))

Traceback (most recent call last):

File "<stdin>", line 8, in <module>

param\_grid = {c\_space: 'C', 'penalty': ['l1', 'l2']}

TypeError: unhashable type: 'numpy.ndarray'

Traceback (most recent call last):

File "script.py", line 8, in <module>

param\_grid = {c\_space: 'C', 'penalty': ['l1', 'l2']}

TypeError: unhashable type: 'numpy.ndarray'

Traceback (most recent call last):

File "script.py", line 8, in <module>

param\_grid = {c\_space: 'C', 'penalty': ['l1', 'l2']}

TypeError: unhashable type: 'numpy.ndarray'

Traceback (most recent call last):

File "script.py", line 14, in <module>

X\_train, X\_test, y\_train, y\_test = tts()

File "script.py", line 2010, in train\_test\_split

raise ValueError("At least one array required as input")

ValueError: At least one array required as input

Traceback (most recent call last):

File "script.py", line 14, in <module>

X\_train, X\_test, y\_train, y\_test = tts(0)

File "script.py", line 2031, in train\_test\_split

arrays = indexable(\*arrays)

File "script.py", line 229, in indexable

check\_consistent\_length(\*result)

File "script.py", line 200, in check\_consistent\_length

lengths = [\_num\_samples(X) for X in arrays if X is not None]

File "script.py", line 200, in <listcomp>

lengths = [\_num\_samples(X) for X in arrays if X is not None]

File "script.py", line 119, in \_num\_samples

" a valid collection." % x)

TypeError: Singleton array array(0) cannot be considered a valid collection.

Traceback (most recent call last):

File "script.py", line 17, in <module>

logreg\_cv = GridSearchCV(test\_size=0.4, random\_state=42, cv=5)

TypeError: \_\_init\_\_() got an unexpected keyword argument 'test\_size'

Traceback (most recent call last):

File "script.py", line 17, in <module>

logreg\_cv = GridSearchCV(test\_size=0.4, random\_state=42, cv=5)

TypeError: \_\_init\_\_() got an unexpected keyword argument 'test\_size'

Traceback (most recent call last):

File "script.py", line 17, in <module>

logreg\_cv = GridSearchCV(cv=5)

TypeError: \_\_init\_\_() missing 2 required positional arguments: 'estimator' and 'param\_grid'

Traceback (most recent call last):

File "script.py", line 17, in <module>

logreg\_cv = GridSearchCV(knn, param\_grid, cv=5)

NameError: name 'knn' is not defined

Traceback (most recent call last):

File "script.py", line 23, in <module>

print("Tuned Logistic Regression Parameter: {}".format(best\_params\_))

NameError: name 'best\_params\_' is not defined

Traceback (most recent call last):

File "script.py", line 23, in <module>

print("Tuned Logistic Regression Parameter: {}".format(best\_params\_))

NameError: name 'best\_params\_' is not defined

<script.py> output:

Tuned Logistic Regression Parameter: {'C': 0.4393970560760795, 'penalty': 'l1'}

Tuned Logistic Regression Accuracy: 0.7652173913043478

In [4]:

+100 XP

Excellent work! You're really mastering the fundamentals of classification!

**Exercise**

**Exercise**

**Hold-out set in practice II: Regression**

Remember lasso and ridge regression from the previous chapter? Lasso used the L1L1 penalty to regularize, while ridge used the L2L2 penalty. There is another type of regularized regression known as the elastic net. In elastic net regularization, the penalty term is a linear combination of the L1L1 and L2L2 penalties:

a∗L1+b∗L2a∗L1+b∗L2

In scikit-learn, this term is represented by the 'l1\_ratio' parameter: An 'l1\_ratio' of 1 corresponds to an L1L1 penalty, and anything lower is a combination of L1L1 and L2L2.

In this exercise, you will GridSearchCV to tune the 'l1\_ratio' of an elastic net model trained on the Gapminder data. As in the previous exercise, use a hold-out set to evaluate your model's performance.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Import the following modules:
  + ElasticNet from sklearn.linear\_model.
  + mean\_squared\_error from sklearn.metrics.
  + GridSearchCV and train\_test\_split from sklearn.model\_selection.
* Create training and test sets, with 40% of the data used for the test set. Use a random state of 42.
* Specify the hyperparameter grid for 'l1\_ratio' using l1\_space as the grid of values to search over.
* Instantiate the ElasticNet regressor.
* Use GridSearchCV with 5-fold cross-validation to tune 'l1\_ratio' on the training data X\_train and y\_train. This involves first instantiating the GridSearchCV object with the correct parameters and then fitting it to the training data.
* Predict on the test set and compute the R2R2 and mean squared error.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of gm\_cv.score(). Did you correctly specify the first argument? Expected something different.

**Hint**

* Use the command from y import x to import x from y.
* Use train\_test\_split() to create training and test sets. Pass in the arguments X and y, and specify the keyword arguments test\_size=0.4 and random\_state=42.
* In the param\_grid dictionary, use l1\_space as the grid of values to tune 'l1\_ratio' over.
* Use ElasticNet() to instantiate the regressor.
* Inside GridSearchCV(), pass in the regressor elastic\_net, parameter grid param\_grid, and specify the number of folds to use. Then use the .fit() method on this with X\_train and y\_train passed in as arguments.
* To compute r2, use the .score() method with X\_test and y\_test as arguments. To compute mse, use the mean\_squared\_error() function with y\_test and y\_pred as arguments.

