**Course Description**



Grow your machine learning skills with scikit-learn and discover how to use this popular Python library to train models using labeled data. In this course, you'll learn how to make powerful predictions, such as whether a customer is will churn from your business, whether an individual has diabetes, and even how to tell classify the genre of a song. Using real-world datasets, you'll find out how to build predictive models, tune their parameters, and determine how well they will perform with unseen data.

1

#### Classification

0%

In this chapter, you'll be introduced to classification problems and learn how to solve them using supervised learning techniques. You'll learn how to split data into training and test sets, fit a model, make predictions, and evaluate accuracy. You’ll discover the relationship between model complexity and performance, applying what you learn to a churn dataset, where you will classify the churn status of a telecom company's customers.

**Transcript**

## 1. Machine learning with scikit-learn

00:00 - 00:08

Hi, and welcome! My name is George Boorman, and I'll be your instructor for this course on supervised learning with scikit-learn.

## 2. What is machine learning?

00:08 - 00:16

Machine learning is the process whereby computers learn to make decisions from data without being explicitly programmed.

## 3. Examples of machine learning

00:16 - 00:33

For example, learning to predict whether an email is spam or not spam given its content and sender. Or learning to cluster books into different categories based on the words they contain, then assigning any new book to one of the existing clusters.

## 4. Unsupervised learning

00:33 - 00:54

Unsupervised learning is the process of uncovering hidden patterns and structures from unlabeled data. For example, a business may wish to group its customers into distinct categories based on their purchasing behavior without knowing in advance what these categories are. This is known as clustering, one branch of unsupervised learning.

## 5. Supervised learning

00:54 - 01:20

Supervised learning is a type of machine learning where the values to be predicted are already known, and a model is built with the aim of accurately predicting values of previously unseen data. Supervised learning uses features to predict the value of a target variable, such as predicting a basketball player's position based on their points per game. This course will exclusively focus on supervised learning.

## 6. Types of supervised learning

01:20 - 01:55

There are two types of supervised learning. Classification is used to predict the label, or category, of an observation. For example, we can predict whether a bank transaction is fraudulent or not. As there are two outcomes here - a fraudulent transaction, or non-fraudulent transaction, this is known as binary classification. Regression is used to predict continuous values. For example, a model can use features such as number of bedrooms, and the size of a property, to predict the target variable, price of the property.

## 7. Naming conventions

01:55 - 02:09

Note that what we call a feature throughout the course, others may call a predictor variable or independent variable. Also, what we call the target variable, others may call dependent variable or response variable.

## 8. Before you use supervised learning

02:09 - 02:37

There are some requirements to satisfy before performing supervised learning. Our data must not have missing values, must be in numeric format, and stored as pandas DataFrames or Series, or NumPy arrays. This requires some exploratory data analysis first to ensure data is in the correct format. Various pandas methods for descriptive statistics, along with appropriate data visualizations, are useful in this step.

## 9. scikit-learn syntax

02:37 - 03:49

scikit-learn follows the same syntax for all supervised learning models, which makes the workflow repeatable. Let's familiarize ourselves with the general scikit-learn workflow syntax, before we explore using real data later in the chapter. We import a Model, which is a type of algorithm for our supervised learning problem, from an sklearn module. For example, the k-Nearest Neighbors model uses distance between observations to predict labels or values. We create a variable named model, and instantiate the Model. A model is fit to the data, where it learns patterns about the features and the target variable. We fit the model to X, an array of our features, and y, an array of our target variable values. We then use the model's dot-predict method, passing six new observations, X\_new. For example, if feeding features from six emails to a spam classification model, an array of six values is returned. A one indicates the model predicts that email is spam, and a zero represents a prediction of not spam.

## 10. Let's practice!

03:49 - 03:56

Now let's check our understanding of supervised learning and how to perform it in scikit-learn.

# Binary classification

In the video, you saw that there are two types of supervised learning — classification and regression. Recall that binary classification is used to predict a target variable that has only two labels, typically represented numerically with a zero or a one.

A dataset, churn\_df, has been preloaded for you in the console.

Your task is to examine the data and choose which column could be the target variable for binary classification.

##### Instructions

* "customer\_service\_calls"
* "total\_night\_charge"
* **"churn"**
* "account\_length"

Correct! churn has values of 0 or 1, so it can be predicted using a binary classification model.

## 1. The classification challenge

00:00 - 00:12

Previously, we learned that supervised learning uses labels. Let's discuss how we can build a classification model, or classifier, to predict the labels of unseen data.

## 2. Classifying labels of unseen data

00:12 - 00:32

There are four steps. First, we build a classifier, which learns from the labeled data we pass to it. We then pass it unlabeled data as input, and have it predict labels for this unseen data. As the classifier learns from the labeled data, we call this the training data.

## 3. k-Nearest Neighbors

00:32 - 01:05

Let's build our first model! We'll use an algorithm called k-Nearest Neighbors, which is popular for classification problems. The idea of k-Nearest Neighbors, or KNN, is to predict the label of any data point by looking at the k, for example, three, closest labeled data points and getting them to vote on what label the unlabeled observation should have. KNN uses majority voting, which makes predictions based on what label the majority of nearest neighbors have.

## 4. k-Nearest Neighbors

01:05 - 01:11

Using this scatter plot as an example, how do we classify the black observation?

## 5. k-Nearest Neighbors

01:11 - 01:18

If k equals three, we would classify it as red. This is because two of the three closest observations are red.

## 6. k-Nearest Neighbors

01:18 - 01:22

If k equals five, we would instead classify it as blue.

## 7. KNN Intuition

01:22 - 01:40

To build intuition for KNN, let's look at this scatter plot displaying total evening charge against total day charge for customers of a telecom company. The observations are colored in blue for customers who have churned, and red for those who have not churned.

## 8. KNN Intuition

01:40 - 02:07

Here we have visualized the results of a KNN algorithm where the number of neighbors is set to 15. KNN creates a decision boundary to predict if customers will churn. Any customers in the area with a gray background are predicted to churn, and those in the area with a red background are predicted to not churn. This boundary would be used to make predictions on unseen data.

## 9. Using scikit-learn to fit a classifier

02:07 - 03:26

To fit a KNN model using scikit-learn, we import KNeighborsClassifier from sklearn-dot-neighbors. We split our data into X, a 2D array of our features, and y, a 1D array of the target values - in this case, churn status. scikit-learn requires that the features are in an array where each column is a feature and each row a different observation. Similarly, the target needs to be a single column with the same number of observations as the feature data. We use the dot-values attribute to convert X and y to NumPy arrays. Printing the shape of X and y, we see there are 3333 observations of two features, and 3333 observations of the target variable. We then instantiate our KNeighborsClassifier, setting n\_neighbors equal to 15, and assign it to the variable knn. Then we can fit this classifier to our labeled data by applying the classifier's dot-fit method and passing two arguments: the feature values, X, and the target values, y.

## 10. Predicting on unlabeled data

03:26 - 04:08

Here we have a set of new observations, X\_new. Checking the shape of X\_new, we see it has three rows and two columns, that is, three observations and two features. We use the classifier's dot-predict method and pass it the unseen data as a 2D NumPy array containing features in columns and observations in rows. Printing the predictions returns a binary value for each observation or row in X\_new. It predicts 1, which corresponds to 'churn', for the first observation, and 0, which corresponds to 'no churn', for the second and third observations.

## 11. Let's practice!

04:08 - 04:17

Now let's build our own KNN model for the churn dataset, which we will use for the remainder of the chapter!

**Daily XP250**

##### Exercise

##### Exercise

# k-Nearest Neighbors: Fit

In this exercise, you will build your first classification model using the churn\_df dataset, which has been preloaded for the remainder of the chapter.

The features to use will be "account\_length" and "customer\_service\_calls". The target, "churn", needs to be a single column with the same number of observations as the feature data.

You will convert the features and the target variable into NumPy arrays, create an instance of a KNN classifier, and then fit it to the data.

numpy has also been preloaded for you as np.

##### Instructions

**100 XP**

* Import KNeighborsClassifier from sklearn.neighbors.
* Create an array called X containing values from the "account\_length" and "customer\_service\_calls" columns, and an array called y for the values of the "churn" column.
* Instantiate a KNeighborsClassifier called knn with 6 neighbors.
* Fit the classifier to the data using the .fit() method.
* # Import KNeighborsClassifier
* from \_\_\_\_.\_\_\_\_ import \_\_\_\_
* # Create arrays for the features and the target variable
* y = \_\_\_\_["\_\_\_\_"].values
* X = \_\_\_\_[["\_\_\_\_", "\_\_\_\_"]].values
* # Create a KNN classifier with 6 neighbors
* knn = \_\_\_\_
* # Fit the classifier to the data
* knn.\_\_\_\_(\_\_\_\_, \_\_\_\_)

Excellent! Now that your KNN classifier has been fit to the data, it can be used to predict the labels of new data points.

# Import KNeighborsClassifier

from sklearn.neighbors import KNeighborsClassifier

# Create arrays for the features and the target variable

y = churn\_df["churn"].values

X = churn\_df[["account\_length", "customer\_service\_calls"]].values

# Create a KNN classifier with 6 neighbors

knn = KNeighborsClassifier(n\_neighbors=6)

# Fit the classifier to the data

knn.fit(X, y)

# Import KNeighborsClassifier

from sklearn.neighbors import KNeighborsClassifier

# Create arrays for the features and the target variable

y = churn\_df["churn"].values

X = churn\_df[["account\_length", "customer\_service\_calls"]].values

# Create a KNN classifier with 6 neighbors

knn = KNeighborsClassifier(n\_neighbors=6)

# Fit the classifier to the data

knn.fit(X, y)

KNeighborsClassifier(n\_neighbors=6)

Excellent! Now that your KNN classifier has been fit to the data, it can be used to predict the labels of new data points.

# k-Nearest Neighbors: Predict

Now you have fit a KNN classifier, you can use it to predict the label of new data points. All available data was used for training, however, fortunately, there are new observations available. These have been preloaded for you as X\_new.

The model knn, which you created and fit the data in the last exercise, has been preloaded for you. You will use your classifier to predict the labels of a set of new data points:

X\_new = np.array([[30.0, 17.5],

[107.0, 24.1],

[213.0, 10.9]])

##### Instructions

**100 XP**

* Create y\_pred by predicting the target values of the unseen features X\_new.
* Print the predicted labels for the set of predictions.
* # Predict the labels for the X\_new
* y\_pred = \_\_\_\_
* # Print the predictions for X\_new
* print("Predictions: {}".format(\_\_\_\_))

# Predict the labels for the X\_new

y\_pred = knn.predict(X\_new)

# Print the predictions for X\_new

print("Predictions: {}".format(y\_pred))

# Predict the labels for the X\_new

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

# Print the predictions for X\_new

print("Predictions: {}".format(y\_pred))

Predictions: [0 1 0]

<script.py> output:

Predictions: [0 1 0]

Great work! The model has predicted the first and third customers will not churn in the new array. But how do we know how accurate these predictions are? Let's explore how to measure a model's performance in the next video.

## 1. Measuring model performance

00:00 - 00:09

Now we can make predictions using a classifier, but how do we know if the model is making correct predictions? We can evaluate its performance!

## 2. Measuring model performance

00:09 - 00:20

In classification, accuracy is a commonly-used metric. Accuracy is the number of correct predictions divided by the total number of observations.

## 3. Measuring model performance

00:20 - 00:37

How do we measure accuracy? We could compute accuracy on the data used to fit the classifier. However, as this data was used to train the model, performance will not be indicative of how well it can generalize to unseen data, which is what we are interested in!

## 4. Computing accuracy

00:37 - 00:42

It is common to split data into a training set and a test set.

## 5. Computing accuracy

00:42 - 00:45

We fit the classifier using the training set,

## 6. Computing accuracy

00:45 - 00:51

then we calculate the model's accuracy against the test set's labels.

## 7. Train/test split

00:51 - 02:24

To do this, we import train\_test\_split from sklearn-dot-model\_selection. We call train\_test\_split, passing our features and targets. We commonly use 20-30% of our data as the test set. By setting the test\_size argument to zero-point-three we use 30% here. The random\_state argument sets a seed for a random number generator that splits the data. Using the same number when repeating this step allows us to reproduce the exact split and our downstream results. It is best practice to ensure our split reflects the proportion of labels in our data. So if churn occurs in 10% of observations, we want 10% of labels in our training and test sets to represent churn. We achieve this by setting stratify equal to y. train\_test\_split returns four arrays: the training data, the test data, the training labels, and the test labels. We unpack these into X\_train, X\_test, y\_train, and y\_test, respectively. We then instantiate a KNN model and fit it to the training data using the dot-fit method. To check the accuracy, we use the dot-score method, passing X test and y test. The accuracy of our model is 88%, which is low given our labels have a 9 to 1 ratio.

## 8. Model complexity

02:24 - 03:05

Let's discuss how to interpret k. Recall that we discussed decision boundaries, which are thresholds for determining what label a model assigns to an observation. In the image shown, as k increases, the decision boundary is less affected by individual observations, reflecting a simpler model. Simpler models are less able to detect relationships in the dataset, which is known as underfitting. In contrast, complex models can be sensitive to noise in the training data, rather than reflecting general trends. This is known as overfitting.

## 9. Model complexity and over/underfitting

03:05 - 03:54

We can also interpret k using a model complexity curve. With a KNN model, we can calculate accuracy on the training and test sets using incremental k values, and plot the results. We create empty dictionaries to store our train and test accuracies, and an array containing a range of k values. We use a for loop to repeat our previous workflow, building several models using a different number of neighbors. We loop through our neighbors array and, inside the loop, we instantiate a KNN model with n\_neighbors equal to the neighbor iterator, and fit to the training data. We then calculate training and test set accuracy, storing the results in their respective dictionaries.

## 10. Plotting our results

03:54 - 04:01

After our for loop, we then plot the training and test values, including a legend and labels.

## 11. Model complexity curve

04:01 - 04:14

Here's the result! As k increases beyond 15 we see overfitting where performance plateaus on both test and training sets, as indicated in this plot.

## 12. Model complexity curve

04:14 - 04:19

The peak test accuracy actually occurs at around 13 neighbors.

## 13. Let's practice!

04:19 - 04:27

Now let's practice splitting data, computing accuracy, and plotting model complexity curves!

# Train/test split + computing 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 churn\_df dataset!

NumPy arrays have been created for you containing the features as X and the target variable as y. You will split them into training and test sets, fit a KNN classifier to the training data, and then compute its accuracy on the test data using the .score() method.

##### Instructions

**100 XP**

* Import train\_test\_split from sklearn.model\_selection.
* Split X and y into training and test sets, setting test\_size equal to 20%, random\_state to 42, and ensuring the target label proportions reflect that of the original dataset.
* Fit the knn model to the training data.
* Compute and print the model's accuracy for the test data.
* # Import the module
* \_\_\_\_
* X = churn\_df.drop("churn", axis=1).values
* y = churn\_df["churn"].values
* # Split into training and test sets
* X\_train, X\_test, y\_train, y\_test = \_\_\_\_(\_\_\_\_, \_\_\_\_, test\_size=\_\_\_\_, random\_state=\_\_\_\_, stratify=\_\_\_\_)
* knn = KNeighborsClassifier(n\_neighbors=5)
* # Fit the classifier to the training data
* \_\_\_\_
* # Print the accuracy
* print(knn.score(\_\_\_\_, \_\_\_\_))

# Import the module

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

X = churn\_df.drop("churn", axis=1).values

y = churn\_df["churn"].values

# Split into training and test sets

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

knn = KNeighborsClassifier(n\_neighbors=5)

# Fit the classifier to the training data

knn.fit(X\_train, y\_train)

# Print the accuracy

print(knn.score(X\_test, y\_test))

# Import the module

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

X = churn\_df.drop("churn", axis=1).values

y = churn\_df["churn"].values

# Split into training and test sets

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

knn = KNeighborsClassifier(n\_neighbors=5)

# Fit the classifier to the training data

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

# Print the accuracy

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

0.8740629685157422

<script.py> output:

0.8740629685157422

Excellent! In a few lines of code you split a dataset, fit a KNN model, and found its accuracy to be 87%!

