**Unsupervised Learning**

Unsupervised learning is a class of machine learning techniques for discovering patterns in data. For instance, finding the natural "clusters" of customers based on their purchase histories, or searching for patterns and correlations among these purchases, and using these patterns to express the data in a compressed form. These are examples of unsupervised learning techniques called "clustering" and "dimension reduction".

**Supervised vs unsupervised learning**

Unsupervised learning is defined in opposition to supervised learning. An example of supervised learning is using the measurements of tumors to classify them as benign or cancerous. In this case, the pattern discovery is guided, or "supervised", so that the patterns are as useful as possible for predicting the label: benign or cancerous. Unsupervised learning, in contrast, is learning without labels. It is pure pattern discovery, unguided by a prediction task. You'll start by learning about clustering. But before we begin, let's introduce a dataset and fix some terminology.

**Iris dataset**

The iris dataset consists of the measurements of many iris plants of three different species. There are four measurements: petal length, petal width, sepal length and sepal width. These are the features of the dataset.

1. 1 https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load\_iris.html

**. Arrays, features & samples**

Throughout this course, datasets like this will be written as two-dimensional numpy arrays. The columns of the array will correspond to the features. The measurements for individual plants are the samples of the dataset. These correspond to rows of the array.

**Iris data is 4-dimensional**

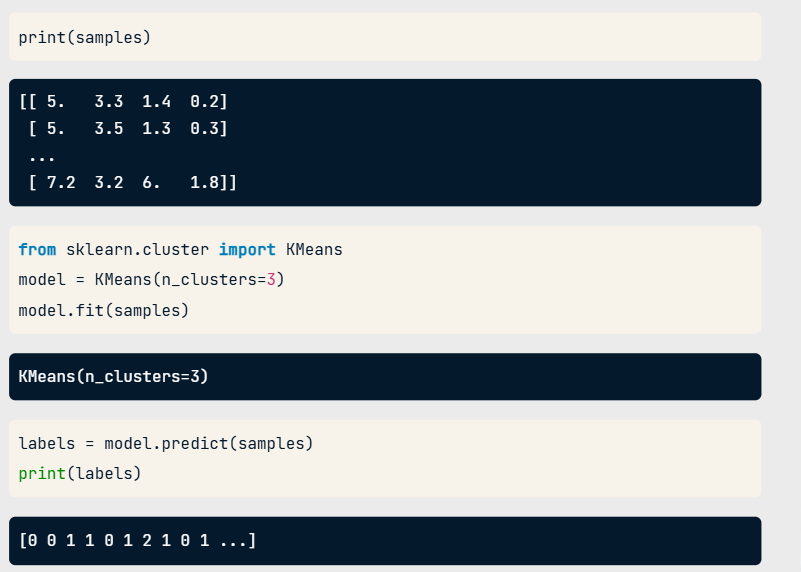
The samples of the iris dataset have four measurements, and so correspond to points in a four-dimensional space. This is the dimension of the dataset. We can't visualize four dimensions directly, but using unsupervised learning techniques we can still gain insight.

**k-means clustering**

In this chapter, we'll cluster these samples using k-means clustering. k-means finds a specified number of clusters in the samples. It's implemented in the scikit-learn or "sklearn" library. Let's see kmeans in action on some samples from the iris dataset.

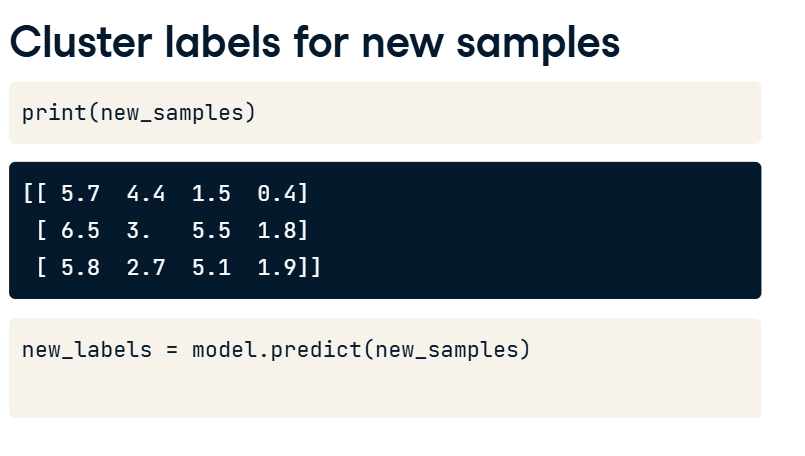
**k-means clustering with scikit-learn**

The iris samples are represented as an array. To start, import kmeans from scikit-learn. Then create a kmeans model, specifying the number of clusters you want to find. Let's specify 3 clusters, since there are three species of iris. Now call the fit method of the model, passing the array of samples. This fits the model to the data, by locating and remembering the regions where the different clusters occur. Then we can use the predict method of the model on these same samples. This returns a cluster label for each sample, indicating to which cluster a sample belongs. Let's assign the result to labels, and print it out.



