Unsupervised Learning in Python

What is unsupervised machine learning?

Black Raven (James Ng)

11 Nov 2020 · 33 min read

This is a memo to share what I have learnt in Unsupervised Learning (in Python), capturing the learning objectives as well as my personal notes. The course is taught by Benjamin Wilson from DataCamp, and it includes 4 chapters:

Chapter 1. Clustering for dataset exploration

Chapter 2. Visualization with hierarchical clustering and t-SNE

Chapter 3. Decorrelating your data and dimension reduction

Chapter 4. Discovering interpretable features

Say you have a collection of customers with a variety of characteristics such as age, location, and financial history, and you wish to discover patterns and sort them into clusters. Or perhaps you have a set of texts, such as wikipedia pages, and you wish to segment them into categories based on their content. This is the world of unsupervised learning, called as such because you are not guiding, or supervising, the pattern discovery by some prediction task, but instead uncovering hidden structure from unlabeled data.

Photo by Element5 Digital on Unsplash

Unsupervised learning encompasses a variety of techniques in machine learning, from clustering to dimension reduction to matrix factorization. In this course, you'll learn the fundamentals of unsupervised learning and implement the essential algorithms using scikit-learn and scipy. You will learn how to cluster, transform, visualize, and extract insights from unlabeled datasets, and end the course by building a recommender system to recommend popular musical artists.

Chapter 1. Clustering for dataset exploration

Learn how to discover the underlying groups (or "clusters") in a dataset. By the end of this chapter, you'll be clustering companies using their stock market prices, and distinguishing different species by clustering their measurements.

Unsupervised Learning

= a class of machine learning techniques for discovering patterns in data, eg. **clustering** customers by their purchase histories, compressing the data using purchase patterns (**dimension reduction**).

= learning without labels, pattern discovery unguided by prediction task

Iris dataset: columns = features, rows = samples, dimension = number of features

k-means clustering: to find clusters of samples, by specifying number of clusters, implemented in sklearn

```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3)
model.fit(samples)
```

```
KMeans(algorithm='auto', ...)
```

```
labels = model.predict(samples)
print(labels)
```

```
[0 0 1 1 0 1 2 1 0 1 ...]
```

```
print(new_samples)
```

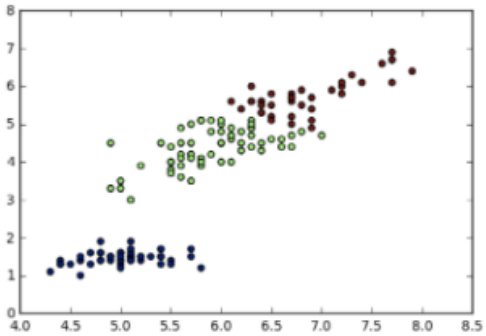
```
[[ 5.7  4.4  1.5  0.4]
 [ 6.5  3.   5.5  1.8]
 [ 5.8  2.7  5.1  1.9]]
```

```
new_labels = model.predict(new_samples)
print(new_labels)
```

```
[0 2 1]
```

```
import matplotlib.pyplot as plt
xs = samples[:,0]
ys = samples[:,2]
plt.scatter(xs, ys, c=labels)
plt.show()
```

- Scatter plot of sepal length vs. petal length
- Each point represents an iris sample
- Color points by cluster labels



How many clusters?