# Overfitting and underfitting

Interpreting model complexity is a great way to evaluate performance when utilizing supervised learning. Your aim is to produce a model that can interpret the relationship between features and the target variable, as well as generalize well when exposed to new observations.

You will generate accuracy scores for the training and test sets using a KNN classifier with different n\_neighbor values, which you will plot in the next exercise.

The training and test sets have been created from the churn\_df dataset and preloaded as X\_train, X\_test, y\_train, and y\_test.

In addition, KNeighborsClassifier has been imported for you along with numpy as np.

##### Instructions

**100 XP**

* Create neighbors as a numpy array of values from 1 up to and including 12.
* Instantiate a KNN classifier, with the number of neighbors equal to the neighbor iterator.
* Fit the model to the training data.
* Calculate accuracy scores for the training set and test set separately using the .score() method, and assign the results to the index of the train\_accuracies and test\_accuracies dictionaries, respectively.
* # Create neighbors
* neighbors = np.arange(\_\_\_\_, \_\_\_\_)
* train\_accuracies = {}
* test\_accuracies = {}
* for neighbor in neighbors:
* # Set up a KNN Classifier
* knn = \_\_\_\_(\_\_\_\_=\_\_\_\_)
* # Fit the model
* knn.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* # Compute accuracy
* train\_accuracies[\_\_\_\_] = knn.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* test\_accuracies[\_\_\_\_] = knn.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* print(neighbors, '\n', train\_accuracies, '\n', test\_accuracies)

# Create neighbors

neighbors = np.arange(1, 13)

train\_accuracies = {}

test\_accuracies = {}

for neighbor in neighbors:

    # Set up a KNN Classifier

    knn = KNeighborsClassifier(n\_neighbors=neighbor)

    # Fit the model

    knn.fit(X\_train, y\_train)

    # Compute accuracy

    train\_accuracies[neighbor] = knn.score(X\_train, y\_train)

    test\_accuracies[neighbor] = knn.score(X\_test, y\_test)

print(neighbors, '\n', train\_accuracies, '\n', test\_accuracies)

# Create neighbors

neighbors = np.arange(1, 13)

train\_accuracies = {}

test\_accuracies = {}

for neighbor in neighbors:

# Set up a KNN Classifier

knn = KNeighborsClassifier(n\_neighbors=neighbor)

# Fit the model

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

# Compute accuracy

train\_accuracies[neighbor] = knn.score(X\_train, y\_train)

test\_accuracies[neighbor] = knn.score(X\_test, y\_test)

print(neighbors, '\n', train\_accuracies, '\n', test\_accuracies)

[ 1 2 3 4 5 6 7 8 9 10 11 12]

{1: 1.0, 2: 0.887943971985993, 3: 0.9069534767383692, 4: 0.8734367183591796, 5: 0.8829414707353677, 6: 0.8689344672336168, 7: 0.8754377188594297, 8: 0.8659329664832416, 9: 0.8679339669834918, 10: 0.8629314657328664, 11: 0.864432216108054, 12: 0.8604302151075538}

{1: 0.7871064467766117, 2: 0.8500749625187406, 3: 0.8425787106446777, 4: 0.856071964017991, 5: 0.8553223388305847, 6: 0.861319340329835, 7: 0.863568215892054, 8: 0.8605697151424287, 9: 0.8620689655172413, 10: 0.8598200899550225, 11: 0.8598200899550225, 12: 0.8590704647676162}

<script.py> output:

[ 1 2 3 4 5 6 7 8 9 10 11 12]

{1: 1.0, 2: 0.887943971985993, 3: 0.9069534767383692, 4: 0.8734367183591796, 5: 0.8829414707353677, 6: 0.8689344672336168, 7: 0.8754377188594297, 8: 0.8659329664832416, 9: 0.8679339669834918, 10: 0.8629314657328664, 11: 0.864432216108054, 12: 0.8604302151075538}

{1: 0.7871064467766117, 2: 0.8500749625187406, 3: 0.8425787106446777, 4: 0.856071964017991, 5: 0.8553223388305847, 6: 0.861319340329835, 7: 0.863568215892054, 8: 0.8605697151424287, 9: 0.8620689655172413, 10: 0.8598200899550225, 11: 0.8598200899550225, 12: 0.8590704647676162}

Notice how training accuracy decreases as the number of neighbors initially gets larger, and vice versa for the testing accuracy? These scores would be much easier to interpret in a line plot, so let's produce a model complexity curve of these results.

# Visualizing model complexity

Now you have calculated the accuracy of the KNN model on the training and test sets using various values of n\_neighbors, you can create a model complexity curve to visualize how performance changes as the model becomes less complex!

The variables neighbors, train\_accuracies, and test\_accuracies, which you generated in the previous exercise, have all been preloaded for you. You will plot the results to aid in finding the optimal number of neighbors for your model.

##### Instructions

**100 XP**

* Add a title "KNN: Varying Number of Neighbors".
* Plot the .values() method of train\_accuracies on the y-axis against neighbors on the x-axis, with a label of "Training Accuracy".
* Plot the .values() method of test\_accuracies on the y-axis against neighbors on the x-axis, with a label of "Testing Accuracy".
* Display the plot.
* # Add a title
* plt.title("\_\_\_\_")
* # Plot training accuracies
* plt.plot(\_\_\_\_, \_\_\_\_, label="\_\_\_\_")
* # Plot test accuracies
* plt.plot(\_\_\_\_, \_\_\_\_, label="\_\_\_\_")
* plt.legend()
* plt.xlabel("Number of Neighbors")
* plt.ylabel("Accuracy")
* # Display the plot
* \_\_\_\_

# Add a title

plt.title("KNN: Varying Number of Neighbors")

# Plot training accuracies

plt.plot(neighbors, train\_accuracies.values(), label="Training Accuracy")

# Plot test accuracies

plt.plot(neighbors, test\_accuracies.values(), label="Testing Accuracy")

plt.legend()

plt.xlabel("Number of Neighbors")

plt.ylabel("Accuracy")

# Display the plot

plt.show()

# Add a title plt.title("KNN: Varying Number of Neighbors") # Plot training accuracies plt.plot(neighbors, train\_accuracies.values(), label="Training Accuracy") # Plot test accuracies plt.plot(neighbors, test\_accuracies.values(), label="Test Accuracy") plt.legend() plt.xlabel("Number of Neighbors") plt.ylabel("Accuracy") # Display the plot plt.show()

Great work! See how training accuracy decreases and test accuracy increases as the number of neighbors gets larger. For the test set, accuracy peaks with 7 neighbors, suggesting it is the optimal value for our model. Now let's explore regression models!

**Transcript**

## 1. Introduction to regression

00:00 - 00:14

Now we're going to check out the other type of supervised learning: regression. In regression tasks, the target variable typically has continuous values, such as a country's GDP, or the price of a house.

## 2. Predicting blood glucose levels

00:14 - 00:47

To conceptualize regression problems, let's use a dataset containing women's health data to predict blood glucose levels. We load the dataset as a pandas DataFrame, and print the first five rows. It contains features including number of pregnancies, triceps skinfold measurements, insulin levels, body mass index, known as BMI, age in years, and diabetes status, with one indicating a diagnosis, and zero representing the absence of a diagnosis.

## 3. Creating feature and target arrays

00:47 - 01:13

Recall that scikit-learn requires features and target values in distinct variables, X and y. To use all of the features in our dataset, we drop our target, blood glucose levels, and store the values attribute as X. For y, we take the the target column's values attribute. We can print the type for X and y to confirm they are now both NumPy arrays.

## 4. Making predictions from a single feature

01:13 - 01:57

To start, let's try to predict blood glucose levels from a single feature: body mass index. To do this, we slice out the BMI column of X, which is the fourth column, storing as the variable X\_bmi. Checking the shape of y and X\_bmi, we see that they are both one-dimensional arrays. This is fine for y, but our features must be formatted as a two-dimensional array to be accepted by scikit-learn. To convert the shape of X\_bmi we apply NumPy's dot-reshape method, passing minus one followed by one. Printing the shape again shows X\_bmi is now the correct shape for our model.

## 5. Plotting glucose vs. body mass index

01:57 - 02:18

Now, let's plot blood glucose levels as a function of body mass index. We import matplotlib-dot-pyplot as plt, then pass X\_bmi and y to plt-dot-scatter. We'll also label our axes using the xlabel and ylabel methods.

## 6. Plotting glucose vs. body mass index

02:18 - 02:26

We can see that, generally, as body mass index increases, blood glucose levels also tend to increase.

## 7. Fitting a regression model

02:26 - 03:40

It's time to fit a regression model to our data. We're going to use a model called linear regression, which fits a straight line to our data. We will explain the mechanics of linear regression in the next video, but first, let's see how to fit it and plot predictions. We import LinearRegression from sklearn-dot-linear\_model, and instantiate our regression model. As we are modeling the relationship between the feature, body mass index, and the target, blood glucose levels, rather than predicting target values for new observations, we fit the model to all of our feature observations. We do this by calling reg-dot-fit and passing in the feature data and the target variable, the same as we did for classification problems. After this, we can create the predictions variable by calling reg-dot-predict and passing in our features. As we are predicting the target values of the features used to train the model, this gives us a line of best fit for our data. We produce our scatter plot again, and then call plt-dot-plot to produce a line plot, passing our features, X\_bmi, followed by our predictions.

## 8. Fitting a regression model

03:40 - 03:50

The black line represents the linear regression model's fit of blood glucose values against body mass index, which appears to have a weak-to-moderate positive correlation.

## 9. Let's practice!

03:50 - 03:55

Now let's build a regression model of our own!

# Creating features

In this chapter, you will work with a dataset called sales\_df, which contains information on advertising campaign expenditure across different media types, and the number of dollars generated in sales for the respective campaign. The dataset has been preloaded for you. Here are the first two rows:

tv radio social\_media sales

1 13000.0 9237.76 2409.57 46677.90

2 41000.0 15886.45 2913.41 150177.83

You will use the advertising expenditure as features to predict sales values, initially working with the "radio" column. However, before you make any predictions you will need to create the feature and target arrays, reshaping them to the correct format for scikit-learn.

##### Instructions

**100 XP**

* Create X, an array of the values from the sales\_df DataFrame's "radio" column.
* Create y, an array of the values from the sales\_df DataFrame's "sales" column.
* Reshape X into a two-dimensional NumPy array.
* Print the shape of X and y.
* import numpy as np
* # Create X from the radio column's values
* X = \_\_\_\_
* # Create y from the sales column's values
* y = \_\_\_\_
* # Reshape X
* X = \_\_\_\_
* # Check the shape of the features and targets
* print(\_\_\_\_)

import numpy as np

# Create X from the radio column's values

X = sales\_df['radio'].values

# Create y from the sales column's values

y = sales\_df['sales'].values

# Reshape X

X = X.reshape(-1, 1)

# Check the shape of the features and targets

print(X.shape, y.shape)

import numpy as np

# Create X from the radio column's values

X = sales\_df['radio'].values

# Create y from the sales column's values

y = sales\_df['sales'].values

# Reshape X

X = X.reshape(-1, 1)

# Check the shape of the features and targets

print(X.shape, y.shape)

(4546, 1) (4546,)

<script.py> output:

(4546, 1) (4546,)

Excellent! See that there are 4546 values in both arrays? Now let's build a linear regression model!

# Building a linear regression model

Now you have created your feature and target arrays, you will train a linear regression model on all feature and target values.

As the goal is to assess the relationship between the feature and target values there is no need to split the data into training and test sets.

X and y have been preloaded for you as follows:

y = sales\_df["sales"].values

X = sales\_df["radio"].values.reshape(-1, 1)

##### Instructions

**100 XP**

* Import LinearRegression.
* Instantiate a linear regression model.
* Predict sales values using X, storing as predictions.
* # Import LinearRegression
* from \_\_\_\_.\_\_\_\_ import \_\_\_\_
* # Create the model
* reg = \_\_\_\_()
* # Fit the model to the data
* \_\_\_\_
* # Make predictions
* predictions = \_\_\_\_
* print(predictions[:5])
* # Import LinearRegression
* from sklearn.linear\_model import LinearRegression
* # Create the model
* reg = LinearRegression()
* # Fit the model to the data
* reg.fit(X,y)
* # Make predictions
* predictions = reg.predict(X)
* print(predictions[:5])

# Import LinearRegression

from sklearn.linear\_model import LinearRegression

# Create the model

reg = LinearRegression()

# Fit the model to the data

reg.fit(X,y)

# Make predictions

predictions = reg.predict(X)

print(predictions[:5])

[ 95491.17119147 117829.51038393 173423.38071499 291603.11444202

111137.28167129]

<script.py> output:

[ 95491.17119147 117829.51038393 173423.38071499 291603.11444202

111137.28167129]

Great model building! See how sales values for the first five predictions range from $95,000 to over $290,000. Let's visualize the model's fit.

# Visualizing a linear regression model

Now you have built your linear regression model and trained it using all available observations, you can visualize how well the model fits the data. This allows you to interpret the relationship between radio advertising expenditure and sales values.

The variables X, an array of radio values, y, an array of sales values, and predictions, an array of the model's predicted values for y given X, have all been preloaded for you from the previous exercise.

##### Instructions

**100 XP**

* Import matplotlib.pyplot as plt.
* Create a scatter plot visualizing y against X, with observations in blue.
* Draw a red line plot displaying the predictions against X.
* Display the plot.
* # Import matplotlib.pyplot
* import \_\_\_\_.\_\_\_\_ as \_\_\_\_
* # Create scatter plot
* plt.scatter(\_\_\_\_, \_\_\_\_, color="\_\_\_\_")
* # Create line plot
* plt.plot(\_\_\_\_, \_\_\_\_, color="\_\_\_\_")
* plt.xlabel("Radio Expenditure ($)")
* plt.ylabel("Sales ($)")
* # Display the plot
* plt.\_\_\_\_()

The model nicely captures a near-perfect linear correlation between radio advertising expenditure and sales! Now let's take a look at what is going on under the hood to calculate this relationship.

# Import matplotlib.pyplot

import matplotlib.pyplot as plt

# Create scatter plot

plt.scatter(X, y, color="blue")

# Create line plot

plt.plot(X, predictions, color="red")

plt.xlabel("Radio Expenditure ($)")

plt.ylabel("Sales ($)")

# Display the plot

plt.show()

# Import matplotlib.pyplot import matplotlib.pyplot as plt # Create scatter plot plt.scatter(X, y, color="blue") # Create line plot plt.plot(X, predictions, color="red") plt.xlabel("Radio Expenditure ($)") plt.ylabel("Sales ($)") # Display the plot plt.show()

**anscript**

## 1. The basics of linear regression

00:00 - 00:03

So, how does linear regression work?