In [1]: # Import necessary modules

from sklearn.model\_selection import train\_test\_split as tts, GridSearchCV as gscv

from sklearn.metrics import mean\_squared\_error as msef

from sklearn.linear\_model import ElasticNet as en

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size=0.4, random\_state=42)

# Create the hyperparameter grid

l1\_space = np.linspace(0, 1, 30)

param\_grid = {'l1\_ratio': l1\_space}

# Instantiate the ElasticNet regressor: elastic\_net

elastic\_net = en()

# Setup the GridSearchCV object: gm\_cv

gm\_cv = gscv(elastic\_net, param\_grid, cv=5)

# Fit it to the training data

gm\_cv.fit(X\_train, y\_train)

# Predict on the test set and compute metrics

y\_pred = gm\_cv.predict(X\_test)

r2 = gm\_cv.score(X\_test, y\_pred)

mse = msef(X\_test, y\_test)

print("Tuned ElasticNet l1 ratio: {}".format(gm\_cv.best\_params\_))

print("Tuned ElasticNet R squared: {}".format(r2))

print("Tuned ElasticNet MSE: {}".format(mse))

Traceback (most recent call last):

File "<stdin>", line 25, in <module>

mse = msef(X\_test, y\_test)

File "<stdin>", line 238, in mean\_squared\_error

y\_true, y\_pred, multioutput)

File "<stdin>", line 87, in \_check\_reg\_targets

"({0}!={1})".format(y\_true.shape[1], y\_pred.shape[1]))

ValueError: y\_true and y\_pred have different number of output (8!=1)

In [2]: # Import necessary modules

from sklearn.model\_selection import train\_test\_split as tts, GridSearchCV as gscv

from sklearn.metrics import mean\_squared\_error as msef

from sklearn.linear\_model import ElasticNet as en

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size=0.4, random\_state=42)

# Create the hyperparameter grid

l1\_space = np.linspace(0, 1, 30)

param\_grid = {'l1\_ratio': l1\_space}

# Instantiate the ElasticNet regressor: elastic\_net

elastic\_net = en()

# Setup the GridSearchCV object: gm\_cv

gm\_cv = gscv(elastic\_net, param\_grid, cv=5)

# Fit it to the training data

gm\_cv.fit(X\_train, y\_train)

# Predict on the test set and compute metrics

y\_pred = gm\_cv.predict(X\_test)

r2 = gm\_cv.score(X\_test, y\_pred)

mse = msef(X\_test, y\_pred)

print("Tuned ElasticNet l1 ratio: {}".format(gm\_cv.best\_params\_))

print("Tuned ElasticNet R squared: {}".format(r2))

print("Tuned ElasticNet MSE: {}".format(mse))

Traceback (most recent call last):

File "<stdin>", line 25, in <module>

mse = msef(X\_test, y\_pred)

File "<stdin>", line 238, in mean\_squared\_error

y\_true, y\_pred, multioutput)

File "<stdin>", line 87, in \_check\_reg\_targets

"({0}!={1})".format(y\_true.shape[1], y\_pred.shape[1]))

ValueError: y\_true and y\_pred have different number of output (8!=1)

In [3]: y

Out[3]:

array([75.3, 58.3, 75.5, 72.5, 81.5, 80.4, 70.6, 72.2, 68.4, 75.3, 70.1,

79.4, 70.7, 63.2, 67.6, 70.9, 61.2, 73.9, 73.2, 59.4, 57.4, 66.2,

56.6, 80.7, 54.8, 78.9, 75.1, 62.6, 58.6, 79.7, 55.9, 76.5, 77.8,

78.7, 61. , 74. , 70.1, 74.1, 56.7, 60.4, 74. , 65.7, 79.4, 81. ,

57.5, 62.2, 72.1, 80. , 62.7, 79.5, 70.8, 58.3, 51.3, 63. , 61.7,

70.9, 73.8, 82. , 64.4, 69.5, 76.9, 79.4, 80.9, 81.4, 75.5, 82.6,

66.1, 61.5, 72.3, 77.6, 45.2, 61. , 72. , 80.7, 63.4, 51.4, 74.5,

78.2, 55.8, 81.4, 63.6, 72.1, 75.7, 69.6, 63.2, 73.3, 55. , 60.8,

68.6, 80.3, 80.2, 75.2, 59.7, 58. , 80.7, 74.6, 64.1, 77.1, 58.2,

73.6, 76.8, 69.4, 75.3, 79.2, 80.4, 73.4, 67.6, 62.2, 64.3, 76.4,

55.9, 80.9, 74.8, 78.5, 56.7, 55. , 81.1, 74.3, 67.4, 69.1, 46.1,

81.1, 81.9, 69.5, 59.7, 74.1, 60. , 71.3, 76.5, 75.1, 57.2, 68.2,

79.5, 78.2, 76. , 68.7, 75.4, 52. , 49. ])

Traceback (most recent call last):

File "script.py", line 24, in <module>

r2 = gm\_cv.score(y\_test, y\_pred)

File "script.py", line 439, in score

return score(self.best\_estimator\_, X, y)

File "script.py", line 244, in \_passthrough\_scorer

return estimator.score(\*args, \*\*kwargs)

File "script.py", line 386, in score

return r2\_score(y, self.predict(X), sample\_weight=sample\_weight,

File "script.py", line 256, in predict

return self.\_decision\_function(X)

File "script.py", line 791, in \_decision\_function

return super(ElasticNet, self).\_decision\_function(X)

File "script.py", line 239, in \_decision\_function

X = check\_array(X, accept\_sparse=['csr', 'csc', 'coo'])

File "script.py", line 441, in check\_array

"if it contains a single sample.".format(array))

ValueError: Expected 2D array, got 1D array instead:

array=[68.7 66.1 76.5 67.4 79.4 70.7 58.3 72.3 60. 70.9 74.8 57.2 63.6 59.4

55. 77.6 79.4 62.6 76. 76.4 70.1 74.3 55.8 81.5 80.7 75.1 79.5 76.9

61.2 73.2 70.1 73.8 80.4 82.6 73.4 62.2 72.1 80.3 74.6 75.3 58. 58.6

74. 54.8 75.5 74.1 52. 57.5 70.9 64.3 60.4 56.6 55.9 80.9 71.3 80. ].

Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a single sample.

In [4]: # Import necessary modules

from sklearn.model\_selection import train\_test\_split as tts, GridSearchCV as gscv

from sklearn.metrics import mean\_squared\_error as msef

from sklearn.linear\_model import ElasticNet as en

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size=0.4, random\_state=42)

# Create the hyperparameter grid

l1\_space = np.linspace(0, 1, 30)

param\_grid = {'l1\_ratio': l1\_space}

# Instantiate the ElasticNet regressor: elastic\_net

elastic\_net = en()

# Setup the GridSearchCV object: gm\_cv

gm\_cv = gscv(elastic\_net, param\_grid, cv=5)

# Fit it to the training data

gm\_cv.fit(X\_train, y\_train)

# Predict on the test set and compute metrics

y\_pred = gm\_cv.predict(X\_test)

r2 = gm\_cv.score(y\_test, y\_pred)

mse = msef(y\_test, y\_pred)

print("Tuned ElasticNet l1 ratio: {}".format(gm\_cv.best\_params\_))

print("Tuned ElasticNet R squared: {}".format(r2))

print("Tuned ElasticNet MSE: {}".format(mse))

Traceback (most recent call last):

File "<stdin>", line 24, in <module>

r2 = gm\_cv.score(y\_test, y\_pred)

File "<stdin>", line 439, in score

return score(self.best\_estimator\_, X, y)

File "<stdin>", line 244, in \_passthrough\_scorer

return estimator.score(\*args, \*\*kwargs)

File "<stdin>", line 386, in score

return r2\_score(y, self.predict(X), sample\_weight=sample\_weight,

File "<stdin>", line 256, in predict

return self.\_decision\_function(X)

File "<stdin>", line 791, in \_decision\_function

return super(ElasticNet, self).\_decision\_function(X)

File "<stdin>", line 239, in \_decision\_function

X = check\_array(X, accept\_sparse=['csr', 'csc', 'coo'])

File "<stdin>", line 441, in check\_array

"if it contains a single sample.".format(array))

ValueError: Expected 2D array, got 1D array instead:

array=[68.7 66.1 76.5 67.4 79.4 70.7 58.3 72.3 60. 70.9 74.8 57.2 63.6 59.4

55. 77.6 79.4 62.6 76. 76.4 70.1 74.3 55.8 81.5 80.7 75.1 79.5 76.9

61.2 73.2 70.1 73.8 80.4 82.6 73.4 62.2 72.1 80.3 74.6 75.3 58. 58.6

74. 54.8 75.5 74.1 52. 57.5 70.9 64.3 60.4 56.6 55.9 80.9 71.3 80. ].

Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a single sample.

In [5]: # Import necessary modules

from sklearn.model\_selection import train\_test\_split as tts, GridSearchCV as gscv

from sklearn.metrics import mean\_squared\_error as msef

from sklearn.linear\_model import ElasticNet as en

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size=0.4, random\_state=42)

# Create the hyperparameter grid

l1\_space = np.linspace(0, 1, 30)

param\_grid = {'l1\_ratio': l1\_space}

# Instantiate the ElasticNet regressor: elastic\_net

elastic\_net = en()

# Setup the GridSearchCV object: gm\_cv

gm\_cv = gscv(elastic\_net, param\_grid, cv=5)

# Fit it to the training data

gm\_cv.fit(X\_train, y\_train)

# Predict on the test set and compute metrics

y\_pred = gm\_cv.predict(X\_test)

r2 = gm\_cv.score(X\_test, y\_test )

mse = msef(y\_test, y\_pred)

print("Tuned ElasticNet l1 ratio: {}".format(gm\_cv.best\_params\_))

print("Tuned ElasticNet R squared: {}".format(r2))

print("Tuned ElasticNet MSE: {}".format(mse))

Tuned ElasticNet l1 ratio: {'l1\_ratio': 0.20689655172413793}

Tuned ElasticNet R squared: 0.8668305372460283

Tuned ElasticNet MSE: 10.05791413339844

<script.py> output:

Tuned ElasticNet l1 ratio: {'l1\_ratio': 0.20689655172413793}

Tuned ElasticNet R squared: 0.8668305372460283

Tuned ElasticNet MSE: 10.05791413339844

In [6]:

+70 XP

Fantastic! Now that you understand how to fine-tune your models, it's time to learn about preprocessing techniques and how to piece together all the different stages of the machine learning process into a pipeline!

+50 XP [from video]

**Exercise**

**Exercise**

**Exploring categorical features**

The Gapminder dataset that you worked with in previous chapters also contained a categorical 'Region' feature, which we dropped in previous exercises since you did not have the tools to deal with it. Now however, you do, so we have added it back in!

Your job in this exercise is to explore this feature. Boxplots are particularly useful for visualizing categorical features such as this.

**Instructions**

**100 XP**

* Import pandas as pd.
* Read the CSV file 'gapminder.csv' into a DataFrame called df.
* Use pandas to create a boxplot showing the variation of life expectancy ('life') by region ('Region'). To do so, pass the column names in to df.boxplot() (in that order).