**Cluster labels for new samples**

If someone comes along with some new iris samples, k-means can determine to which clusters they belong without starting over. k-means does this by remembering the mean (or average) of the samples in each cluster. These are called the "centroids". New samples are assigned to the cluster whose centroid is closest.



**Cluster labels for new samples**

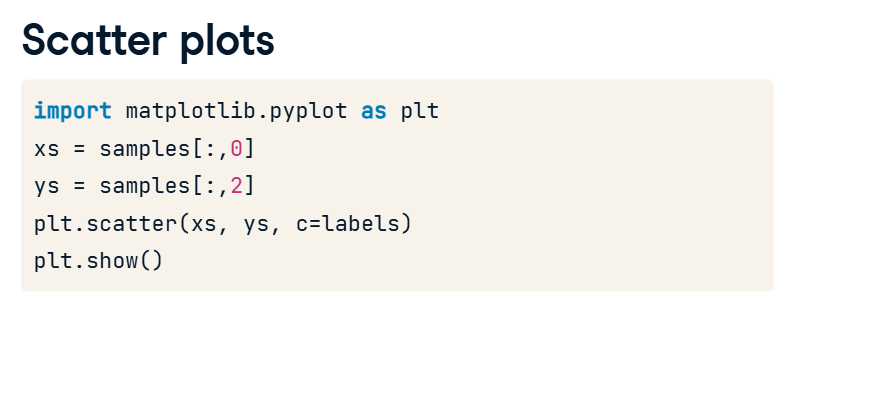
Suppose you've got an array of new samples. To assign the new samples to the existing clusters, pass the array of new samples to the predict method of the kmeans model. This returns the cluster labels of the new samples.

**Scatter plots**

In the next video, you'll learn how to evaluate the quality of your clustering. But for now, let's visualize our clustering of the iris samples using scatter plots. Here is a scatter plot of the sepal length vs petal length of the iris samples. Each point represents an iris sample, and is colored according to the cluster of the sample. To create a scatter plot like this, use PyPlot.

**Scatter plots**

Firstly, import PyPlot. It is conventionally imported as plt. Now get the x- and y- co-ordinates of each sample. Sepal length is in the 0th column of the array, while petal length is in the 2nd column. Now call the plt dot scatter function, passing the x- and y- co-ordinates and specifying c=labels to color by cluster label. When you are ready to show your plot, call plt dot show.



**Evaluating a clustering**

In the previous video, we used k-means to cluster the iris samples into three clusters. But how can we evaluate the quality of this clustering?

A direct approach is to compare the clusters with the iris species. You'll learn about this first, before considering the problem of how to measure the quality of a clustering in a way that doesn't require our samples to come pre-grouped into species. This measure of quality can then be used to make an informed choice about the number of clusters to look for.

**3. Iris: clusters vs species**

Firstly, let's check whether the 3 clusters of iris samples have any correspondence to the iris species. The correspondence is described by this table. There is one column for each of the three species of iris: setosa, versicolor and virginica, and one row for each of the three cluster labels: 0, 1 and 2. The table shows the number of samples that have each possible cluster label/species combination. For example, we see that cluster 1 corresponds perfectly with the species setosa. On the other hand, while cluster 0 contains mainly virginica samples, there are also some virginica samples in cluster 2.

**4. Cross tabulation with pandas**

Tables like these are called "cross-tabulations". To construct one, we are going to use the pandas library. Let's assume the species of each sample is given as a list of strings.

**5. Aligning labels and species**

Import pandas, and then create a two-column DataFrame, where the first column is cluster labels and the second column is the iris species, so that each row gives the cluster label and species of a single sample.

**6. Crosstab of labels and species**

Now use the pandas crosstab function to build the cross tabulation, passing the two columns of the DataFrame. Cross tabulations like these provide great insights into which sort of samples are in which cluster. But in most datasets, the samples are not labelled by species. How can the quality of a clustering be evaluated in these cases?

**7. Measuring clustering quality**

We need a way to measure the quality of a clustering that uses only the clusters and the samples themselves. A good clustering has tight clusters, meaning that the samples in each cluster are bunched together, not spread out.