## 2. Regression mechanics

00:03 - 00:47

We want to fit a line to the data, and in two dimensions this takes the form of y equals ax plus b. Using a single feature is known as simple linear regression, where y is the target, x is the feature, and a and b are the model parameters that we want to learn. a and b are also called the model coefficients, or the slope and intercept, respectively. So how do we accurately choose values for a and b? We can define an error function for any given line and then choose the line that minimizes this function. Error functions are also called loss or cost functions.

## 3. The loss function

00:47 - 00:54

Let's visualize a loss function using this scatter plot. We want the line to be as close to the

## 4. The loss function

00:54 - 01:05

observations as possible. Therefore, we want to minimize the vertical distance between the fit and the data. So for each observation,

## 5. The loss function

01:05 - 01:09

we calculate the vertical distance between it and the line.

## 6. The loss function

01:09 - 01:15

This distance is called a residual. We could try to minimize the sum of the residuals,

## 7. The loss function

01:15 - 01:18

but then each positive residual would cancel out

## 8. Ordinary Least Squares

01:18 - 01:40

each negative residual. To avoid this, we square the residuals. By adding all the squared residuals, we calculate the residual sum of squares, or RSS. This type of linear regression is called Ordinary Least Squares, or OLS, where we aim to minimize the RSS.

## 9. Linear regression in higher dimensions

01:40 - 02:24

When we have two features, x1 and x2, and one target, y, a line takes the form y = a1x1 + a2x2 + b. So to fit a linear regression model we specify three variables, a1, a2, and the intercept, b. When adding more features, it is known as multiple linear regression. Fitting a multiple linear regression model means specifying a coefficient, a n, for n number of features, and b. For multiple linear regression models, scikit-learn expects one variable each for feature and target values.

## 10. Linear regression using all features

02:24 - 02:51

Let's perform linear regression to predict blood glucose levels using all of the features from the diabetes dataset. We import LinearRegression from sklearn-dot-linear\_model. Then we split the data into training and test sets, instantiate the model, fit it on the training set, and predict on the test set. Note that linear regression in scikit-learn performs OLS under the hood.

## 11. R-squared

02:51 - 03:14

The default metric for linear regression is R-squared, which quantifies the amount of variance in the target variable that is explained by the features. Values can range from zero to one, with one meaning the features completely explain the target's variance. Here are two plots visualizing high and low R-squared respectively.

## 12. R-squared in scikit-learn

03:14 - 03:28

To compute R-squared, we call the model's dot-score method, passing the test features and targets. Here the features only explain about 35 percent of blood glucose level variance.

## 13. Mean squared error and root mean squared error

03:28 - 03:59

Another way to assess a regression model's performance is to take the mean of the residual sum of squares. This is known as the mean squared error, or MSE. MSE is measured in units of our target variable, squared. For example, if a model is predicting a dollar value, MSE will be in dollars squared. To convert to dollars, we can take the square root, known as the root mean squared error, or RMSE.

## 14. RMSE in scikit-learn

03:59 - 04:24

To calculate RMSE, we import mean\_squared\_error from sklearn-dot-metrics, then call mean\_squared\_error. We pass y\_test and y\_pred, and set squared equal to False, which returns the square root of the MSE. The model has an average error for blood glucose levels of around 24 milligrams per deciliter.

## 15. Let's practice!

04:24 - 04:29

Now let's build and evaluate a multiple linear regression model!

# Fit and predict for regression

Now you have seen how linear regression works, your task is to create a multiple linear regression model using all of the features in the sales\_df dataset, which has been preloaded for you. As a reminder, here are the first two rows:

tv radio social\_media sales

1 13000.0 9237.76 2409.57 46677.90

2 41000.0 15886.45 2913.41 150177.83

You will then use this model to predict sales based on the values of the test features.

LinearRegression and train\_test\_split have been preloaded for you from their respective modules.

##### Instructions

**100 XP**

* Create X, an array containing values of all features in sales\_df, and y, containing all values from the "sales" column.
* Instantiate a linear regression model.
* Fit the model to the training data.
* Create y\_pred, making predictions for sales using the test features.
* # Create X and y arrays
* X = sales\_df.\_\_\_\_("\_\_\_\_", axis=\_\_\_\_).\_\_\_\_
* y = sales\_df["\_\_\_\_"].\_\_\_\_
* X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)
* # Instantiate the model
* reg = \_\_\_\_
* # Fit the model to the data
* \_\_\_\_
* # Make predictions
* y\_pred = reg.\_\_\_\_(\_\_\_\_)
* print("Predictions: {}, Actual Values: {}".format(y\_pred[:2], y\_test[:2]))
* # Create X and y arrays
* X = sales\_df.drop("sales", axis=1).values
* y = sales\_df["sales"].values
* X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)
* # Instantiate the model
* reg = LinearRegression()
* # Fit the model to the data
* reg.fit(X\_train,y\_train)
* # Make predictions
* y\_pred = reg.predict(X\_test)
* print("Predictions: {}, Actual Values: {}".format(y\_pred[:2], y\_test[:2]))

# Create X and y arrays

X = sales\_df.drop("sales", axis=1).values

y = sales\_df["sales"].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Instantiate the model

reg = LinearRegression()

# Fit the model to the data

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

# Make predictions

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

print("Predictions: {}, Actual Values: {}".format(y\_pred[:2], y\_test[:2]))

Predictions: [53176.66154234 70996.19873235], Actual Values: [55261.28 67574.9 ]

<script.py> output:

Predictions: [53176.66154234 70996.19873235], Actual Values: [55261.28 67574.9 ]

Great work! The first two predictions appear to be within around 5% of the actual values from the test set!

# Regression performance

Now you have fit a model, reg, using all features from sales\_df, and made predictions of sales values, you can evaluate performance using some common regression metrics.

The variables X\_train, X\_test, y\_train, y\_test, and y\_pred, along with the fitted model, reg, all from the last exercise, have been preloaded for you.

Your task is to find out how well the features can explain the variance in the target values, along with assessing the model's ability to make predictions on unseen data.

##### Instructions

**100 XP**

* Import mean\_squared\_error.
* Calculate the model's R-squared score by passing the test feature values and the test target values to an appropriate method.
* Calculate the model's root mean squared error using y\_test and y\_pred.
* Print r\_squared and rmse.
* # Import mean\_squared\_error
* from \_\_\_\_.\_\_\_\_ import \_\_\_\_
* # Compute R-squared
* r\_squared = reg.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* # Compute RMSE
* rmse = \_\_\_\_(\_\_\_\_, \_\_\_\_, squared=\_\_\_\_)
* # Print the metrics
* print("R^2: {}".format(\_\_\_\_))
* print("RMSE: {}".format(\_\_\_\_))

# Import mean\_squared\_error

from sklearn.metrics import mean\_squared\_error

# Compute R-squared

r\_squared = reg.score(X\_test, y\_test)

# Compute RMSE

rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)

# Print the metrics

print("R^2: {}".format(r\_squared))

print("RMSE: {}".format(rmse))

# Import mean\_squared\_error

from sklearn.metrics import mean\_squared\_error

# Compute R-squared

r\_squared = reg.score(X\_test, y\_test)

# Compute RMSE

rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)

# Print the metrics

print("R^2: {}".format(r\_squared))

print("RMSE: {}".format(rmse))

R^2: 0.9990165886162027

RMSE: 2942.372219812037

<script.py> output:

R^2: 0.9990165886162027

RMSE: 2942.372219812037

Wow, the features explain 99.9% of the variance in sales values! Looks like this company's advertising strategy is working well!

## 1. Cross-validation

00:00 - 00:14

Great work on those regression challenges! Hopefully we are now feeling familiar with train test split and computing model performance metrics on our test set. But, there is a potential pitfall of this process.

## 2. Cross-validation motivation

00:14 - 00:41

If we're computing R-squared on our test set, the R-squared returned is dependent on the way that we split up the data! The data points in the test set may have some peculiarities that mean the R-squared computed on it is not representative of the model's ability to generalize to unseen data. To combat this dependence on what is essentially a random split, we use a technique called cross-validation.

## 3. Cross-validation basics

00:41 - 00:46

We begin by splitting the dataset into five groups or folds.

## 4. Cross-validation basics

00:46 - 00:50

Then we set aside the first fold as a test set,

## 5. Cross-validation basics

00:50 - 00:54

fit our model on the remaining four folds, predict on our test set,

## 6. Cross-validation basics

00:54 - 00:58

and compute the metric of interest, such as R-squared.

## 7. Cross-validation basics

00:58 - 01:02

Next, we set aside the second fold as our test set,

## 8. Cross-validation basics

01:02 - 01:06

fit on the remaining data, predict on the test set,

## 9. Cross-validation basics

01:06 - 01:08

and compute the metric of interest.

## 10. Cross-validation basics

01:08 - 01:11

Then similarly with the third fold,

## 11. Cross-validation basics

01:11 - 01:12

the fourth fold,

## 12. Cross-validation basics

01:12 - 01:25

and the fifth fold. As a result we get five values of R-squared from which we can compute statistics of interest, such as the mean, median, and 95% confidence intervals.

## 13. Cross-validation and model performance

01:25 - 01:55

As we split the dataset into five folds, we call this process 5-fold cross-validation. If we use 10 folds, it is called 10-fold cross-validation. More generally, if we use k folds, it is called k-fold cross-validation or k-fold CV. There is, however, a trade-off. Using more folds is more computationally expensive. This is because we are fitting and predicting more times.

## 14. Cross-validation in scikit-learn

01:55 - 03:23

To perform k-fold cross-validation in scikit-learn, we import cross\_val\_score from sklearn-dot-model\_selection. We also import KFold, which allows us to set a seed and shuffle our data, making our results repeatable downstream. We first call KFold. The n\_splits argument has a default of five, but in this case we assign six, allowing us to use six folds from our dataset for cross-validation. We also set shuffle to True, which shuffles our dataset before splitting into folds. We also assign a seed to the random\_state keyword argument, ensuring our data would be split in the same way if we repeat the process making the results repeatable downstream. We save this as the variable kf. As usual, we instantiate our model, in this case, linear regression. We then call cross\_val\_score, passing the model, the feature data, and the target data as the first three positional arguments. We also specify the number of folds by setting the keyword argument cv equal to our kf variable. This returns an array of cross-validation scores, which we assign to cv\_results. The length of the array is the number of folds utilized. Note that the score reported is R squared, as this is the default score for linear regression.

## 15. Evaluating cross-validation peformance

03:23 - 03:53

We can now print the scores. This returns six results ranging from zero-point-seven to approximately zero-point-seven-seven. We can calculate the mean score using np-dot-mean, and the standard deviation using np-dot-std. Additionally, we can calculate the 95% confidence interval using the np-dot-quantile function, passing our results followed by a list containing the upper and lower limits of our interval as decimals.

## 16. Let's practice!

03:53 - 03:59

Now let's apply k-fold cross-validation on our sales dataset!

# Cross-validation for R-squared

Cross-validation is a vital approach to evaluating a model. It maximizes the amount of data that is available to the model, as the model is not only trained but also tested on all of the available data.

In this exercise, you will build a linear regression model, then use 6-fold cross-validation to assess its accuracy for predicting sales using social media advertising expenditure. You will display the individual score for each of the six-folds.

The sales\_df dataset has been split into y for the target variable, and X for the features, and preloaded for you. LinearRegression has been imported from sklearn.linear\_model.

##### Instructions

**100 XP**

* Import KFold and cross\_val\_score.
* Create kf by calling KFold(), setting the number of splits to six, shuffle to True, and setting a seed of 5.
* Perform cross-validation using reg on X and y, passing kf to cv.
* Print the cv\_scores.
* # Import the necessary modules
* from \_\_\_\_.\_\_\_\_ import \_\_\_\_, \_\_\_\_
* # Create a KFold object
* kf = \_\_\_\_(n\_splits=\_\_\_\_, shuffle=\_\_\_\_, random\_state=\_\_\_\_)
* reg = LinearRegression()
* # Compute 6-fold cross-validation scores
* cv\_scores = \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_, cv=\_\_\_\_)
* # Print scores
* print(\_\_\_\_)

# Import the necessary modules

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

# Create a KFold object

kf = KFold(n\_splits=6, shuffle=True, random\_state=5)

reg = LinearRegression()

# Compute 6-fold cross-validation scores

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

# Print scores

print(cv\_scores)

# Import the necessary modules

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

# Create a KFold object

kf = KFold(n\_splits=6, shuffle=True, random\_state=5)

reg = LinearRegression()

# Compute 6-fold cross-validation scores

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

# Print scores

print(cv\_scores)

[0.74451678 0.77241887 0.76842114 0.7410406 0.75170022 0.74406484]

<script.py> output:

[0.74451678 0.77241887 0.76842114 0.7410406 0.75170022 0.74406484]

Notice how R-squared for each fold ranged between 0.74 and 0.77? By using cross-validation, we can see how performance varies depending on how the data is split!

# Analyzing cross-validation metrics

Now you have performed cross-validation, it's time to analyze the results.

You will display the mean, standard deviation, and 95% confidence interval for cv\_results, which has been preloaded for you from the previous exercise.

numpy has been imported for you as np.

##### Instructions

**100 XP**

* Calculate and print the mean of the results.
* Calculate and print the standard deviation of cv\_results.
* Display the 95% confidence interval for your results using np.quantile().
* # Print the mean
* print(\_\_\_\_(\_\_\_\_))
* # Print the standard deviation
* print(\_\_\_\_(\_\_\_\_))
* # Print the 95% confidence interval
* print(\_\_\_\_(\_\_\_\_, [\_\_\_\_, \_\_\_\_]))

# Print the mean

print(np.mean(cv\_results))

# Print the standard deviation

print(np.std(cv\_results))

# Print the 95% confidence interval

print(np.quantile(cv\_results, [0.025, 0.975]))

# Print the mean

print(np.mean(cv\_results))

# Print the standard deviation

print(np.std(cv\_results))

# Print the 95% confidence interval

print(np.quantile(cv\_results, [0.025, 0.975]))

0.7536937416666666

0.012305386274436092

[0.74141863 0.77191915]

<script.py> output:

0.7536937416666666

0.012305386274436092

[0.74141863 0.77191915]

An average score of 0.75 with a low standard deviation is pretty good for a model out of the box! Now let's learn how to apply regularization to our regression models

## 1. Regularized regression

00:00 - 00:07

Now let's explore regularization in regression, a technique used to avoid overfitting.

## 2. Why regularize?

00:07 - 00:32

Recall that fitting a linear regression model minimizes a loss function to choose a coefficient, a, for each feature, and the intercept, b. If we allow these coefficients to be very large, we can get overfitting. Therefore, it is common practice to alter the loss function so that it penalizes large coefficients. This is called regularization.

## 3. Ridge regression

00:32 - 01:40

The first type of regularized regression that we'll look at is called ridge. With ridge, we use the Ordinary Least Squares loss function plus the squared value of each coefficient, multiplied by a constant, alpha. So, when minimizing the loss function, models are penalized for coefficients with large positive or negative values. When using ridge, we need to choose the alpha value in order to fit and predict. Essentially, we can select the alpha for which our model performs best. Picking alpha for ridge is similar to picking k in KNN. Alpha in ridge is known as a hyperparameter, which is a variable used for selecting a model's parameters. Alpha controls model complexity. When alpha equals zero, we are performing OLS, where large coefficients are not penalized and overfitting may occur. A high alpha means that large coefficients are significantly penalized, which can lead to underfitting.