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you call pd.read\_csv()?

# Import pandas

import pandas as pd

# Read 'gapminder.csv' into a DataFrame: df

df = pd.read\_csv('gapminder.csv')

# Create a boxplot of life expectancy per region

df.boxplot('life', 'Region', rot=60)

# Show the plot

plt.show()

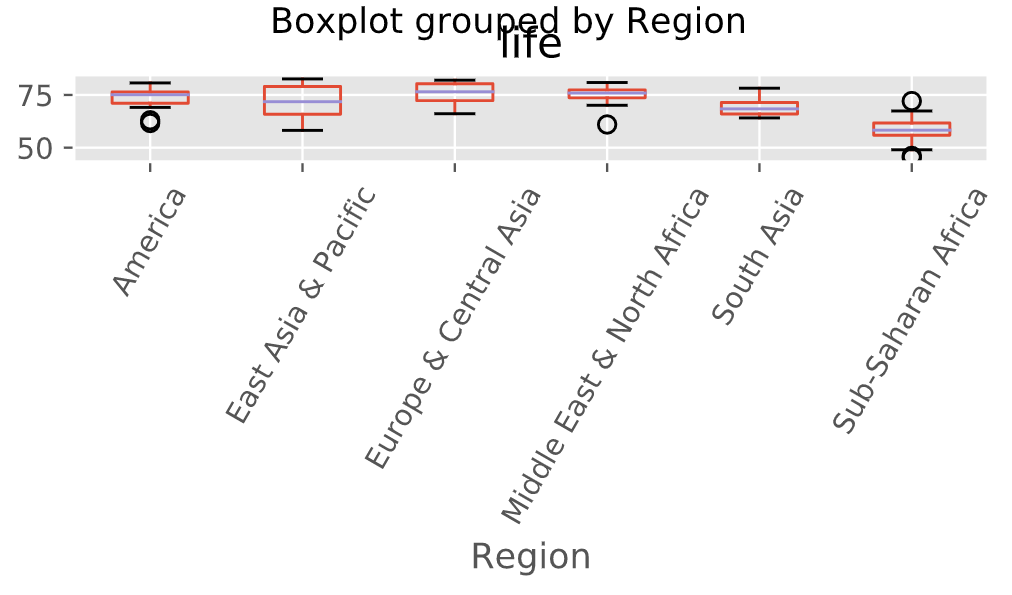
Traceback (most recent call last):

File "script.py", line 5, in <module>

df = pd.read('gapminder.csv')

AttributeError: module 'pandas' has no attribute 'read'

In [1]:



+100 XP

Great work! Exploratory data analysis should always be the precursor to model building.

**Exercise**

**Exercise**

**Creating dummy variables**

As Andy discussed in the video, scikit-learn does not accept non-numerical features. You saw in the previous exercise that the 'Region' feature contains very useful information that can predict life expectancy. For example, Sub-Saharan Africa has a lower life expectancy compared to Europe and Central Asia. Therefore, if you are trying to predict life expectancy, it would be preferable to retain the 'Region' feature. To do this, you need to binarize it by creating dummy variables, which is what you will do in this exercise.

**Instructions**

**100 XP**

* Use the pandas get\_dummies() function to create dummy variables from the df DataFrame. Store the result as df\_region.
* Print the columns of df\_region. This has been done for you.
* Use the get\_dummies() function again, this time specifying drop\_first=True to drop the unneeded dummy variable (in this case, 'Region\_America').
* Hit 'Submit Answer to print the new columns of df\_region and take note of how one column was dropped!

[**Take Hint (-30 XP)**](javascript:void(0))

# Create dummy variables: df\_region

df\_region = pd.get\_dummies(df)

# Print the columns of df\_region

print(df\_region.columns)

# Create dummy variables with drop\_first=True: df\_region

df\_region = pd.get\_dummies(df, drop\_first=True)

# Print the new columns of df\_region

print(df\_region.columns)

<script.py> output:

Index(['population', 'fertility', 'HIV', 'CO2', 'BMI\_male', 'GDP',

'BMI\_female', 'life', 'child\_mortality', 'Region\_America',

'Region\_East Asia & Pacific', 'Region\_Europe & Central Asia',

'Region\_Middle East & North Africa', 'Region\_South Asia',

'Region\_Sub-Saharan Africa'],

dtype='object')

Index(['population', 'fertility', 'HIV', 'CO2', 'BMI\_male', 'GDP',

'BMI\_female', 'life', 'child\_mortality', 'Region\_East Asia & Pacific',

'Region\_Europe & Central Asia', 'Region\_Middle East & North Africa',

'Region\_South Asia', 'Region\_Sub-Saharan Africa'],

dtype='object')

In [1]:

+100 XP

Excellent! Now that you have created the dummy variables, you can use the 'Region' feature to predict life expectancy!

**Exercise**

**Exercise**

**Regression with categorical features**

Having created the dummy variables from the 'Region' feature, you can build regression models as you did before. Here, you'll use ridge regression to perform 5-fold cross-validation.

The feature array X and target variable array y have been pre-loaded.

**Instructions**

**100 XP**

* Import Ridge from sklearn.linear\_model and cross\_val\_score from sklearn.model\_selection.
* Instantiate a ridge regressor called ridge with alpha=0.5 and normalize=True.
* Perform 5-fold cross-validation on X and y using the cross\_val\_score() function.
* Print the cross-validated scores.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Your code can not be executed due to a syntax error:  
invalid syntax (script.py, line 2).

Check your call of cross\_val\_score(). Did you correctly specify the first argument? Expected ridge, but got X.

Check your call of cross\_val\_score(). Did you specify the argument cv?