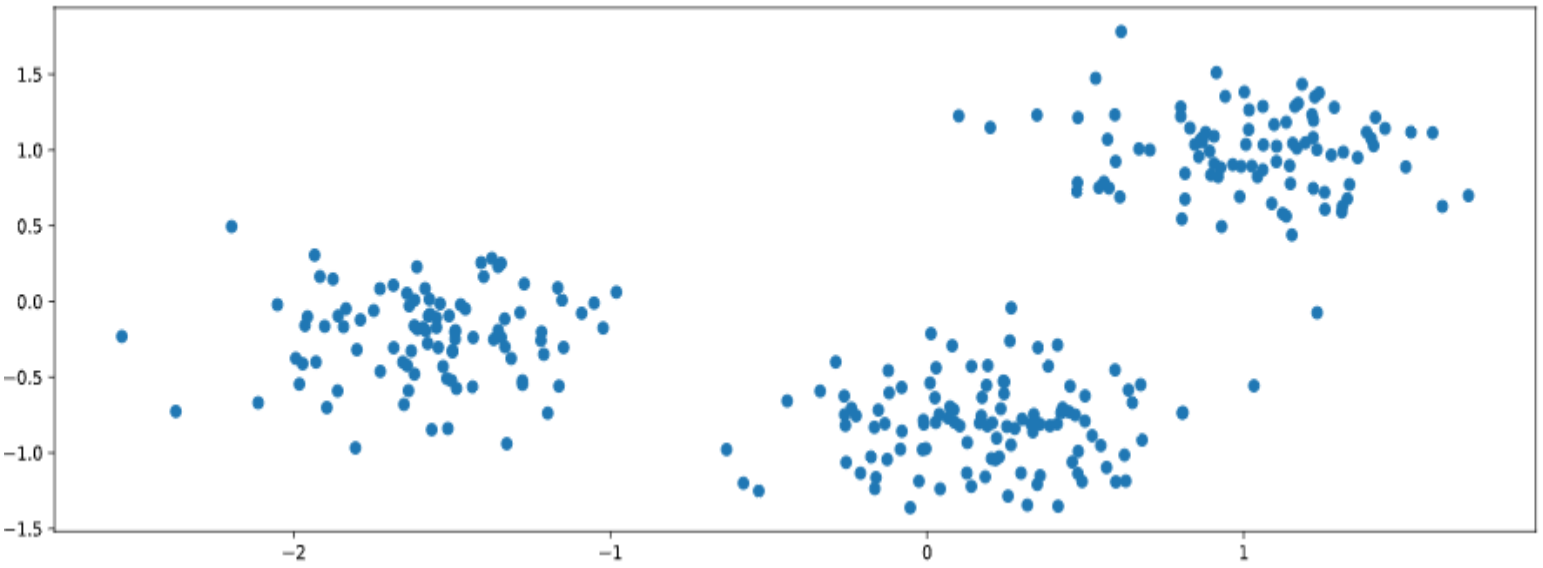
You are given an array `points` of size 300x2, where each row gives the (x, y) co-ordinates of a point on a map. Make a scatter plot of these points, and use the scatter plot to guess how many clusters there are.

`matplotlib.pyplot` has already been imported as `plt`. In the IPython Shell:

- Create an array called `xs` that contains the values of `points[:,0]` - that is, column `0` of `points`.
- Create an array called `ys` that contains the values of `points[:,1]` - that is, column `1` of `points`.
- Make a scatter plot by passing `xs` and `ys` to the `plt.scatter()` function.
- Call the `plt.show()` function to show your plot.

How many clusters do you see?

```
In [1]: xs = points[:,0]
        ys = points[:,1]
        plt.scatter(xs, ys)
        plt.show()
```



Possible Answers

- 2
- 3
- 300

Answer: The scatter plot suggests that there are 3 distinct clusters.

Clustering 2D points

From the scatter plot of the previous exercise, you saw that the points seem to separate into 3 clusters. You'll now create a KMeans model to find 3 clusters, and fit it to the data points from the previous exercise. After the model has been fit, you'll obtain the cluster labels for some new points using the `.predict()` method.

You are given the array `points` from the previous exercise, and also an array `new_points`.

```
# Import KMeans
from sklearn.cluster import KMeans

# Create a KMeans instance with 3 clusters: model
model = KMeans(n_clusters=3)

# Fit model to points
model.fit(points)

# Determine the cluster labels of new_points: labels
labels = model.predict(new_points)

# Print cluster labels of new_points
print(labels)
```

```
<script.py> output:
    [1 2 0 1 2 1 2 2 2 0 1 2 2 0 0 2 0 0 2 2 0 2 1 2 1 0 2 0 0 1 1 2 2 2 0 1 2
     2 1 2 0 1 1 0 1 2 0 0 2 2 2 2 0 0 1 1 0 0 0 1 1 2 2 2 1 2 0 2 1 0 1 1 1 2
     1 0 0 1 2 0 1 0 1 2 0 2 0 1 2 2 2 1 2 2 1 0 0 0 0 1 2 1 0 0 1 1 2 1 0 0 1
     0 0 0 2 2 2 2 0 0 2 1 2 0 2 1 0 2 0 0 2 0 2 0 1 2 1 1 2 0 1 2 1 1 0 2 2 1
     0 1 0 2 1 0 0 1 0 2 2 0 2 0 0 2 2 1 2 2 0 1 0 1 1 2 1 2 2 1 1 0 1 1 1 0 2
     2 1 0 1 0 0 2 2 2 1 2 2 2 0 0 1 2 1 1 1 0 2 2 2 2 2 2 0 0 2 0 0 0 0 2 0 0
     2 2 1 0 1 1 0 1 0 1 0 2 2 0 2 2 2 0 1 1 0 2 2 0 2 0 0 2 0 0 1 0 1 1 1 2 0
     0 0 1 2 1 0 1 0 0 2 1 1 1 0 2 2 2 1 2 0 0 2 1 1 0 1 1 0 1 2 1 0 0 0 0 2 0
     0 2 2 1]
```