## 4. Ridge regression in scikit-learn

01:40 - 02:27

To perform ridge regression in scikit-learn, we import Ridge from sklearn-dot-linear\_model. To highlight the impact of different alpha values, we create an empty list for our scores, then loop through a list of different alpha values. Inside the for loop we instantiate Ridge, setting the alpha keyword argument equal to the iterator, also called alpha. We fit on the training data, and predict on the test data. We save the model's R-squared value to the scores list. Finally, outside of the loop, we print the scores for the models with five different alpha values. We see performance gets worse as alpha increases.

## 5. Lasso regression

02:27 - 02:42

There is another type of regularized regression called lasso, where our loss function is the OLS loss function plus the absolute value of each coefficient multiplied by some constant, alpha.

## 6. Lasso regression in scikit-learn

02:42 - 02:57

To use Lasso we import it from sklearn-dot-linear\_model. The actual method for performing lasso regression in scikit-learn mirrors ridge regression, as we can see here. Performance drops substantially as alpha goes over 20!

## 7. Lasso regression for feature selection

02:57 - 03:17

Lasso regression can actually be used to assess feature importance. This is because it tends to shrink the coefficients of less important features to zero. The features whose coefficients are not shrunk to zero are selected by the lasso algorithm. Let's check this out in practice.

## 8. Lasso for feature selection in scikit-learn

03:17 - 03:56

We import Lasso. Next, we create our feature and target arrays, and use the dataset's dot-columns attribute to access the feature names and store as the variable names. As we are calculating feature importance we use the entire dataset, rather than splitting it. We then instantiate Lasso, setting alpha to zero-point-one. We fit the model to the data and extract the coefficients using the dot-coef-underscore attribute, storing as lasso\_coef. We then plot the coefficients for each feature.

## 9. Lasso for feature selection in scikit-learn

03:56 - 04:24

We can see that the most important predictor for our target variable, blood glucose levels, is the binary value for whether an individual has diabetes or not! This is not surprising, but is a great sanity check. This type of feature selection is very important because it allows us to communicate results to non-technical audiences. It is also useful for identifying which factors are important predictors for various physical phenomena.

## 10. Let's practice!

04:24 - 04:29

Now let's apply regularization to our regression models!

# Regularized regression: Ridge

Ridge regression performs regularization by computing the squared values of the model parameters multiplied by alpha and adding them to the loss function.

In this exercise, you will fit ridge regression models over a range of different alpha values, and print their

scores. You will use all of the features in the sales\_df dataset to predict "sales". The data has been split into X\_train, X\_test, y\_train, y\_test for you.

A variable called alphas has been provided as a list containing different alpha values, which you will loop through to generate scores.

##### Instructions

**100 XP**

* Import Ridge.
* Instantiate Ridge, setting alpha equal to alpha.
* Fit the model to the training data.
* Calculate the
* score for each iteration of ridge.
* # Import Ridge
* from \_\_\_\_.\_\_\_\_ import \_\_\_\_
* alphas = [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]
* ridge\_scores = []
* for alpha in alphas:
* # Create a Ridge regression model
* ridge = \_\_\_\_
* # Fit the data
* \_\_\_\_
* # Obtain R-squared
* score = \_\_\_\_
* ridge\_scores.append(score)
* print(ridge\_scores)
* # Import Ridge
* from sklearn.linear\_model import Ridge
* alphas = [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]
* ridge\_scores = []
* for alpha in alphas:
* # Create a Ridge regression model
* ridge = Ridge(alpha=alpha)
* # Fit the data
* ridge.fit(X\_train, y\_train)
* # Obtain R-squared
* score = ridge.score(X\_test, y\_test)
* ridge\_scores.append(score)
* print(ridge\_scores)
* # Import Ridge
* from sklearn.linear\_model import Ridge
* alphas = [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]
* ridge\_scores = []
* for alpha in alphas:
* # Create a Ridge regression model
* ridge = Ridge(alpha=alpha)
* # Fit the data
* ridge.fit(X\_train, y\_train)
* # Obtain R-squared
* score = ridge.score(X\_test, y\_test)
* ridge\_scores.append(score)
* print(ridge\_scores)
* [0.9990152104759369, 0.9990152104759373, 0.9990152104759419, 0.9990152104759871, 0.9990152104764387, 0.9990152104809561]
* <script.py> output:
* [0.9990152104759369, 0.9990152104759373, 0.9990152104759419, 0.9990152104759871, 0.9990152104764387, 0.9990152104809561]

Well done! The scores don't appear to change much as alpha increases, which is indicative of how well the features explain the variance in the target—even by heavily penalizing large coefficients, underfitting does not occur!

# Lasso regression for feature importance

In the video, you saw how lasso regression can be used to identify important features in a dataset.

In this exercise, you will fit a lasso regression model to the sales\_df data and plot the model's coefficients.

The feature and target variable arrays have been pre-loaded as X and y, along with sales\_columns, which contains the dataset's feature names.

##### Instructions

**100 XP**

* Import Lasso from sklearn.linear\_model.
* Instantiate a Lasso regressor with an alpha of 0.3.
* Fit the model to the data.
* Compute the model's coefficients, storing as lasso\_coef.
* # Import Lasso
* from \_\_\_\_.\_\_\_\_ import \_\_\_\_
* # Instantiate a lasso regression model
* lasso = \_\_\_\_
* # Fit the model to the data
* \_\_\_\_
* # Compute and print the coefficients
* lasso\_coef = \_\_\_\_
* print(lasso\_coef)
* plt.bar(sales\_columns, lasso\_coef)
* plt.xticks(rotation=45)
* plt.show()
* # Import Lasso
* from sklearn.linear\_model import Lasso
* # Instantiate a lasso regression model
* lasso = Lasso(alpha=0.3)
* # Fit the model to the data
* lasso.fit(X, y)
* # Compute and print the coefficients
* lasso\_coef = lasso.fit(X, y).coef\_
* print(lasso\_coef)
* plt.bar(sales\_columns, lasso\_coef)
* plt.xticks(rotation=45)
* plt.show()
* # Import Lasso
* from sklearn.linear\_model import Lasso
* # Instantiate a lasso regression model
* lasso = Lasso(alpha=0.3)
* # Fit the model to the data
* lasso.fit(X, y)
* # Compute and print the coefficients
* lasso\_coef = lasso.fit(X, y).coef\_
* print(lasso\_coef)
* plt.bar(sales\_columns, lasso\_coef)
* plt.xticks(rotation=45)
* plt.show()
* [ 3.56256962 -0.00397035 0.00496385]
* <script.py> output:
* [ 3.56256962 -0.00397035 0.00496385]

See how the figure makes it clear that expenditure on TV advertising is the most important feature in the dataset to predict sales values! In the next chapter, we will learn how to further assess and improve our model's performance!

## 1. How good is your model?

00:00 - 00:02

Thinking back to classification problems,

## 2. Classification metrics

00:02 - 00:13

recall that we can use accuracy, the fraction of correctly classified labels, to measure model performance. However, accuracy is not always a useful metric.

## 3. Class imbalance

00:13 - 00:54

# Deciding on a primary metric

As you have seen, several metrics can be useful to evaluate the performance of classification models, including accuracy, precision, recall, and F1-score.

In this exercise, you will be provided with three different classification problems, and your task is to select the problem where **precision** is best suited as the primary metric.

##### Answer the question

**50XP**

#### Possible Answers

* 

A model predicting the presence of cancer as the positive class.

press1

* 

A classifier predicting the positive class of a computer program containing malware.

press2

* 

A model predicting if a customer is a high-value lead for a sales team with limited capacity.

press3

* A model predicting the presence of cancer as the positive class. This model should minimize the number of false negatives, so **recall** is a more appropriate metric.
* A classifier predicting the positive class of a computer program containing malware. To avoid installing malware, false negatives should be minimized, hence **recall** or **F1-score** are better metrics for this model.
* **A model predicting if a customer is a high-value lead for a sales team with limited capacity. Correct! With limited capacity, the sales team needs the model to return the highest proportion of true positives compared to all predicted positives, thus minimizing wasted effort.**

# Assessing a diabetes prediction classifier

In this chapter you'll work with the diabetes\_df dataset introduced previously.

The goal is to predict whether or not each individual is likely to have diabetes based on the features body mass index (BMI) and age (in years). Therefore, it is a binary classification problem. A target value of 0 indicates that the individual does not have diabetes, while a value of 1 indicates that the individual does have diabetes.

diabetes\_df has been preloaded for you as a pandas DataFrame and split into X\_train, X\_test, y\_train, and y\_test. In addition, a KNeighborsClassifier() has been instantiated and assigned to knn.

You will fit the model, make predictions on the test set, then produce a confusion matrix and classification report.

##### Instructions

**100 XP**

* Import confusion\_matrix and classification\_report.
* Fit the model to the training data.
* Predict the labels of the test set, storing the results as y\_pred.
* Compute and print the confusion matrix and classification report for the test labels versus the predicted labels.

Consider a model for predicting whether a bank transaction is fraudulent, where only 1% of transactions are actually fraudulent. We could build a model that classifies every transaction as legitimate; this model would have an accuracy of 99%! However, it does a terrible job of actually predicting fraud, so it fails at its original purpose. The situation where one class is more frequent is called class imbalance. Here, the class of legitimate transactions contains way more instances than the class of fraudulent transactions. This is a common situation in practice and requires a different approach to assessing the model's performance.

## 4. Confusion matrix for assessing classification performance

00:54 - 01:05

Given a binary classifier, such as our fraudulent transactions example, we can create a 2-by-2 matrix that summarizes performance called a confusion matrix.

## 5. Assessing classification performance

01:05 - 01:09

Across the top are the predicted labels,

## 6. Assessing classification performance

01:09 - 01:11

and down the side are the actual labels.

## 7. Assessing classification performance

01:11 - 01:17

Given any model, we can fill in the confusion matrix according to its predictions.

## 8. Assessing classification performance

01:17 - 01:22

The true positives are the number of fraudulent transactions correctly labeled;

## 9. Assessing classification performance

01:22 - 01:27

The true negatives are the number of legitimate transactions correctly labeled;

## 10. Assessing classification performance

01:27 - 01:32

The false negatives are the number of legitimate transactions incorrectly labeled;

## 11. Assessing classification performance

01:32 - 01:38

And the false positives are the number of transactions incorrectly labeled as fraudulent.

## 12. Assessing classification performance

01:38 - 01:59

Usually, the class of interest is called the positive class. As we aim to detect fraud, the positive class is an illegitimate transaction. So why is the confusion matrix important? Firstly, we can retrieve accuracy: it's the sum of true predictions divided by the total sum of the matrix.

## 13. Precision

01:59 - 02:33

Secondly, there are other important metrics we can calculate from the confusion matrix. Precision is the number of true positives divided by the sum of all positive predictions. It is also called the positive predictive value. In our case, this is the number of correctly labeled fraudulent transactions divided by the total number of transactions classified as fraudulent. High precision means having a lower false positive rate. For our classifier, this translates to fewer legitimate transactions being classified as fraudulent.

## 14. Recall

02:33 - 02:53

Recall is the number of true positives divided by the sum of true positives and false negatives. This is also called sensitivity. High recall reflects a lower false negative rate. For our classifier, it means predicting most fraudulent transactions correctly.

## 15. F1 score

02:53 - 03:18

The F1-score is the harmonic mean of precision and recall. This metric gives equal weight to precision and recall, therefore it factors in both the number of errors made by the model and the type of errors. The F1 score favors models with similar precision and recall, and is a useful metric if we are seeking a model which performs reasonably well across both metrics.

## 16. Confusion matrix in scikit-learn

03:18 - 03:38

Using our churn dataset, to compute the confusion matrix, along with the metrics, we import classification\_report and confusion\_matrix from sklearn-dot-metrics. We instantiate our classifier, split the data, fit the training data, and predict the labels of the test set.

## 17. Confusion matrix in scikit-learn

03:38 - 03:49

We pass the test set labels and the predicted labels to the confusion matrix function. We can see 1106 true negatives in the top left.

## 18. Classification report in scikit-learn

03:49 - 04:12

Passing the same arguments to classification report outputs all the relevant metrics. It includes precision and recall by class, point-seven-six and point-one-six for the churn class respectively, which highlights how poorly the model's recall is on the churn class. Support represents the number of instances for each class within the true labels.

## 19. Let's practice!

04:12 - 04:19

Now let's evaluate a classification model using our diabetes dataset!

# Assessing a diabetes prediction classifier

In this chapter you'll work with the diabetes\_df dataset introduced previously.

The goal is to predict whether or not each individual is likely to have diabetes based on the features body mass index (BMI) and age (in years). Therefore, it is a binary classification problem. A target value of 0 indicates that the individual does not have diabetes, while a value of 1 indicates that the individual does have diabetes.

diabetes\_df has been preloaded for you as a pandas DataFrame and split into X\_train, X\_test, y\_train, and y\_test. In addition, a KNeighborsClassifier() has been instantiated and assigned to knn.

You will fit the model, make predictions on the test set, then produce a confusion matrix and classification report.

##### Instructions

**100 XP**

* Import confusion\_matrix and classification\_report.
* Fit the model to the training data.
* Predict the labels of the test set, storing the results as y\_pred.
* Compute and print the confusion matrix and classification report for the test labels versus the predicted labels.

Excellent! The model produced 34 true positives and 35 false positives, meaning precision was less than 50%, which is confirmed in the classification report. The output also shows a better F1-score for the zero class, which represents individuals who do not have diabetes

# Import confusion matrix

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

knn = KNeighborsClassifier(n\_neighbors=6)

# Fit the model 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(confusion\_matrix(y\_test, y\_pred))

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

# Import confusion matrix

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

knn = KNeighborsClassifier(n\_neighbors=6)

# Fit the model 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(confusion\_matrix(y\_test, y\_pred))

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

[[116 35]

[ 46 34]]

precision recall f1-score support

0 0.72 0.77 0.74 151

1 0.49 0.42 0.46 80

accuracy 0.65 231

macro avg 0.60 0.60 0.60 231

weighted avg 0.64 0.65 0.64 231

<script.py> output:

[[116 35]

[ 46 34]]

precision recall f1-score support

0 0.72 0.77 0.74 151

1 0.49 0.42 0.46 80

accuracy 0.65 231

macro avg 0.60 0.60 0.60 231

weighted avg 0.64 0.65 0.64 231

## 1. Logistic regression and the ROC curve

00:00 - 00:04

It's time to introduce another model: logistic regression.

## 2. Logistic regression for binary classification

00:04 - 00:37

Despite its name, logistic regression is used for classification. This model calculates the probability, p, that an observation belongs to a binary class. Using our diabetes dataset as an example, if p is more than or equal to zero-point-five, we label the data as one, representing a prediction that an individual is more likely to have diabetes; if p is less than zero-point-five, we label it zero to represent that they are more likely to not have diabetes.

## 3. Linear decision boundary

00:37 - 00:44

Note that logistic regression produces a linear decision boundary, as we can see in this image.