File "script.py", line 2

import Ridge from sklearn.linear\_model

^

SyntaxError: invalid syntax

Traceback (most recent call last):

File "script.py", line 9, in <module>

ridge\_cv = cross\_val\_score(X, y)

File "script.py", line 335, in cross\_val\_score

scorer = check\_scoring(estimator, scoring=scoring)

File "script.py", line 274, in check\_scoring

"'fit' method, %r was passed" % estimator)

TypeError: estimator should be an estimator implementing 'fit' method, array([[3.4811059e+07, 2.7300000e+00, 1.0000000e-01, ..., 1.0000000e+00,

0.0000000e+00, 0.0000000e+00],

[1.9842251e+07, 6.4300000e+00, 2.0000000e+00, ..., 0.0000000e+00,

0.0000000e+00, 1.0000000e+00],

[4.0381860e+07, 2.2400000e+00, 5.0000000e-01, ..., 0.0000000e+00,

0.0000000e+00, 0.0000000e+00],

...,

[8.6589342e+07, 1.8600000e+00, 4.0000000e-01, ..., 0.0000000e+00,

0.0000000e+00, 0.0000000e+00],

[1.3114579e+07, 5.8800000e+00, 1.3600000e+01, ..., 0.0000000e+00,

0.0000000e+00, 1.0000000e+00],

[1.3495462e+07, 3.8500000e+00, 1.5100000e+01, ..., 0.0000000e+00,

0.0000000e+00, 1.0000000e+00]]) was passed

In [1]: # Import necessary modules

from sklearn.linear\_model import Ridge

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

# Instantiate a ridge regressor: ridge

ridge = Ridge(alpha=0.5, normalize=True)

# Perform 5-fold cross-validation: ridge\_cv

ridge\_cv = cross\_val\_score(ridge, X, y)

# Print the cross-validated scores

print(ridge\_cv)

[0.86136601 0.83522099 0.82255088]

<script.py> output:

[0.86136601 0.83522099 0.82255088]

In [2]: # Import necessary modules

from sklearn.linear\_model import Ridge

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

# Instantiate a ridge regressor: ridge

ridge = Ridge(alpha=0.5, normalize=True)

# Perform 5-fold cross-validation: ridge\_cv

ridge\_cv = cross\_val\_score(ridge, X, y, cv=5)

# Print the cross-validated scores

print(ridge\_cv)

[0.86808336 0.80623545 0.84004203 0.7754344 0.87503712]

<script.py> output:

[0.86808336 0.80623545 0.84004203 0.7754344 0.87503712]

In [3]:

+100 XP

Excellent! You now know how to build models using data that includes categorical features.

**Exercise**

**Exercise**

**Dropping missing data**

The voting dataset from Chapter 1 contained a bunch of missing values that we dealt with for you behind the scenes. Now, it's time for you to take care of these yourself!

The unprocessed dataset has been loaded into a DataFrame df. Explore it in the IPython Shell with the .head() method. You will see that there are certain data points labeled with a '?'. These denote missing values. As you saw in the video, different datasets encode missing values in different ways. Sometimes it may be a '9999', other times a 0 - real-world data can be very messy! If you're lucky, the missing values will already be encoded as NaN. We use NaN because it is an efficient and simplified way of internally representing missing data, and it lets us take advantage of pandas methods such as .dropna() and .fillna(), as well as scikit-learn's Imputation transformer Imputer().

In this exercise, your job is to convert the '?'s to NaNs, and then drop the rows that contain them from the DataFrame.

**Instructions**

**100 XP**

* Explore the DataFrame df in the IPython Shell. Notice how the missing value is represented.
* Convert all '?' data points to np.nan.
* Count the total number of NaNs using the .isnull() and .sum() methods. This has been done for you.
* Drop the rows with missing values from df using .dropna().
* Hit 'Submit Answer' to see how many rows were lost by dropping the missing values.

[**Take Hint (-30 XP)**](javascript:void(0))

# Convert '?' to NaN

df[df == '?'] = np.nan

# Print the number of NaNs

print(df.isnull().sum())

# Print shape of original DataFrame

print("Shape of Original DataFrame: {}".format(df.shape))

# Drop missing values and print shape of new DataFrame

df = df.dropna()

# Print shape of new DataFrame

print("Shape of DataFrame After Dropping All Rows with Missing Values: {}".format(df.shape))

+100 XP

Great work! When many values in your dataset are missing, if you drop them, you may end up throwing away valuable information along with the missing data. It's better instead to develop an imputation strategy. This is where domain knowledge is useful, but in the absence of it, you can impute missing values with the mean or the median of the row or column that the missing value is in.

**Exercise**

**Exercise**

**Imputing missing data in a ML Pipeline I**

As you've come to appreciate, there are many steps to building a model, from creating training and test sets, to fitting a classifier or regressor, to tuning its parameters, to evaluating its performance on new data. Imputation can be seen as the first step of this machine learning process, the entirety of which can be viewed within the context of a pipeline. Scikit-learn provides a pipeline constructor that allows you to piece together these steps into one process and thereby simplify your workflow.

You'll now practice setting up a pipeline with two steps: the imputation step, followed by the instantiation of a classifier. You've seen three classifiers in this course so far: k-NN, logistic regression, and the decision tree. You will now be introduced to a fourth one - the Support Vector Machine, or [**SVM**](http://scikit-learn.org/stable/modules/svm.html). For now, do not worry about how it works under the hood. It works exactly as you would expect of the scikit-learn estimators that you have worked with previously, in that it has the same .fit() and .predict() methods as before.