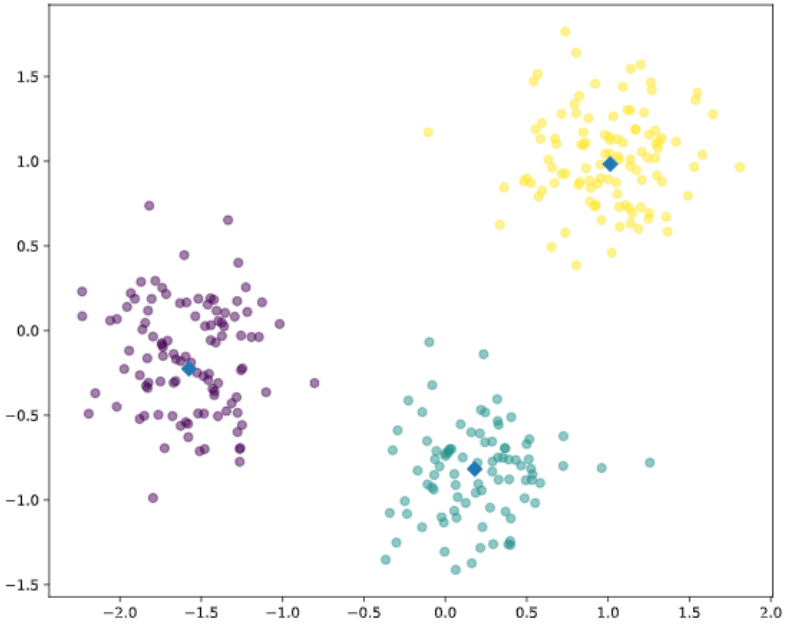
You've successfully performed k-Means clustering and predicted the labels of new points. But it is not easy to inspect the clustering by just looking at the printed labels. A visualization would be far more useful. In the next exercise, you'll inspect your clustering with a scatter plot!

Inspect your clustering

Let's now inspect the clustering you performed in the previous exercise!

A solution to the previous exercise has already run, so `new_points` is an array of points and `labels` is the array of their cluster labels.

```
# Import pyplot
from matplotlib import pyplot as plt

# Assign the columns of new_points: xs and ys
xs = new_points[:,0]
ys = new_points[:,1]

# Make a scatter plot of xs and ys, using labels to define the colors
plt.scatter(xs, ys, c=labels, alpha=0.5)

# Assign the cluster centers: centroids
centroids = model.cluster_centers_

# Assign the columns of centroids: centroids_x, centroids_y
centroids_x = centroids[:,0]
centroids_y = centroids[:,1]

# Make a scatter plot of centroids_x and centroids_y
plt.scatter(centroids_x, centroids_y, marker='D', s=50)
plt.show()
```



The clustering looks great! But how can you be sure that 3 clusters is the correct choice? In other words, how can you evaluate the quality of a clustering?

Evaluating a clustering

Measure of quality can be used to make an informed choice about the number of clusters to look for.

Cross tabulation with pandas: eg. clusters vs species (Iris dataset)

```
import pandas as pd
df = pd.DataFrame({'labels': labels, 'species': species})
print(df)
```

```
    labels     species
0        1      setosa
1        1      setosa
2        2  versicolor
3        2   virginica
4        1      setosa
...
```

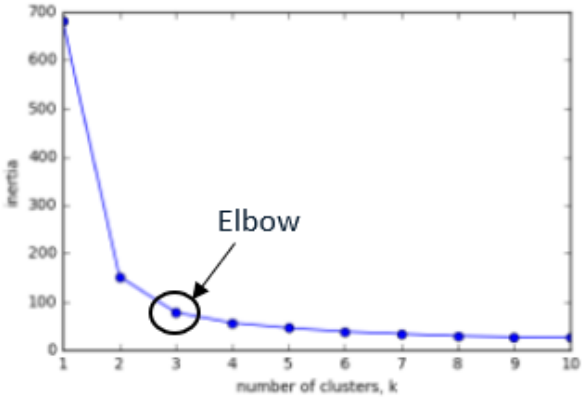
```
ct = pd.crosstab(df['labels'], df['species'])
print(ct)
```

```
species  setosa  versicolor  virginica
labels
0             0           2         36
1            50           0          0
2             0          48         14
```

Measuring clustering quality using only the clusters and the samples themselves. A good clustering has tight clusters, meaning the samples in eacch cluster are bunched together, not spread out. Use “inertia” to measure how spread out the samples are within each cluster, distance from each sample to centroid of its cluster, lower is better.

```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=3)
model.fit(samples)
print(model.inertia_)
```

```
78.9408414261
```



A good clustering has tight clusters (low interia) and not too many clusters.
Choose an “elbow” in the inertia plot, where the inertia begins to decrease more slowly.