## 4. Logistic regression in scikit-learn

00:44 - 01:07

Using logistic regression in scikit-learn follows the same approach as used for other models. We first import LogisticRegression from sklearn-dot-linear\_model. Next we instantiate the classifier, split our data, fit the model on our training data, and predict on our test set. In this video we use the churn dataset.

## 5. Predicting probabilities

01:07 - 01:43

We can predict probabilities of each instance belonging to a class by calling logistic regression's predict\_proba method and passing the test features. This returns a 2-dimensional array with probabilities for both classes, in this case, that the individual did not churn, or did churn, respectively. We slice the second column, representing the positive class probabilities, and store the results as y\_pred\_probs. Here we see the model predicts a probability of point-zero-eight-nine that the first observation has churned.

## 6. Probability thresholds

01:43 - 01:56

The default probability threshold for logistic regression in scikit-learn is zero-point-five. This threshold can also apply to other models such as KNN. So what happens as we vary this threshold?

## 7. The ROC curve

01:56 - 02:13

We can use a receiver operating characteristic, or ROC curve, to visualize how different thresholds affect true positive and false positive rates. Here, the dotted line represents a chance model, which randomly guesses labels.

## 8. The ROC curve

02:13 - 02:25

When the threshold equals zero, the model predicts one for all observations, meaning it will correctly predict all positive values, and incorrectly predict all negative values.

## 9. The ROC curve

02:25 - 02:33

If the threshold equals one, the model predicts zero for all data, which means that both true and false positive rates

## 10. The ROC curve

02:33 - 02:35

are zero. If we

## 11. The ROC curve

02:35 - 02:41

vary the threshold, we get a series of different false positive and true positive rates.

## 12. The ROC curve

02:41 - 02:45

A line plot of the thresholds helps to visualize the trend.

## 13. Plotting the ROC curve

02:45 - 03:16

To plot the ROC curve, we import roc\_curve from sklearn-dot-metrics. We then call the function roc\_curve; we pass the test labels as the first argument, and the predicted probabilities as the second. We unpack the results into three variables: false positive rate, FPR; true positive rate, TPR; and the thresholds. We can then plot a dotted line from zero to one, along with the FPR and TPR;

## 14. Plotting the ROC curve

03:16 - 03:23

to produce a figure such as this. This looks great, but how do we quantify the model's performance based on this plot?

## 15. ROC AUC

03:23 - 03:49

If we have a model with one for true positive rate and zero for false positive rate, this would be the perfect model. Therefore, we calculate the area under the ROC curve, a metric known as AUC. Scores range from zero to one, with one being ideal. Here, the model scores point-six-seven, which is only 34% better than a model making random guesses.

## 16. ROC AUC in scikit-learn

03:49 - 04:13

We can calculate AUC in scikit-learn by importing roc\_auc\_score from sklearn-dot-metrics. We call roc\_auc\_score, passing our test labels and our predicted probabilities, calculated by using the model's predict\_proba method on X\_test. As expected, we get a score of zero-point-six-seven.

## 17. Let's practice!

04:13 - 04:19

Now let's build a logistic regression model and evaluate its performance!

# Building a logistic regression model

In this exercise, you will build a logistic regression model using all features in the diabetes\_df dataset. The model will be used to predict the probability of individuals in the test set having a diabetes diagnosis.

The diabetes\_df dataset has been split into X\_train, X\_test, y\_train, and y\_test, and preloaded for you.

##### Instructions

**100 XP**

* Import LogisticRegression.
* Instantiate a logistic regression model, logreg.
* Fit the model to the training data.
* Predict the probabilities of each individual in the test set having a diabetes diagnosis, storing the array of positive probabilities as y\_pred\_probs.
* # Import LogisticRegression
* \_\_\_\_
* # Instantiate the model
* logreg = \_\_\_\_
* # Fit the model
* \_\_\_\_
* # Predict probabilities
* y\_pred\_probs = logreg.\_\_\_\_(\_\_\_\_)[\_\_\_\_, \_\_\_\_]
* print(y\_pred\_probs[:10])

# Import LogisticRegression

from sklearn.linear\_model import LogisticRegression

# Instantiate the model

logreg = LogisticRegression()

# Fit the model

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

# Predict probabilities

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

print(y\_pred\_probs[:10])

[0.26551031 0.18336542 0.12119596 0.15613565 0.49611285 0.44582236

0.01359235 0.61646125 0.55640546 0.7931187 ]

<script.py> output:

[0.26551031 0.18336542 0.12119596 0.15613565 0.49611285 0.44582236

0.01359235 0.61646125 0.55640546 0.7931187 ]

# Import LogisticRegression

from sklearn.linear\_model import LogisticRegression

# Instantiate the model

logreg = LogisticRegression()

# Fit the model

logreg.fit(X\_train, y\_train)

# Predict probabilities

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

print(y\_pred\_probs[:10])

Nicely done! Notice how the probability of a diabetes diagnosis for the first 10 individuals in the test set ranges from 0.01 to 0.79. Now let's plot the ROC curve to visualize performance using different thresholds.

# The ROC curve

Now you have built a logistic regression model for predicting diabetes status, you can plot the ROC curve to visualize how the true positive rate and false positive rate vary as the decision threshold changes.

The test labels, y\_test, and the predicted probabilities of the test features belonging to the positive class, y\_pred\_probs, have been preloaded for you, along with matplotlib.pyplot as plt.

You will create a ROC curve and then interpret the results.

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Import roc\_curve.
* Calculate the ROC curve values, using y\_test and y\_pred\_probs, and unpacking the results into fpr, tpr, and thresholds.
* Plot true positive rate against false positive rate.
* # Import roc\_curve
* \_\_\_\_
* # Generate ROC curve values: fpr, tpr, thresholds
* fpr, tpr, thresholds = \_\_\_\_(\_\_\_\_, \_\_\_\_)
* plt.plot([0, 1], [0, 1], 'k--')
* # Plot tpr against fpr
* plt.plot(\_\_\_\_, \_\_\_\_)
* plt.xlabel('False Positive Rate')
* plt.ylabel('True Positive Rate')
* plt.title('ROC Curve for Diabetes Prediction')
* plt.show()

# Import roc\_curve

from sklearn.metrics import roc\_curve

# Generate ROC curve values: fpr, tpr, thresholds

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

plt.plot([0, 1], [0, 1], 'k--')

# Plot tpr against fpr

plt.plot(fpr, tpr)

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve for Diabetes Prediction')

plt.show()

# Import roc\_curve from sklearn.metrics import roc\_curve # Generate ROC curve values: fpr, tpr, thresholds fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred\_probs) plt.plot([0, 1], [0, 1], 'k--') # Plot tpr against fpr plt.plot(fpr, tpr) plt.xlabel('False Positive Rate') plt.ylabel('True Positive Rate') plt.title('ROC Curve for Diabetes Prediction')

#### Question

Well done on producing the ROC curve for the diabetes prediction model.

But, what does the plot tell you about the model's performance?

##### Possible Answers

* 

The model is about as good as randomly guessing the class of each observation.

* 

The model is much worse than randomly guessing the class of each observation.

* 

The model is much better than randomly guessing the class of each observation.

* 

It is not possible to conclude whether the model performs better or worse than randomly guessing the class of each observation.

**Well done! The ROC curve is above the dotted line, so the model performs better than randomly guessing the class of each observation.**

# ROC AUC

The ROC curve you plotted in the last exercise looked promising.

Now you will compute the area under the ROC curve, along with the other classification metrics you have used previously.

The confusion\_matrix and classification\_report functions have been preloaded for you, along with the logreg model you previously built, plus X\_train, X\_test, y\_train, y\_test. Also, the model's predicted test set labels are stored as y\_pred, and probabilities of test set observations belonging to the positive class stored as y\_pred\_probs.

A knn model has also been created and the performance metrics printed in the console, so you can compare the roc\_auc\_score, confusion\_matrix, and classification\_report between the two models.

##### Instructions

**100 XP**

* Import roc\_auc\_score.
* Calculate and print the ROC AUC score, passing the test labels and the predicted positive class probabilities.
* Calculate and print the confusion matrix.
* Call classification\_report().
* # Import roc\_auc\_score
* \_\_\_\_
* # Calculate roc\_auc\_score
* print(\_\_\_\_(\_\_\_\_, \_\_\_\_))
* # Calculate the confusion matrix
* print(\_\_\_\_(\_\_\_\_, \_\_\_\_))
* # Calculate the classification report
* print(\_\_\_\_(\_\_\_\_, \_\_\_\_))

0.7300496688741722 [[124 27] [ 42 38]] precision recall f1-score support 0 0.75 0.82 0.78 151 1 0.58 0.47 0.52 80 accuracy 0.70 231 macro avg 0.67 0.65 0.65 231 weighted avg 0.69 0.70 0.69 231

# Import roc\_auc\_score

from sklearn.metrics import roc\_auc\_score

# Calculate roc\_auc\_score

print(roc\_auc\_score(y\_test, y\_pred\_probs))

# Calculate the confusion matrix

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

# Calculate the classification report

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

Did you notice that logistic regression performs better than the KNN model across all the metrics you calculated? A ROC AUC score of 0.8002 means this model is 60% better than a chance model at correctly predicting labels! scikit-learn makes it easy to produce several classification metrics with only a few lines of code.

# Import roc\_auc\_score

from sklearn.metrics import roc\_auc\_score

# Calculate roc\_auc\_score

print(roc\_auc\_score(y\_test, y\_pred\_probs))

# Calculate the confusion matrix

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

# Calculate the classification report

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

0.8002483443708608

[[121 30]

[ 30 50]]

precision recall f1-score support

0 0.80 0.80 0.80 151

1 0.62 0.62 0.62 80

accuracy 0.74 231

macro avg 0.71 0.71 0.71 231

weighted avg 0.74 0.74 0.74 231

<script.py> output:

0.8002483443708608

[[121 30]

[ 30 50]]

precision recall f1-score support

0 0.80 0.80 0.80 151

1 0.62 0.62 0.62 80

accuracy 0.74 231

macro avg 0.71 0.71 0.71 231

weighted avg 0.74 0.74 0.74 231

**Transcript**

## 1. Hyperparameter tuning

00:00 - 00:06

Now that we know how to evaluate model performance, let's explore how to optimize our model.

## 2. Hyperparameter tuning

00:06 - 00:29

Recall that we had to choose a value for alpha in ridge and lasso regression before fitting it. Likewise, before fitting and predicting KNN, we choose n\_neighbors. Parameters that we specify before fitting a model, like alpha and n\_neighbors, are called hyperparameters. So, a fundamental step for building a successful model:

## 3. Choosing the correct hyperparameters

00:29 - 01:01

is choosing the correct hyperparameters. We can try lots of different values, fit all of them separately, see how well they perform, and choose the best values! This is called hyperparameter tuning. When fitting different hyperparameter values, we use cross-validation to avoid overfitting the hyperparameters to the test set. We can still split the data, but perform cross-validation on the training set. We withhold the test set and use it for evaluating the tuned model.

## 4. Grid search cross-validation

01:01 - 01:31

One approach for hyperparameter tuning is called grid search, where we choose a grid of possible hyperparameter values to try. For example, we can search across two hyperparameters for a KNN model - the type of metric and a different number of neighbors. Here we have n neighbors between two and eleven in increments of three, and two metrics: euclidean and manhattan. Therefore, we can create a grid of values like this.

## 5. Grid search cross-validation

01:31 - 01:40

We perform k-fold cross-validation for each combination of hyperparameters. The mean scores for each combination are shown here.

## 6. Grid search cross-validation

01:40 - 01:44

We then choose hyperparameters that performed best, as shown here.

## 7. GridSearchCV in scikit-learn

01:44 - 02:40

Let's perform a grid search on a regression model using our sales dataset. We import GridSearchCV from sklearn-dot-model\_selection. We instantiate KFold. We then specify the names and values of the hyperparameters we wish to tune as the keys and values of a dictionary, param\_grid. As always, we instantiate our model. We then call GridSearchCV and pass it our model, the grid we wish to tune over and set cv equal to kf. This returns a GridSearch object that we can then fit to the training data, and this fit performs the actual cross-validated grid search. We can then print the model's attributes best-params-underscore and best-score-underscore, respectively, to retrieve the hyperparameters that perform the best along with the mean cross-validation score over that fold.

## 8. Limitations and an alternative approach

02:40 - 03:12

Grid search is great. However, the number of fits is equal to the number of hyperparameters multiplied by the number of values multiplied by the number of folds. Therefore, it doesn't scale well! So, performing 3-fold cross-validation for one hyperparameter with 10 values each means 30 fits, while 10-fold cross-validation on 3 hyperparameters with 10 values each equals 900 fits! However, there is another way.

## 9. RandomizedSearchCV

03:12 - 04:01

We can perform a random search, which picks random hyperparameter values rather than exhaustively searching through all options. Let's demonstrate this approach. We import RandomizedSearchCV from sklearn-dot-model\_selection. We set up KFold and param\_grid, and instantiate the model as before. We call RandomizedSearchCV using the same arguments and variables as GridSearchCV. We can optionally set the n\_iter argument, which determines the number of hyperparameter values tested. So five-fold cross-validation with n\_iter set to two performs 10 fits. Again we can access the best hyperparameters and their score. In this case it is able to find the best hyperparameters from our previous grid search!

## 10. Evaluating on the test set

04:01 - 04:13

We can evaluate model performance on the test set by passing it to a call of the random search object's dot-score method. It actually performs slightly better than the best score in our grid search!

## 11. Let's practice!

04:13 - 04:18

Now let's perform some hyperparameter tuning!

# Hyperparameter tuning with GridSearchCV

Now you have seen how to perform grid search hyperparameter tuning, you are going to build a lasso regression model with optimal hyperparameters to predict blood glucose levels using the features in the diabetes\_df dataset.

X\_train, X\_test, y\_train, and y\_test have been preloaded for you. A KFold() object has been created and stored for you as kf, along with a lasso regression model as lasso.

##### Instructions

**100 XP**

* Import GridSearchCV.
* Set up a parameter grid for "alpha", using np.linspace() to create 20 evenly spaced values ranging from 0.00001 to 1.
* Call GridSearchCV(), passing lasso, the parameter grid, and setting cv equal to kf.
* Fit the grid search object to the training data to perform a cross-validated grid search.
* # Import GridSearchCV
* \_\_\_\_
* # Set up the parameter grid
* param\_grid = {"\_\_\_\_": np.linspace(\_\_\_\_, \_\_\_\_, \_\_\_\_)}
* # Instantiate lasso\_cv
* lasso\_cv = \_\_\_\_(\_\_\_\_, \_\_\_\_, cv=\_\_\_\_)
* # Fit to the training data
* \_\_\_\_
* print("Tuned lasso paramaters: {}".format(lasso\_cv.best\_params\_))
* print("Tuned lasso score: {}".format(lasso\_cv.best\_score\_))

Well done! Unfortunately, the best model only has an R-squared score of 0.33, highlighting that using the optimal hyperparameters does not guarantee a high performing model!