**Instructions**

**100 XP**

* Import Imputer from sklearn.preprocessing and SVC from sklearn.svm. SVC stands for Support Vector Classification, which is a type of SVM.
* Setup the Imputation transformer to impute missing data (represented as 'NaN') with the 'most\_frequent' value in the column (axis=0).
* Instantiate a SVC classifier. Store the result in clf.
* Create the steps of the pipeline by creating a list of tuples:
  + The first tuple should consist of the imputation step, using imp.
  + The second should consist of the classifier.

[**Take Hint (-30 XP)**](javascript:void(0))

# Import the Imputer module

from sklearn.preprocessing import Imputer

from sklearn.svm import SVC

# Setup the Imputation transformer: imp

imp = Imputer(missing\_values='NaN', strategy='most\_frequent', axis=0)

# Instantiate the SVC classifier: clf

clf = SVC()

# Setup the pipeline with the required steps: steps

steps = [('imputation', imp),

('SVM', clf)]

+100 XP

Fantastic! Having set up the pipeline steps, you can now use it for classification.

**Exercise**

**Exercise**

**Imputing missing data in a ML Pipeline II**

Having setup the steps of the pipeline in the previous exercise, you will now use it on the voting dataset to classify a Congressman's party affiliation. What makes pipelines so incredibly useful is the simple interface that they provide. You can use the .fit() and .predict() methods on pipelines just as you did with your classifiers and regressors!

Practice this for yourself now and generate a classification report of your predictions. The steps of the pipeline have been set up for you, and the feature array X and target variable array y have been pre-loaded. Additionally, train\_test\_split and classification\_report have been imported from sklearn.model\_selection and sklearn.metrics respectively.

**Instructions**

**70 XP**

**Instructions**

**70 XP**

* Import the following modules:
  + Imputer from sklearn.preprocessing and Pipeline from sklearn.pipeline.
  + SVC from sklearn.svm.
* Create the pipeline using Pipeline() and steps.
* Create training and test sets. Use 30% of the data for testing and a random state of 42.
* Fit the pipeline to the training set and predict the labels of the test set.
* Compute the classification report.

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* Use the command from y import x to import x from y.
* To create the pipeline, pass in steps to Pipeline().
* Use the train\_test\_split() function to create the training and test sets. Pass in X and y, and specify the keyword arguments test\_size=0.3 and random\_state=42.
* Use the .fit() method on pipeline with X\_train, y\_train as arguments. Then, use the .predict() method with X\_test as argument to predict the labels.
* To generate the classification report, pass y\_test and y\_pred as arguments to classification\_report().

 Awesome, thanks for your feedback!

**Incorrect Submission**

Check your call of print(). Did you correctly specify the first argument? Did you call classification\_report()?

# Import necessary modules

from sklearn.preprocessing import Imputer

from sklearn.pipeline import Pipeline

from sklearn.svm import SVC

tts, cr = train\_test\_split, classification\_report

# Setup the pipeline steps: steps

steps = [('imputation', Imputer(missing\_values='NaN', strategy='most\_frequent', axis=0)),

('SVM', SVC())]

# Create the pipeline: pipeline

pipeline = Pipeline(steps)

# Create training and test sets

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size = 0.3, random\_state=42)

# Fit the pipeline to the train set

pipeline.fit(X\_train, y\_train)

# Predict the labels of the test set

y\_pred = pipeline.predict(X\_test)

# Compute metrics

print(cr(y\_test, y\_pred))

# print(classification\_report(y\_test, y\_pred))

In [1]: train\_test\_split

Out[1]: <function sklearn.model\_selection.\_split.train\_test\_split>

In [2]: tts = train\_test\_split

In [3]: X, y, test\_size = 0.2, random\_state=42

File "<stdin>", line 1

X, y, test\_size = 0.2, random\_state=42

^

SyntaxError: can't assign to literal

In [4]: classification\_report

Out[4]: <function sklearn.metrics.classification.classification\_report>

<script.py> output:

precision recall f1-score support

democrat 0.99 0.96 0.98 85

republican 0.94 0.98 0.96 46

avg / total 0.97 0.97 0.97 131

<script.py> output:

precision recall f1-score support

democrat 0.96 0.99 0.98 83

republican 0.98 0.94 0.96 48

avg / total 0.97 0.97 0.97 131

<script.py> output:

precision recall f1-score support

democrat 0.99 0.96 0.98 85

republican 0.94 0.98 0.96 46

avg / total 0.97 0.97 0.97 131

In [5]:

+70 XP

Great work! Your pipeline has performed imputation as well as classification!

**Exercise**

**Exercise**

**Centering and scaling your data**

In the video, Hugo demonstrated how significantly the performance of a model can improve if the features are scaled. Note that this is not always the case: In the Congressional voting records dataset, for example, all of the features are binary. In such a situation, scaling will have minimal impact.

You will now explore scaling for yourself on a new dataset - [**White Wine Quality**](https://archive.ics.uci.edu/ml/datasets/Wine+Quality)! Hugo used the Red Wine Quality dataset in the video. We have used the 'quality' feature of the wine to create a binary target variable: If 'quality' is less than 5, the target variable is 1, and otherwise, it is 0.

The DataFrame has been pre-loaded as df, along with the feature and target variable arrays X and y. Explore it in the IPython Shell. Notice how some features seem to have different units of measurement. 'density', for instance, takes values between 0.98 and 1.04, while 'total sulfur dioxide' ranges from 9 to 440. As a result, it may be worth scaling the features here. Your job in this exercise is to scale the features and compute the mean and standard deviation of the unscaled features compared to the scaled features.