**8. Inertia measures clustering quality**

How spread out the samples within each cluster are can be measured by the "inertia". Intuitively, inertia measures how far samples are from their centroids. You can find the precise definition in the scikit-learn documentation. We want clusters that are not spread out, so lower values of the inertia are better. The inertia of a kmeans model is measured automatically when any of the fit methods are called, and is available afterwards as the inertia attribute. In fact, kmeans aims to place the clusters in a way that minimizes the inertia.

**9. The number of clusters**

Here is a plot of the inertia values of clusterings of the iris dataset with different numbers of clusters. Our kmeans model with 3 clusters has relatively low inertia, which is great. But notice that the inertia continues to decrease slowly. So what's the best number of clusters to choose?

**10. How many clusters to choose?**

Ultimately, this is a trade-off. A good clustering has tight clusters (meaning low inertia). But it also doesn't have too many clusters. A good rule of thumb is to choose an elbow in the inertia plot, that is, a point where the inertia begins to decrease more slowly. For example, by this criterion, 3 is a good number of clusters for the iris dataset.

**Transforming features for better clusterings**

Let's look now at another dataset,

**2. Piedmont wines dataset**

00:04 - 00:23

the Piedmont wines dataset. We have 178 samples of red wine from the Piedmont region of Italy. The features measure chemical composition (like alcohol content) and visual properties like color intensity. The samples come from 3 distinct varieties of wine.

1. 1 Source: https://archive.ics.uci.edu/ml/datasets/Wine

**3. Clustering the wines**

00:23 - 00:30

Let's take the array of samples and use KMeans to find 3 clusters.

**4. Clusters vs. varieties**

00:30 - 00:46

There are three varieties of wine, so let's use pandas crosstab to check the cluster label - wine variety correspondence. As you can see, this time things haven't worked out so well. The KMeans clusters don't correspond well with the wine varieties.

**5. Feature variances**

00:46 - 01:00

The problem is that the features of the wine dataset have very different variances. The variance of a feature measures the spread of its values. For example, the malic acid feature has a higher variance

**6. Feature variances**

01:00 - 01:12

than the od280 feature, and this can also be seen in their scatter plot. The differences in some of the feature variances is enormous, as seen here, for example, in the scatter plot of the od280 and proline features.

**7. StandardScaler**

01:12 - 01:46

In KMeans clustering, the variance of a feature corresponds to its influence on the clustering algorithm. To give every feature a chance, the data needs to be transformed so that features have equal variance. This can be achieved with the StandardScaler from scikit-learn. It transforms every feature to have mean 0 and variance 1. The resulting "standardized" features can be very informative. Using standardized od280 and proline, for example, the three wine varieties are much more distinct.

**8. sklearn StandardScaler**

01:46 - 02:07

Let's see the StandardScaler in action. First, import StandardScaler from sklearn.preprocessing. Then create a StandardScaler object, and fit it to the samples. The transform method can now be used to standardize any samples, either the same ones, or completely new ones.

**9. Similar methods**

02:07 - 02:26

The APIs of StandardScaler and KMeans are similar, but there is an important difference. StandardScaler transforms data, and so has a transform method. KMeans, in contrast, assigns cluster labels to samples, and this done using the predict method.

**10. StandardScaler, then KMeans**

02:26 - 02:54

Let's return to the problem of clustering the wines. We need to perform two steps. Firstly, to standardize the data using StandardScaler, and secondly to take the standardized data and cluster it using KMeans. This can be conveniently achieved by combining the two steps using a scikit-learn pipeline. Data then flows from one step into the next, automatically.

**11. Pipelines combine multiple steps**

02:54 - 03:25

The first steps are the same: creating a StandardScaler and a KMeans object. After that, import the make\_pipeline function from sklearn.pipeline. Apply the make\_pipeline function to the steps that you want to compose in this case, the scaler and the kmeans objects. Now use the fit method of the pipeline to fit both the scaler and kmeans, and use its predict method to obtain the cluster labels.

**12. Feature standardization improves clustering**

03:25 - 03:45

Checking the correspondence between the cluster labels and the wine varieties reveals that this new clustering, incorporating standardization, is fantastic. Its three clusters correspond almost exactly to the three wine varieties. This is a huge improvement on the clustering without standardization.

**13. sklearn preprocessing steps**

03:45 - 03:58

StandardScaler is an example of a "preprocessing" step. There are several of these available in scikit-learn, for example MaxAbsScaler and Normalizer.