# Import GridSearchCV

from sklearn.model\_selection import GridSearchCV

# Set up the parameter grid

param\_grid = {"alpha": np.linspace(0.00001, 1, 20)}

# Instantiate lasso\_cv

lasso\_cv = GridSearchCV(lasso, param\_grid, cv=kf)

# Fit to the training data

lasso\_cv.fit(X\_train, y\_train)

print("Tuned lasso paramaters: {}".format(lasso\_cv.best\_params\_))

print("Tuned lasso score: {}".format(lasso\_cv.best\_score\_))

# Import GridSearchCV

from sklearn.model\_selection import GridSearchCV

# Set up the parameter grid

param\_grid = {"alpha": np.linspace(0.00001, 1, 20)}

# Instantiate lasso\_cv

lasso\_cv = GridSearchCV(lasso, param\_grid, cv=kf)

# Fit to the training data

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

print("Tuned lasso paramaters: {}".format(lasso\_cv.best\_params\_))

print("Tuned lasso score: {}".format(lasso\_cv.best\_score\_))

Tuned lasso paramaters: {'alpha': 1e-05}

Tuned lasso score: 0.33078807238121977

<script.py> output:

Tuned lasso paramaters: {'alpha': 1e-05}

Tuned lasso score: 0.33078807238121977

# Hyperparameter tuning with RandomizedSearchCV

As you saw, GridSearchCV can be computationally expensive, especially if you are searching over a large hyperparameter space. In this case, you can use RandomizedSearchCV, which tests a fixed number of hyperparameter settings from specified probability distributions.

Training and test sets from diabetes\_df have been pre-loaded for you as X\_train. X\_test, y\_train, and y\_test, where the target is "diabetes". A logistic regression model has been created and stored as logreg, as well as a KFold variable stored as kf.

You will define a range of hyperparameters and use RandomizedSearchCV, which has been imported from sklearn.model\_selection, to look for optimal hyperparameters from these options.

##### Instructions

**100 XP**

* Create params, adding "l1" and "l2" as penalty values, setting C to a range of 50 float values between 0.1 and 1.0, and class\_weight to either "balanced" or a dictionary containing 0:0.8, 1:0.2.
* Create the Randomized Search CV object, passing the model and the parameters, and setting cv equal to kf.
* Fit logreg\_cv to the training data.
* Print the model's best parameters and accuracy score.
* # Create the parameter space
* params = {"penalty": ["\_\_\_\_", "\_\_\_\_"],
* "tol": np.linspace(0.0001, 1.0, 50),
* "C": np.linspace(\_\_\_\_, \_\_\_\_, \_\_\_\_),
* "class\_weight": ["\_\_\_\_", {0:\_\_\_\_, 1:\_\_\_\_}]}
* # Instantiate the RandomizedSearchCV object
* logreg\_cv = \_\_\_\_(\_\_\_\_, \_\_\_\_, cv=\_\_\_\_)
* # Fit the data to the model
* logreg\_cv.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* # Print the tuned parameters and score
* print("Tuned Logistic Regression Parameters: {}".format(\_\_\_\_.\_\_\_\_))
* print("Tuned Logistic Regression Best Accuracy Score: {}".format(\_\_\_\_.\_\_\_\_))

# Create the parameter space

params = {"penalty": ["l1", "l2"],

         "tol": np.linspace(0.0001, 1.0, 50),

         "C": np.linspace(0.1, 1.0, 50),

         "class\_weight": ["balanced", {0:0.8, 1:0.2}]}

# Instantiate the RandomizedSearchCV object

logreg\_cv = RandomizedSearchCV  (logreg, params, cv=kf)

# Fit the data to the model

logreg\_cv.fit(X\_train, y\_train)

# Print the tuned parameters and score

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

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

**# Create the parameter space**

**params = {"penalty": ["l1", "l2"],**

**"tol": np.linspace(0.0001, 1.0, 50),**

**"C": np.linspace(0.1, 1.0, 50),**

**"class\_weight": ["balanced", {0:0.8, 1:0.2}]}**

**# Instantiate the RandomizedSearchCV object**

**logreg\_cv = RandomizedSearchCV (logreg, params, cv=kf)**

**# Fit the data to the model**

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

**# Print the tuned parameters and score**

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

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

**Tuned Logistic Regression Parameters: {'tol': 0.14294285714285712, 'penalty': 'l2', 'class\_weight': 'balanced', 'C': 0.6326530612244898}**

**Tuned Logistic Regression Best Accuracy Score: 0.7460082633613221**

**<script.py> output:**

**Tuned Logistic Regression Parameters: {'tol': 0.14294285714285712, 'penalty': 'l2', 'class\_weight': 'balanced', 'C': 0.6326530612244898}**

**Tuned Logistic Regression Best Accuracy Score: 0.7460082633613221**

**Great searching! Even without exhaustively trying every combination of hyperparameters, the model has an accuracy of over 70% on the test set! So far we have worked with clean datasets; however, in the next chapter, we will discuss the steps required to transform messy data before building supervised learning models.**

## 1. Preprocessing data

00:00 - 00:03

Welcome to the final chapter of the course!

## 2. scikit-learn requirements

00:03 - 00:21

Recall that scikit-learn requires numeric data, with no missing values. All the data that we have used so far has been in this format. However, with real-world data, this will rarely be the case, and instead we need to preprocess our data before we can build models.

## 3. Dealing with categorical features

00:21 - 00:47

Say we have a dataset containing categorical features, such as color. As these are not numeric, scikit-learn will not accept them and we need to convert them into numeric features. We achieve this by splitting the feature into multiple binary features called dummy variables, one for each category. Zero means the observation was not that category, while one means it was.

## 4. Dummy variables

00:47 - 00:56

Say we are working with a music dataset that has a genre feature with ten values such as Electronic, Hip-Hop, and Rock.

## 5. Dummy variables

00:56 - 01:17

We create binary features for each genre. As each song has one genre, each row will have a 1 in one of the ten columns and zeros in the rest. If a song is not any of the first nine genres, then implicitly, it is a rock song. That means we only need nine features, so we can

## 6. Dummy variables

01:17 - 01:25

delete the Rock column. If we do not do this, we are duplicating information, which might be an issue for some models.

## 7. Dealing with categorical features in Python

01:25 - 01:34

To create dummy variables we can use scikit-learn's OneHotEncoder, or pandas' get\_dummies. We will use get\_dummies.

## 8. Music dataset

01:34 - 01:53

We will be working with a music dataset in this chapter, for both classification and regression problems. Initially, we will build a regression model using all features in the dataset to predict song popularity. There is one categorical feature, genre, with ten possible values.

## 9. EDA w/ categorical feature

01:53 - 02:01

This box plot shows how popularity varies by genre. Let's encode this feature using dummy variables.

## 10. Encoding dummy variables

02:01 - 02:53

We import pandas, read in the DataFrame, and call pd-dot-get\_dummies, passing the categorical column. As we only need to keep nine out of our ten binary features, we can set the drop\_first argument to True. Printing the first five rows, we see pandas creates nine new binary features. The first song is Jazz, and the second is Rap, indicated by a 1 in the respective columns. To bring these binary features back into our original DataFrame we can use pd-dot-concat, passing a list containing the music DataFrame and our dummies DataFrame, and setting axis equal to one. Lastly, we can remove the original genre column using df-dot-drop, passing the column, and setting axis equal to one.

## 11. Encoding dummy variables

02:53 - 03:23

If the DataFrame only has one categorical feature, we can pass the entire DataFrame, thus skipping the step of combining variables. If we don't specify a column, the new DataFrame's binary columns will have the original feature name prefixed, so they will start with genre-underscore - as shown here. Notice the original genre column is automatically dropped. Once we have dummy variables, we can fit models as before.

## 12. Linear regression with dummy variables

03:23 - 04:08

Using the music\_dummies DataFrame, the process for creating training and test sets remains unchanged. To perform cross-validation we then create a KFold object, instantiate a linear regression model, and call cross\_val\_score. We set scoring equal to neg\_mean\_squared\_error, which returns the negative MSE. This is because scikit-learn's cross-validation metrics presume a higher score is better, so MSE is changed to negative to counteract this. We can calculate the training RMSE by taking the square root and converting to positive, achieved by calling numpy-dot-square-root and passing our scores with a minus sign in front.

## 13. Let's practice!

04:08 - 04:13

Now let's practice working with categorical features.

# Creating dummy variables

Being able to include categorical features in the model building process can enhance performance as they may add information that contributes to prediction accuracy.

The music\_df dataset has been preloaded for you, and its shape is printed. Also, pandas has been imported as pd.

Now you will create a new DataFrame containing the original columns of music\_df plus dummy variables from the "genre" column.

##### Instructions

**100 XP**

* Use a relevant function, passing the entire music\_df DataFrame, to create music\_dummies, dropping the first binary column.
* Print the shape of music\_dummies.
* # Create music\_dummies
* music\_dummies = \_\_\_\_
* # Print the new DataFrame's shape
* print("Shape of music\_dummies: {}".format(\_\_\_\_))

# Create music\_dummies

music\_dummies = pd.get\_dummies(music\_df, drop\_first=True)

# Print the new DataFrame's shape

print("Shape of music\_dummies: {}".format(music\_dummies.shape))

# Create music\_dummies

music\_dummies = pd.get\_dummies(music\_df, drop\_first=True)

# Print the new DataFrame's shape

print("Shape of music\_dummies: {}".format(music\_dummies.shape))

Shape of music\_dummies: (1000, 20)

<script.py> output:

Shape of music\_dummies: (1000, 20)

As there were ten values in the "genre" column, nine new columns were added by a call of pd.get\_dummies() using drop\_first=True. After dropping the original "genre" column, there are still eight new columns in the DataFrame!

# Regression with categorical features

Now you have created music\_dummies, containing binary features for each song's genre, it's time to build a ridge regression model to predict song popularity.

music\_dummies has been preloaded for you, along with Ridge, cross\_val\_score, numpy as np, and a KFold object stored as kf.

The model will be evaluated by calculating the average RMSE, but first, you will need to convert the scores for each fold to positive values and take their square root. This metric shows the average error of our model's predictions, so it can be compared against the standard deviation of the target value—"popularity".

##### Instructions

**100 XP**

* Create X, containing all features in music\_dummies, and y, consisting of the "popularity" column, respectively.
* Instantiate a ridge regression model, setting alpha equal to 0.2.
* Perform cross-validation on X and y using the ridge model, setting cv equal to kf, and using negative mean squared error as the scoring metric.
* Print the RMSE values by converting negative scores to positive and taking the square root.
* # Create X and y
* X = \_\_\_\_
* y = \_\_\_\_
* # Instantiate a ridge model
* ridge = \_\_\_\_
* # Perform cross-validation
* scores = \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_, cv=\_\_\_\_, scoring="\_\_\_\_")
* # Calculate RMSE
* rmse = np.\_\_\_\_(\_\_\_\_)
* print("Average RMSE: {}".format(np.mean(rmse)))
* print("Standard Deviation of the target array: {}".format(np.std(y)))

# Create X and y

X = music\_dummies.drop('popularity', axis=1).values

y = music\_dummies['popularity'].values

# Instantiate a ridge model

ridge = Ridge(alpha=0.2)

# Perform cross-validation

scores = cross\_val\_score(ridge, X, y, cv=kf, scoring="neg\_mean\_squared\_error")

# Calculate RMSE

rmse = np.sqrt(-scores)

print("Average RMSE: {}".format(np.mean(rmse)))

print("Standard Deviation of the target array: {}".format(np.std(y)))

Average RMSE: 8.236853840202299

Standard Deviation of the target array: 14.02156909907019

<script.py> output:

Average RMSE: 8.236853840202299

Standard Deviation of the target array: 14.02156909907019

Great work! An average RMSE of approximately 8.24 is lower than the standard deviation of the target variable (song popularity), suggesting the model is reasonably accurate.

## 1. Handling missing data

00:00 - 00:03

Now let's look at how to handle missing data.

## 2. Missing data

00:03 - 00:17

When there is no value for a feature in a particular row, we call it missing data. This can happen because there was no observation or the data might be corrupt. Whatever the reason, we need to deal with it.

## 3. Music dataset

00:17 - 00:37

Previously we worked with a modified music dataset. Now let's inspect the original version, which contains one thousand rows. We do this by chaining pandas' dot-isna with dot-sum and dot-sort\_values. Each feature is missing between 8 and 200 values!

## 4. Dropping missing data

00:37 - 01:04

A common approach is to remove missing observations accounting for less than 5% of all data. To do this, we use pandas' dot-dropna method, passing a list of columns with less than 5% missing values to the subset argument. If there are missing values in our subset column, the entire row is removed. Rechecking the DataFrame, we see fewer missing values.

## 5. Imputing values

01:04 - 01:36

Another option is to impute missing data. This means making an educated guess as to what the missing values could be. We can impute the mean of all non-missing entries for a given feature. We can also use other values like the median. For categorical values we commonly impute the most frequent value. Note we must split our data before imputing to avoid leaking test set information to our model, a concept known as data leakage.

## 6. Imputation with scikit-learn

01:36 - 02:39

Here is a workflow for imputation to predict song popularity. We import SimpleImputer from sklearn-dot-impute. As we will use different imputation methods for categorical and numeric features, we first split them, storing as X\_cat and X\_num respectively, along with our target array as y. We create categorical training and test sets. We repeat this for the numeric features. By using the same value for the random\_state argument, the target arrays' values remain unchanged. To impute missing categorical values we instantiate a SimpleImputer, setting strategy as most frequent. By default, SimpleImputer expects NumPy-dot-NaN to represent missing values. Now we call dot-fit\_transform to impute the training categorical features' missing values! For the test categorical features, we call dot-transform.

## 7. Imputation with scikit-learn

02:39 - 03:12

For our numeric data, we instantiate another imputer. By default, it fills values with the mean. We fit and transform the training features, and transform the test features. We then combine our training data using numpy-dot-append, passing our two arrays, and set axis equal to 1. We repeat this for our test data. Due to their ability to transform our data, imputers are known as transformers.

## 8. Imputing within a pipeline

03:12 - 03:51

We can also impute using a pipeline, which is an object used to run a series of transformations and build a model in a single workflow. To do this, we import Pipeline from sklearn-dot-pipeline. Here we perform binary classification to predict whether a song is rock or another genre. We drop missing values accounting for less than five percent of our data. We convert values in the genre column, which will be the target, to a 1 if Rock, else 0, using numpy-dot-where. We then create X and y.

## 9. Imputing within a pipeline

03:51 - 04:22

To build a pipeline we construct a list of steps containing tuples with the step names specified as strings, and instantiate the transformer or model. We pass this list when instantiating a Pipeline. We then split our data, and fit the pipeline to the training data, as with any other model. Finally, we compute accuracy. Note that, in a pipeline, each step but the last must be a transformer.

## 10. Let's practice!

04:22 - 04:28

Now let's create a pipeline to handle missing data and build a model!

# Dropping missing data

Over the next three exercises, you are going to tidy the music\_df dataset. You will create a pipeline to impute missing values and build a KNN classifier model, then use it to predict whether a song is of the "Rock" genre.

In this exercise specifically, you will drop missing values accounting for less than 5% of the dataset, and convert the "genre" column into a binary feature.

##### Instructions 1/3

* Print the number of missing values for each column in the music\_df dataset, sorted in ascending order.
* # Print missing values for each column
* \_\_\_\_
* Print the number of missing values for each column in the music\_df dataset, sorted in ascending order.

**# Print missing values for each column**

**print(music\_df.isna().sum().sort\_values())**

<script.py> output:

genre 8

popularity 31

loudness 44 liveness 46

tempo 46

speechiness 59

duration\_ms 91 instrumentalness 91

danceability 143

valence 143

acousticness 200

energy 200

dtype: int64

Remove values for all columns with 50 or fewer missing values.