**Instructions**

**100 XP**

* Import scale from sklearn.preprocessing.
* Scale the features X using scale().
* Print the mean and standard deviation of the unscaled features X, and then the scaled features X\_scaled. Use the numpy functions np.mean() and np.std() to compute the mean and standard deviations.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you used print("Mean of Scaled Features: {}".format(np.mean(X\_scaled))) to do the appropriate printouts?

# Import scale

from sklearn.preprocessing import scale

# Scale the features: X\_scaled

X\_scaled = scale(X)

# Print the mean and standard deviation of the unscaled features

print("Mean of Unscaled Features: {}".format(np.mean(X)))

print("Standard Deviation of Unscaled Features: {}".format(np.std(X)))

# Print the mean and standard deviation of the scaled features

print("Mean of Scaled Features: {}".format(np.mean(X\_scaled)))

print("Standard Deviation of Scaled Features: {}".format(np.std(X\_scaled)))

<script.py> output:

Mean of Unscaled Features: 18.432687072460002

Standard Deviation of Unscaled Features: 41.54494764094571

Mean of Unscaled Features: 2.7314972981668206e-15

Standard Deviation of Unscaled Features: 0.9999999999999999

<script.py> output:

Mean of Unscaled Features: 18.432687072460002

Standard Deviation of Unscaled Features: 41.54494764094571

Mean of Scaled Features: 2.7314972981668206e-15

Standard Deviation of Scaled Features: 0.9999999999999999

In [1]:

+100 XP

Great work! Notice the difference in the mean and standard deviation of the scaled features compared to the unscaled features.

**Exercise**

**Exercise**

**Centering and scaling in a pipeline**

With regard to whether or not scaling is effective, the proof is in the pudding! See for yourself whether or not scaling the features of the White Wine Quality dataset has any impact on its performance. You will use a k-NN classifier as part of a pipeline that includes scaling, and for the purposes of comparison, a k-NN classifier trained on the unscaled data has been provided.

The feature array and target variable array have been pre-loaded as X and y. Additionally, KNeighborsClassifier and train\_test\_split have been imported from sklearn.neighbors and sklearn.model\_selection, respectively.

**Instructions**

**70 XP**

**Instructions**

**70 XP**

* Import the following modules:
  + StandardScaler from sklearn.preprocessing.
  + Pipeline from sklearn.pipeline.
* Complete the steps of the pipeline with StandardScaler() for 'scaler' and KNeighborsClassifier() for 'knn'.
* Create the pipeline using Pipeline() and steps.
* Create training and test sets, with 30% used for testing. Use a random state of 42.
* Fit the pipeline to the training set.
* Compute the accuracy scores of the scaled and unscaled models by using the .score() method inside the provided print() functions.

[**Show Answer (-70 XP)**](javascript:void(0))

**Hint**

* Use the command from y import x to import x from y.
* The two tuples needed in the pipeline steps are ('scaler', StandardScaler()), and ('knn', KNeighborsClassifier()).
* To create the pipeline, pass in steps to Pipeline().
* Use the train\_test\_split() function to create the training and test sets. Pass in X and y, and specify the keyword arguments test\_size=0.3 and random\_state=42.
* Use the .fit() method on pipeline with X\_train, y\_train as arguments.
* Use the .score() method on knn\_scaled and knn\_unscaled with X\_test and y\_test as arguments to compute the accuracy scores.

Did you find this hint helpful?

[YesNo](javascript:void(0))

# Import the necessary modules

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

tts = train\_test\_split

#tts, cr = train\_test\_split, classification\_report

# Setup the pipeline steps: steps

steps = [('scaler', StandardScaler()),

('knn', KNeighborsClassifier())]

# Create the pipeline: pipeline

pipeline = Pipeline(steps)

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size = 0.3, random\_state=42)

# Fit the pipeline to the training set: knn\_scaled

knn\_scaled = pipeline.fit(X\_train, y\_train)

# Instantiate and fit a k-NN classifier to the unscaled data

knn\_unscaled = KNeighborsClassifier().fit(X\_train, y\_train)

# Compute and print metrics

print('Accuracy with Scaling: {}'.format(knn\_scaled.score(X\_test, y\_test)))

print('Accuracy without Scaling: {}'.format(knn\_unscaled.score(X\_test, y\_test)))

**Incorrect Submission**

Did you define the variable steps without errors?

Traceback (most recent call last):

File "script.py", line 5, in <module>

tts, cr = train\_test\_split, classification\_report

NameError: name 'classification\_report' is not defined

<script.py> output:

Accuracy with Scaling: 0.7700680272108843

Accuracy without Scaling: 0.6979591836734694

In [1]:

+70 XP

Fantastic! It looks like scaling has significantly improved model performance!

**Exercise**

**Exercise**

**Bringing it all together I: Pipeline for classification**

It is time now to piece together everything you have learned so far into a pipeline for classification! Your job in this exercise is to build a pipeline that includes scaling and hyperparameter tuning to classify wine quality.

You'll return to using the SVM classifier you were briefly introduced to earlier in this chapter. The hyperparameters you will tune are CC and gammagamma. CC controls the regularization strength. It is analogous to the CC you tuned for logistic regression in Chapter 3, while gammagamma controls the kernel coefficient: Do not worry about this now as it is beyond the scope of this course.