# Print missing values for each column

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.\_\_\_\_(subset=[\_\_\_\_])

# Print missing values for each column

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=['genre', 'popularity', 'loudness', 'liveness', 'tempo'])

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=['genre', 'popularity', 'loudness', 'liveness', 'tempo'])

genre 8

popularity 31

loudness 44

liveness 46

tempo 46

speechiness 59

duration\_ms 91

instrumentalness 91

danceability 143

valence 143

acousticness 200

energy 200

dtype: int64

**Convert music\_df["genre"] to values of 1 if the row contains "Rock", otherwise change the value to 0.**

# Print missing values for each column

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])

# Convert genre to a binary feature

music\_df["genre"] = np.where(\_\_\_\_["\_\_\_\_"] == "\_\_\_\_", \_\_\_\_, \_\_\_\_)

print(music\_df.isna().sum().sort\_values())

print("Shape of the `music\_df`: {}".format(music\_df.shape))

# Print missing values for each column

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])

# Convert genre to a binary feature

music\_df["genre"] = np.where(music\_df["genre"] == "Rock", 1,0)

print(music\_df.isna().sum().sort\_values())

print("Shape of the `music\_df`: {}".format(music\_df.shape))

# Print missing values for each column

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])

# Convert genre to a binary feature

music\_df["genre"] = np.where(music\_df["genre"] == "Rock", 1,0)

print(music\_df.isna().sum().sort\_values())

print("Shape of the `music\_df`: {}".format(music\_df.shape))

genre 8

popularity 31

loudness 44

liveness 46

tempo 46

speechiness 59

duration\_ms 91

instrumentalness 91

danceability 143

valence 143

acousticness 200

energy 200

dtype: int64

popularity 0

liveness 0

loudness 0

tempo 0

genre 0

duration\_ms 29

instrumentalness 29

speechiness 53

danceability 127

valence 127

acousticness 178

energy 178

dtype: int64

Shape of the `music\_df`: (892, 12)

Well done! The dataset has gone from 1000 observations down to 892, but it is now in the correct format for binary classification and the remaining missing values can be imputed as part of a pipeline.

# Print missing values for each column

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])

# Convert genre to a binary feature

music\_df["genre"] = np.where(music\_df["genre"] == "Rock", 1,0)

print(music\_df.isna().sum().sort\_values())

print("Shape of the `music\_df`: {}".format(music\_df.shape))

Print missing values for each column

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])

# Convert genre to a binary feature

music\_df["genre"] = np.where(music\_df["genre"] == "Rock", 1,0)

print(music\_df.isna().sum().sort\_values())

print("Shape of the `music\_df`: {}".format(music\_df.shape))

genre 8

popularity 31

loudness 44

liveness 46

tempo 46

speechiness 59

duration\_ms 91

instrumentalness 91

danceability 143

valence 143

acousticness 200

energy 200

dtype: int64

popularity 0

liveness 0

loudness 0

tempo 0

genre 0

duration\_ms 29

instrumentalness 29

speechiness 53

danceability 127

valence 127

acousticness 178

energy 178

dtype: int64

Shape of the `music\_df`: (892, 12)

<script.py> output:

genre 8

popularity 31

loudness 44

liveness 46

tempo 46

speechiness 59

duration\_ms 91

instrumentalness 91

danceability 143

valence 143

acousticness 200

energy 200

dtype: int64

popularity 0

liveness 0

loudness 0

tempo 0

genre 0

duration\_ms 29

instrumentalness 29

speechiness 53

danceability 127

valence 127

acousticness 178

energy 178

dtype: int64

Shape of the `music\_df`: (892, 12)

# Pipeline for song genre prediction: I

Now it's time to build a pipeline. It will contain steps to impute missing values using the mean for each feature and build a KNN model for the classification of song genre.

The modified music\_df dataset that you created in the previous exercise has been preloaded for you, along with KNeighborsClassifier and train\_test\_split.

##### Instructions

**100 XP**

* Import SimpleImputer and Pipeline.
* Instantiate an imputer.
* Instantiate a KNN classifier with three neighbors.
* Create steps, a list of tuples containing the imputer variable you created, called "imputer", followed by the knn model you created, called "knn".
* # Import modules
* \_\_\_\_
* \_\_\_\_
* # Instantiate an imputer
* imputer = \_\_\_\_()
* # Instantiate a knn model
* knn = \_\_\_\_
* # Build steps for the pipeline
* steps = [("\_\_\_\_", \_\_\_\_),
* ("\_\_\_\_", \_\_\_\_)]

# Import modules

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

# Instantiate an imputer

imputer = SimpleImputer()

# Instantiate a knn model

knn = KNeighborsClassifier(n\_neighbors=3)

# Build steps for the pipeline

steps = [("imputer", imputer),

         ("knn", knn)]

# Import modules from sklearn.impute import SimpleImputer from sklearn.pipeline import Pipeline # Instantiate an imputer imputer = SimpleImputer() # Instantiate a knn model knn = KNeighborsClassifier(n\_neighbors=3) # Build steps for the pipeline steps = [("imputer", imputer), ("knn", knn)]

Perfect pipeline skills! You are now ready to build and evaluate a song genre classification model.

# Pipeline for song genre prediction: II

Having set up the steps of the pipeline in the previous exercise, you will now use it on the music\_df dataset to classify the genre of songs. What makes pipelines so incredibly useful is the simple interface that they provide.

X\_train, X\_test, y\_train, and y\_test have been preloaded for you, and confusion\_matrix has been imported from sklearn.metrics.

##### Instructions

**100 XP**

* Create a pipeline using the steps you previously defined.
* Fit the pipeline to the training data.
* Make predictions on the test set.
* Calculate and print the confusion matrix.
* steps = [("imputer", imp\_mean),
* ("knn", knn)]
* # Create the pipeline
* pipeline = \_\_\_\_(\_\_\_\_)
* # Fit the pipeline to the training data
* \_\_\_\_
* # Make predictions on the test set
* y\_pred = \_\_\_\_
* # Print the confusion matrix
* print(\_\_\_\_(\_\_\_\_, \_\_\_\_))

steps = [("imputer", imp\_mean),

        ("knn", knn)]

# Create the pipeline

pipeline = Pipeline(steps)

# Fit the pipeline to the training data

pipeline.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = pipeline.predict(X\_test)

# Print the confusion matrix

steps = [("imputer", imp\_mean),

("knn", knn)]

# Create the pipeline

pipeline = Pipeline(steps)

# Fit the pipeline to the training data

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

# Make predictions on the test set

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

# Print the confusion matrix

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

[[79 9]

[ 4 82]]

<script.py> output:

[[79 9]

[ 4 82]]

Excellent! See how easy it is to scale our model building workflow using pipelines. In this case, the confusion matrix highlights that the model had 79 true positives and 82 true negatives!

## 1. Centering and scaling

00:00 - 00:10

Data imputation is one of several important preprocessing steps for machine learning. Now let's cover another: centering and scaling our data.

## 2. Why scale our data?

00:10 - 00:33

Let's use df-dot-describe to check out the ranges of some of our feature variables in the music dataset. We see that the ranges vary widely: duration\_ms ranges from zero to one-point-six-two million, speechiness contains only decimal places, and loudness only has negative values!

## 3. Why scale our data?

00:33 - 01:03

Many machine learning models use some form of distance to inform them, so if we have features on far larger scales, they can disproportionately influence our model. For example, KNN uses distance explicitly when making predictions. For this reason, we actually want features to be on a similar scale. To achieve this, we can normalize or standardize our data, often referred to as scaling and centering.

## 4. How to scale our data

01:03 - 01:42

There are several ways to scale our data: given any column, we can subtract the mean and divide by the variance so that all features are centered around zero and have a variance of one. This is called standardization. We can also subtract the minimum and divide by the range of the data so the normalized dataset has minimum zero and maximum one. Or, we can center our data so that it ranges from -1 to 1 instead. In this video, we will perform standardization, but scikit-learn has functions available for other types of scaling.

## 5. Scaling in scikit-learn

01:42 - 02:17

To scale our features, we import StandardScaler from sklearn-dot-preprocessing. We create our feature and target arrays. Before scaling, we split our data to avoid data leakage. We then instantiate a StandardScaler object, and call its fit\_transform method, passing our training features. Next, we use scaler-dot-transform on the test features. Looking at the mean and standard deviation of the columns of both the original and scaled data verifies the change has taken place.

## 6. Scaling in a pipeline

02:17 - 02:42

We can also put a scaler in a pipeline! Here we build a pipeline object to scale our data and use a KNN model with six neighbors. We then split our data, fit the pipeline to our training set, and predict on our test set. Computing the accuracy yields a result of zero-point-eight-one. Let's compare this to using unscaled data.

## 7. Comparing performance using unscaled data

02:42 - 02:57

Here we fit a KNN model to our unscaled training data and print the accuracy. It is only zero-point-five-three, so just by scaling our data we improved accuracy by over 50 percent!

## 8. CV and scaling in a pipeline

02:57 - 03:52

Let's also look at how we can use cross-validation with a pipeline. We first build our pipeline. We then specify our hyperparameter space by creating a dictionary: the keys are the pipeline step name followed by a double underscore, followed by the hyperparameter name. The corresponding value is a list or an array of the values to try for that particular hyperparameter. In this case, we are tuning n\_neighbors in the KNN model. Next we split our data into training and test sets. We then perform a grid search over our parameters by instantiating the GridSearchCV object, passing our pipeline and setting the param\_grid argument equal to parameters. We then fit it to our training data. Lastly, we make predictions using our test set.

## 9. Checking model parameters

03:52 - 04:04

Printing GridSearchCV's best\_score\_ attribute, we see the score is very slightly better than our previous model's performance. Printing the best parameters, the optimal model has 12 neighbors.

## 10. Let's practice!

04:04 - 04:11

Now let's incorporate scaling into our supervised learning workflow!

**Daily XP150**

##### Exercise

##### Exercise

# Centering and scaling for regression

Now you have seen the benefits of scaling your data, you will use a pipeline to preprocess the music\_df features and build a lasso regression model to predict a song's loudness.

X\_train, X\_test, y\_train, and y\_test have been created from the music\_df dataset, where the target is "loudness" and the features are all other columns in the dataset. Lasso and Pipeline have also been imported for you.

Note that "genre" has been converted to a binary feature where 1 indicates a rock song, and 0 represents other genres.

##### Instructions

**100 XP**

* Import StandardScaler.
* Create the steps for the pipeline object, a StandardScaler object called "scaler", and a lasso model called "lasso" with alpha set to 0.5.
* Instantiate a pipeline with steps to scale and build a lasso regression model.
* Calculate the R-squared value on the test data.
* # Import StandardScaler
* \_\_\_\_
* # Create pipeline steps
* steps = [("\_\_\_\_", \_\_\_\_()),
* ("\_\_\_\_", \_\_\_\_(alpha=\_\_\_\_))]
* # Instantiate the pipeline
* pipeline = \_\_\_\_(\_\_\_\_)
* pipeline.fit(X\_train, y\_train)
* # Calculate and print R-squared
* print(\_\_\_\_.\_\_\_\_(\_\_\_\_, \_\_\_\_))

# Import StandardScaler

from sklearn.preprocessing import StandardScaler

# Create pipeline steps

steps = [("scaler", StandardScaler()),

         ("lasso", Lasso(alpha=0.5))]

# Instantiate the pipeline

pipeline = Pipeline(steps)

pipeline.fit(X\_train, y\_train)

# Calculate and print R-squared

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

# Import StandardScaler

from sklearn.preprocessing import StandardScaler

# Create pipeline steps

steps = [("scaler", StandardScaler()),

("lasso", Lasso(alpha=0.5))]

# Instantiate the pipeline

pipeline = Pipeline(steps)

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

# Calculate and print R-squared

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

0.6193523316282489

<script.py> output:

0.6193523316282489

Awesome scaling! The model may have only produced an R-squared of 0.619, but without scaling this exact model would have only produced a score of 0.35, which proves just how powerful scaling can be!

##### Exercise

##### Exercise

# Centering and scaling for classification

Now you will bring together scaling and model building into a pipeline for cross-validation.

Your task is to build a pipeline to scale features in the music\_df dataset and perform grid search cross-validation using a logistic regression model with different values for the hyperparameter C. The target variable here is "genre", which contains binary values for rock as 1 and any other genre as 0.

StandardScaler, LogisticRegression, and GridSearchCV have all been imported for you.

##### Instructions

**100 XP**

* Build the steps for the pipeline: a StandardScaler() object named "scaler", and a logistic regression model named "logreg".
* Create the parameters, searching 20 equally spaced float values ranging from 0.001 to 1.0 for the logistic regression model's C hyperparameter within the pipeline.
* Instantiate the grid search object.
* Fit the grid search object to the training data.
* # Build the steps
* steps = [("\_\_\_\_", \_\_\_\_()),
* ("\_\_\_\_", \_\_\_\_())]
* pipeline = Pipeline(steps)
* # Create the parameter space
* parameters = {"\_\_\_\_": np.\_\_\_\_(\_\_\_\_, \_\_\_\_, 20)}
* X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,
* random\_state=21)
* # Instantiate the grid search object
* cv = \_\_\_\_(\_\_\_\_, param\_grid=\_\_\_\_)
* # Fit to the training data
* cv.\_\_\_\_(\_\_\_\_, \_\_\_\_)
* print(cv.best\_score\_, "\n", cv.best\_params\_)
* # Build the steps
* steps = [("scaler", StandardScaler()),
* ("logreg", LogisticRegression())]
* pipeline = Pipeline(steps)
* # Create the parameter space
* parameters = {"logreg\_\_C": np.linspace(0.001, 1.0, 20)}
* X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,
* random\_state=21)
* # Instantiate the grid search object
* cv = GridSearchCV(pipeline, param\_grid=parameters)
* # Fit to the training data
* cv.fit(X\_train, y\_train)
* print(cv.best\_score\_, "\n", cv.best\_params\_)
* 0.8425
* {'logreg\_\_C': 0.1061578947368421}
* <script.py> output:
* 0.8425
* {'logreg\_\_C': 0.1061578947368421}

###### Well done! Using a pipeline shows that a logistic regression model with "C" set to approximately 0.1 produces a model with 0.8425 accuracy!

# Build the steps

steps = [("scaler", StandardScaler()),

("logreg", LogisticRegression())]

pipeline = Pipeline(steps)

# Create the parameter space

parameters = {"logreg\_\_C": np.linspace(0.001, 1.0, 20)}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,

random\_state=21)

# Instantiate the grid search object

cv = GridSearchCV(pipeline, param\_grid=parameters)

# Fit to the training data

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

print(cv.best\_score\_, "\n", cv.best\_params\_)

0.8425

{'logreg\_\_C': 0.1061578947368421}

<script.py> output:

0.8425

{'logreg\_\_C': 0.1061578947368421}

###### Hint

* Enter the names of your steps as strings in the first part of each tuple in steps.
* Pass a string the model name followed by two underscores, then the hyperparameter name when creating parameters; for example the C parameter would be logreg\_\_C.
* Pass pipeline when instantiating GridSearchCV, setting param\_grid equal to the parameter space you created.
* # Build the steps
* steps = [("scaler", StandardScaler()),
* ("logreg", LogisticRegression())]
* pipeline = Pipeline(steps)
* # Create the parameter space
* parameters = {"logreg\_\_C": np.arange(0.001, 1.0, 20)}
* X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,
* random\_state=21)
* # Instantiate the grid search object
* cv = GridSearchCV(pipeline, param\_grid=parameters)
* # Fit to the training data
* cv.fit(X\_train, y\_train)
* print(cv.best\_score\_, "\n", cv.best\_params\_)
* 0.7849999999999999
* {'logreg\_\_C': 0.001}
* <script.py> output:
* 0.7849999999999999
* {'logreg\_\_C': 0.001}
* <script.py> output:
* 0.7849999999999999
* {'logreg\_\_C': 0.001}

## 1. Evaluating multiple models

00:00 - 00:07

We've covered all parts of the supervised learning workflow. But how do we decide which model to use in the first place?

## 2. Different models for different problems

00:07 - 01:02

This is a complex question, and the answer depends on our situation. However, there are some principles that can guide us when making this decision. The size of our dataset plays a role. Fewer features means a simpler model, and can reduce training time. Also, some models, such as Artificial Neural Networks, require a lot of data to perform well. We may need an interpretable model, so we can explain to stakeholders how predictions were made. An example is linear regression, where we can calculate and interpret the model coefficients. Alternatively, flexibility might be important to get the most accurate predictions. Generally, flexible models make fewer assumptions about the data; for example, a KNN model does not assume a linear relationship between the features and the target.

## 3. It's all in the metrics

01:02 - 01:35

Notice that scikit-learn allows the same methods to be used for most models. This makes it easy to compare them! Regression models can be evaluated using the root mean squared error, or the R-squared value. Likewise, classification models can all be analyzed using accuracy, a confusion matrix and its associated metrics, or the ROC AUC. Therefore, one approach is to select several models and a metric, then evaluate their performance without any form of hyperparameter tuning.

## 4. A note on scaling

01:35 - 01:51

Recall that the performance of some models, such as KNN, linear regression, and logistic regression, are affected by scaling our data. Therefore, it is generally best to scale our data before evaluating models out of the box.

## 5. Evaluating classification models

01:51 - 02:34

We will evaluate three models for binary classification of song genre: KNN, logistic regression, and a new model called a decision tree classifier. We import our required modules, including DecisionTreeClassifier from sklearn-dot-tree. The workings of decision trees are outside the scope of this course, but the steps for building this model are the same as for other models in scikit-learn. As usual, we create our feature and target arrays, then split our data. We then scale our features using the scaler's dot-fit\_transform method on the training set, and the dot-transform method on the test set.

## 6. Evaluating classification models

02:34 - 03:30

We create a dictionary with our model names as strings for the keys, and instantiate models as the dictionary's values. We also create an empty list to store the results. Now we loop through the models in our models dictionary, using its dot-values method. Inside the loop, we instantiate a KFold object. Next we perform cross-validation, using the model being iterated, along with our scaled training features, and target training array. We set cv equal to our kfold variable. By default, the scoring here will be accuracy. We then append the cross-validation results to our results list. Lastly, outside of the loop, we create a boxplot of our results, and set the labels argument equal to a call of models-dot-keys to retrieve each model's name.

## 7. Visualizing results

03:30 - 03:45

The output shows us the range of cross-validation accuracy scores. We can also see each model's median cross-validation score, represented by the orange line in each box. We can see logistic regression has the best median score.

## 8. Test set performance

03:45 - 04:04

To evaluate on the test set we loop through the names and values of the dictionary using the dot-items method. Inside the loop we fit the model, calculate accuracy, and print it. Logistic regression performs best for this problem if we are using accuracy as the metric.

## 9. Let's practice!

04:04 - 04:11

Now let's choose which models to optimize for our supervised learning problems.

# Visualizing regression model performance

Now you have seen how to evaluate multiple models out of the box, you will build three regression models to predict a song's "energy" levels.

The music\_df dataset has had dummy variables for "genre" added. Also, feature and target arrays have been created, and these have been split into X\_train, X\_test, y\_train, and y\_test.

The following have been imported for you: LinearRegression, Ridge, Lasso, cross\_val\_score, and KFold.

##### Instructions

**100 XP**

* Write a for loop using model as the iterator, and model.values() as the iterable.
* Perform cross-validation on the training features and the training target array using the model, setting cv equal to the KFold object.
* Append the model's cross-validation scores to the results list.
* Create a box plot displaying the results, with the x-axis labels as the names of the models.
* models = {"Linear Regression": LinearRegression(), "Ridge": Ridge(alpha=0.1), "Lasso": Lasso(alpha=0.1)}
* results = []
* # Loop through the models' values
* for \_\_\_\_ in models.values():
* kf = KFold(n\_splits=6, random\_state=42, shuffle=True)
* # Perform cross-validation
* cv\_scores = \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_, cv=\_\_\_\_)
* # Append the results
* \_\_\_\_.\_\_\_\_(\_\_\_\_)
* # Create a box plot of the results
* plt.\_\_\_\_(\_\_\_\_, labels=\_\_\_\_.\_\_\_\_())
* plt.show()

models = {"Linear Regression": LinearRegression(), "Ridge": Ridge(alpha=0.1), "Lasso": Lasso(alpha=0.1)}

results = []

# Loop through the models' values

for model in models.values():

  kf = KFold(n\_splits=6, random\_state=42, shuffle=True)

  # Perform cross-validation

  cv\_scores = cross\_val\_score(model, X\_train, y\_train, cv=kf )

  # Append the results

  results.append(cv\_scores)

# Create a box plot of the results

plt.boxplot(results, labels=models.keys())

plt.show()

models = {"Linear Regression": LinearRegression(), "Ridge": Ridge(alpha=0.1), "Lasso": Lasso(alpha=0.1)} results = [] # Loop through the models' values for model in models.values(): kf = KFold(n\_splits=6, random\_state=42, shuffle=True) # Perform cross-validation cv\_scores = cross\_val\_score(model, X\_train, y\_train, cv=kf ) # Append the results results.append(cv\_scores) # Create a box plot of the results plt.boxplot(results, labels=models.keys()) plt.show()

Nicely done! Lasso regression is not a good model for this problem, while linear regression and ridge perform fairly equally. Let's make predictions on the test set, and see if the RMSE can guide us on model selection.

# Predicting on the test set

In the last exercise, linear regression and ridge appeared to produce similar results. It would be appropriate to select either of those models; however, you can check predictive performance on the test set to see if either one can outperform the other.

You will use root mean squared error (RMSE) as the metric. The dictionary models, containing the names and instances of the two models, has been preloaded for you along with the training and target arrays X\_train\_scaled, X\_test\_scaled, y\_train, and y\_test.

##### Instructions

**100 XP**

* Import mean\_squared\_error.
* Fit the model to the scaled training features and the training labels.
* Make predictions using the scaled test features.
* Calculate RMSE by passing the test set labels and the predicted labels.
* # Import mean\_squared\_error
* from \_\_\_\_.\_\_\_\_ import \_\_\_\_
* for name, model in models.items():
* # Fit the model to the training data
* \_\_\_\_
* # Make predictions on the test set
* y\_pred = \_\_\_\_
* # Calculate the test\_rmse
* test\_rmse = \_\_\_\_(\_\_\_\_, \_\_\_\_, squared=\_\_\_\_)
* print("{} Test Set RMSE: {}".format(name, test\_rmse))

# Import mean\_squared\_error

from sklearn.metrics import mean\_squared\_error

for name, model in models.items():

  # Fit the model to the training data

  model.fit(X\_train\_scaled, y\_train)

  # Make predictions on the test set

  y\_pred = model.predict(X\_test\_scaled)

  # Calculate the test\_rmse

  test\_rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)

  print("{} Test Set RMSE: {}".format(name, test\_rmse))

# Import mean\_squared\_error

from sklearn.metrics import mean\_squared\_error

for name, model in models.items():

# Fit the model to the training data

model.fit(X\_train\_scaled, y\_train)

# Make predictions on the test set

y\_pred = model.predict(X\_test\_scaled)

# Calculate the test\_rmse

test\_rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)

print("{} Test Set RMSE: {}".format(name, test\_rmse))

Linear Regression Test Set RMSE: 0.11988851505947569

Ridge Test Set RMSE: 0.11987066103299668

<script.py> output:

Linear Regression Test Set RMSE: 0.11988851505947569

Ridge Test Set RMSE: 0.11987066103299668

The linear regression model just edges the best performance, although the difference is a RMSE of 0.00001 for popularity! Now let's look at classification model selection.

# Visualizing classification model performance

In this exercise, you will be solving a classification problem where the "popularity" column in the music\_df dataset has been converted to binary values, with 1 representing popularity more than or equal to the median for the "popularity" column, and 0 indicating popularity below the median.

Your task is to build and visualize the results of three different models to classify whether a song is popular or not.

The data has been split, scaled, and preloaded for you as X\_train\_scaled, X\_test\_scaled, y\_train, and y\_test. Additionally, KNeighborsClassifier, DecisionTreeClassifier, and LogisticRegression have been imported.

##### Instructions

**100 XP**

* Create a dictionary of "Logistic Regression", "KNN", and "Decision Tree Classifier", setting the dictionary's values to a call of each model.
* Loop through the values in models.
* Instantiate a KFold object to perform 6 splits, setting shuffle to True and random\_state to 12.
* Perform cross-validation using the model, the scaled training features, the target training set, and setting cv equal to kf.
* # Create models dictionary
* models = {"\_\_\_\_": \_\_\_\_(), "\_\_\_\_": \_\_\_\_(), "\_\_\_\_": \_\_\_\_()}
* results = []
* # Loop through the models' values
* for model in \_\_\_\_.\_\_\_\_():
* # Instantiate a KFold object
* kf = \_\_\_\_(n\_splits=\_\_\_\_, random\_state=\_\_\_\_, shuffle=\_\_\_\_)
* # Perform cross-validation
* cv\_results = \_\_\_\_(\_\_\_\_, \_\_\_\_, \_\_\_\_, cv=\_\_\_\_)
* results.append(cv\_results)
* plt.boxplot(results, labels=models.keys())
* plt.show()

# Create models dictionary

models = {"Logistic Regression": LogisticRegression(), "KNN": KNeighborsClassifier(), "Decision Tree Classifier": DecisionTreeClassifier()}

results = []

# Loop through the models' values

for model in models.values():

  # Instantiate a KFold object

  kf = KFold(n\_splits=6, random\_state=12, shuffle=True)

  # Perform cross-validation

  cv\_results = cross\_val\_score(model, X\_train\_scaled, y\_train, cv=kf)

  results.append(cv\_results)

plt.boxplot(results, labels=models.keys())

plt.show()

# Create models dictionary models = {"Logistic Regression": LogisticRegression(), "KNN": KNeighborsClassifier(), "Decision Tree Classifier": DecisionTreeClassifier()} results = [] # Loop through the models' values for model in models.values(): # Instantiate a KFold object kf = KFold(n\_splits=6, random\_state=12, shuffle=True) # Perform cross-validation cv\_results = cross\_val\_score(model, X\_train\_scaled, y\_train, cv=kf) results.append(cv\_results) plt.boxplot(results, labels=models.keys()) plt.show()

Looks like logistic regression is the best candidate based on the cross-validation results! Let's wrap up by building a pipeline

# Pipeline for predicting song popularity

For the final exercise, you will build a pipeline to impute missing values, scale features, and perform hyperparameter tuning of a logistic regression model. The aim is to find the best parameters and accuracy when predicting song genre!

All the models and objects required to build the pipeline have been preloaded for you.

##### Instructions

**100 XP**

* Create the steps for the pipeline by calling a simple imputer, a standard scaler, and a logistic regression model.
* Create a pipeline object, and pass the steps variable.
* Instantiate a grid search object to perform cross-validation using the pipeline and the parameters.
* Print the best parameters and compute and print the test set accuracy score for the grid search object.
* # Create steps
* steps = [("imp\_mean", \_\_\_\_()),
* ("scaler", \_\_\_\_()),
* ("logreg", \_\_\_\_())]
* # Set up pipeline
* pipeline = \_\_\_\_(\_\_\_\_)
* params = {"logreg\_\_solver": ["newton-cg", "saga", "lbfgs"],
* "logreg\_\_C": np.linspace(0.001, 1.0, 10)}
* # Create the GridSearchCV object
* tuning = \_\_\_\_(\_\_\_\_, param\_grid=\_\_\_\_)
* tuning.fit(X\_train, y\_train)
* y\_pred = tuning.predict(X\_test)
* # Compute and print performance
* print("Tuned Logistic Regression Parameters: {}, Accuracy: {}".format(\_\_\_\_.\_\_\_\_, \_\_\_\_.\_\_\_\_))

# Create steps

steps = [("imp\_mean", SimpleImputer()),

         ("scaler", StandardScaler()),

         ("logreg", LogisticRegression())]

# Set up pipeline

pipeline = Pipeline(steps)

params = {"logreg\_\_solver": ["newton-cg", "saga", "lbfgs"],

         "logreg\_\_C": np.linspace(0.001, 1.0, 10)}

# Create the GridSearchCV object

tuning = GridSearchCV(pipeline, param\_grid=params)

tuning.fit(X\_train, y\_train)

y\_pred = tuning.predict(X\_test)

# Compute and print performance

print("Tuned Logistic Regression Parameters: {}, Accuracy: {}".format(tuning.best\_params\_, tuning.score(X\_test, y\_test)))

# Create steps

steps = [("imp\_mean", SimpleImputer()),

("scaler", StandardScaler()),

("logreg", LogisticRegression())]

# Set up pipeline

pipeline = Pipeline(steps)

params = {"logreg\_\_solver": ["newton-cg", "saga", "lbfgs"],

"logreg\_\_C": np.linspace(0.001, 1.0, 10)}

# Create the GridSearchCV object

tuning = GridSearchCV(pipeline, param\_grid=params)

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

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

# Compute and print performance

print("Tuned Logistic Regression Parameters: {}, Accuracy: {}".format(tuning.best\_params\_, tuning.score(X\_test, y\_test)))

Tuned Logistic Regression Parameters: {'logreg\_\_C': 0.112, 'logreg\_\_solver': 'newton-cg'}, Accuracy: 0.82

<script.py> output:

Tuned Logistic Regression Parameters: {'logreg\_\_C': 0.112, 'logreg\_\_solver': 'newton-cg'}, Accuracy: 0.82

## 1. Congratulations

00:00 - 00:04

Well done on completing the course, I predicted that you would!

## 2. What you've covered

00:04 - 00:22

To recap, you have learned the fundamentals of using supervised learning techniques to build predictive models for both regression and classification problems. You have learned the concepts of underfitting and overfitting, how to split data, and perform cross-validation.

## 3. What you've covered

00:22 - 00:33

You also learned about data preprocessing techniques, selected which model to build, performed hyperparameter tuning, assessed model performance, and used pipelines!

## 4. Where to go from here?

00:33 - 01:02

We covered several models, but there are plenty of others, so to learn more we recommend checking out some of our courses. We also have courses that dive deeper into topics we introduced, such as preprocessing, or model validation. There are other courses on topics we did not cover, such as feature engineering, and unsupervised learning. Additionally, we have many machine learning projects where you can apply the skills you've learned here!

## 5. Thank you!

01:02 - 01:12

Congratulations again, and thank you for taking the course! I hope you enjoy using scikit-learn for your supervised learning problems from now on!