The following modules and functions have been pre-loaded: Pipeline, SVC, train\_test\_split, GridSearchCV, classification\_report, accuracy\_score. The feature and target variable arrays X and y have also been pre-loaded.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Setup the pipeline with the following steps:
  + Scaling, called 'scaler' with StandardScaler().
  + Classification, called 'SVM' with SVC().
* Specify the hyperparameter space using the following notation: 'step\_name\_\_parameter\_name'. Here, the step\_name is SVM, and the parameter\_names are C and gamma.
* Create training and test sets, with 20% of the data used for the test set. Use a random state of 21.
* Instantiate GridSearchCV with the pipeline and hyperparameter space and fit it to the training set. Use 3-fold cross-validation (This is the default, so you don't have to specify it).
* Predict the labels of the test set and compute the metrics. The metrics have been computed for you.

[**Take Hint (-30 XP)**](javascript:void(0))

# Setup the pipeline

steps = [('scaler', StandardScaler()),

('SVM', SVC())]

pipeline = Pipeline(steps)

# Specify the hyperparameter space

parameters = {'SVM\_\_C':[1, 10, 100],

'SVM\_\_gamma':[0.1, 0.01]}

# Create train and test sets

#X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size = 0.2, random\_state=21)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state=21)

# Instantiate the GridSearchCV object: cv

cv = GridSearchCV()

# Fit to the training set

cv.fit(X\_train, y\_train)

# Predict the labels of the test set: y\_pred

y\_pred = cv.predict(X\_test)

# Compute and print metrics

print("Accuracy: {}".format(cv.score(X\_test, y\_test)))

print(classification\_report(y\_test, y\_pred))

print("Tuned Model Parameters: {}".format(cv.best\_params\_))

**Incorrect Submission**

Did you call train\_test\_split()?

Have you specified the arguments for GridSearchCV() using the right syntax?

Traceback (most recent call last):

File "script.py", line 12, in <module>

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size = 0.2, random\_state=21)

NameError: name 'tts' is not defined

Traceback (most recent call last):

File "script.py", line 15, in <module>

cv = GridSearchCV()

TypeError: \_\_init\_\_() missing 2 required positional arguments: 'estimator' and 'param\_grid'

Traceback (most recent call last):

File "script.py", line 15, in <module>

cv = GridSearchCV(3)

TypeError: \_\_init\_\_() missing 1 required positional argument: 'param\_grid'

Traceback (most recent call last):

File "script.py", line 15, in <module>

cv = GridSearchCV(cv=3)

TypeError: \_\_init\_\_() missing 2 required positional arguments: 'estimator' and 'param\_grid'

In [1]: GridSearchCV

Out[1]: sklearn.model\_selection.\_search.GridSearchCV

<script.py> output:

Accuracy: 0.7795918367346939

precision recall f1-score support

False 0.83 0.85 0.84 662

True 0.67 0.63 0.65 318

avg / total 0.78 0.78 0.78 980

Tuned Model Parameters: {'SVM\_\_C': 10, 'SVM\_\_gamma': 0.1}

In [2]:

+100 XP

Excellent!

**Exercise**

**Exercise**

**Bringing it all together II: Pipeline for regression**

For this final exercise, you will return to the Gapminder dataset. Guess what? Even this dataset has missing values that we dealt with for you in earlier chapters! Now, you have all the tools to take care of them yourself!

Your job is to build a pipeline that imputes the missing data, scales the features, and fits an ElasticNet to the Gapminder data. You will then tune the l1\_ratio of your ElasticNet using GridSearchCV.

All the necessary modules have been imported, and the feature and target variable arrays have been pre-loaded as X and y.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Set up a pipeline with the following steps:
  + 'imputation', which uses the Imputer() transformer and the 'mean' strategy to impute missing data ('NaN') using the mean of the column.
  + 'scaler', which scales the features using StandardScaler().
  + 'elasticnet', which instantiates an ElasticNet() regressor.
* Specify the hyperparameter space for the l1l1 ratio using the following notation: 'step\_name\_\_parameter\_name'. Here, the step\_name is elasticnet, and the parameter\_name is l1\_ratio.
* Create training and test sets, with 40% of the data used for the test set. Use a random state of 42.
* Instantiate GridSearchCV with the pipeline and hyperparameter space. Use 3-fold cross-validation (This is the default, so you don't have to specify it).
* Fit the GridSearchCV object to the training set.
* Compute R2R2 and the best parameters. This has been done for you, so hit 'Submit Answer' to see the results!

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Have you specified the arguments for GridSearchCV() using the right syntax?

# Setup the pipeline steps: steps

steps = [('imputation', Imputer(missing\_values='NaN', strategy='mean', axis=0)),

('scaler', StandardScaler()),

('elasticnet', ElasticNet())]

# Create the pipeline: pipeline

pipeline = Pipeline(steps)

# Specify the hyperparameter space

parameters = {'elasticnet\_\_l1\_ratio':np.linspace(0,1,30)}

# Create train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.4, random\_state=42)

# Create the GridSearchCV object: gm\_cv

gm\_cv = GridSearchCV(pipeline, parameters) #cv=3

# Fit to the training set

gm\_cv.fit(X\_train, y\_train)

# Compute and print the metrics

r2 = gm\_cv.score(X\_test, y\_test)

print("Tuned ElasticNet Alpha: {}".format(gm\_cv.best\_params\_))

print("Tuned ElasticNet R squared: {}".format(r2))

Traceback (most recent call last):

File "script.py", line 16, in <module>

gm\_cv = GridSearchCV()

TypeError: \_\_init\_\_() missing 2 required positional arguments: 'estimator' and 'param\_grid'

<script.py> output:

Tuned ElasticNet Alpha: {'elasticnet\_\_l1\_ratio': 1.0}

Tuned ElasticNet R squared: 0.8862016570888217

In [1]:

+100 XP

Fantastic work! You have now mastered the fundamentals of supervised learning with scikit-learn!