How many clusters of grain?

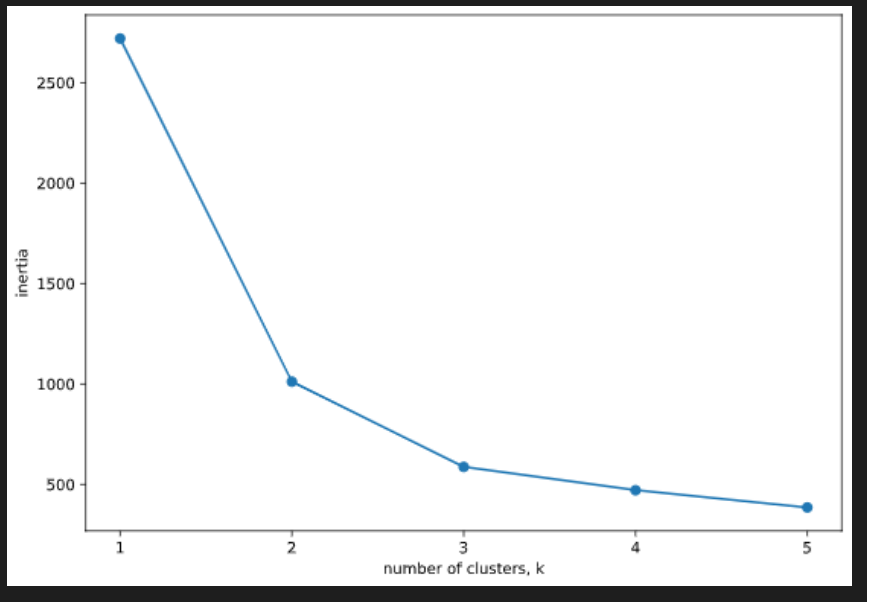
You have learnt how to choose a good number of clusters for a dataset using the k-means inertia graph. You are given an array `samples` containing the measurements (such as area, perimeter, length, and several others) of samples of grain. What's a good number of clusters in this case?

`KMeans` and PyPlot (`plt`) have already been imported for you.

This dataset was sourced from the **UCI Machine Learning Repository**.

```
ks = range(1, 6)
inertias = []

for k in ks:
```

```
    # Create a KMeans instance with k clusters: model
    model = KMeans(n_clusters=k)

    # Fit model to samples
    model.fit(samples)

    # Append the inertia to the list of inertias
    inertias.append(model.inertia_)

# Plot ks vs inertias
plt.plot(ks, inertias, '-o')
plt.xlabel('number of clusters, k')
plt.ylabel('inertia')
plt.xticks(ks)
plt.show()
```



The inertia decreases very slowly from 3 clusters to 4. So 3 clusters would be a good choice for this data.

Evaluating the grain clustering

In the previous exercise, you observed from the inertia plot that 3 is a good number of clusters for the grain data. In fact, the grain samples come from a mix of 3 different grain varieties: "Kama", "Rosa" and "Canadian". In this exercise, cluster the grain samples into three clusters, and compare the clusters to the grain varieties using a cross-tabulation.

You have the array `samples` of grain samples, and a list `varieties` giving the grain variety for each sample. Pandas (`pd`) and `KMeans` have already been imported for you.

```
# Create a KMeans model with 3 clusters: model
model = KMeans(n_clusters=3)

# Use fit_predict to fit model and obtain cluster labels: labels
labels = model.fit_predict(samples)

# Create a DataFrame with clusters and varieties as columns: df
```

```
df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create crosstab: ct
ct = pd.crosstab(df['labels'], df['varieties'])

# Display ct
print(ct)
```

```
<script.py> output:
    varieties  Canadian wheat  Kama wheat  Rosa wheat
    labels
    0                       0           1          60
    1                      68           9           0
    2                       2          60          10
```

The cross-tabulation shows that the 3 varieties of grain separate really well into 3 clusters. But depending on the type of data you are working with, the clustering may not always be this good. Is there anything you can do in such situations to improve your clustering?

Transforming features for better clusterings

Piedmont wines dataset: 178 samles, various chemical composition and visual properties
3 distinct varieties of red wine: Barolo, Gringnolino, Barbera

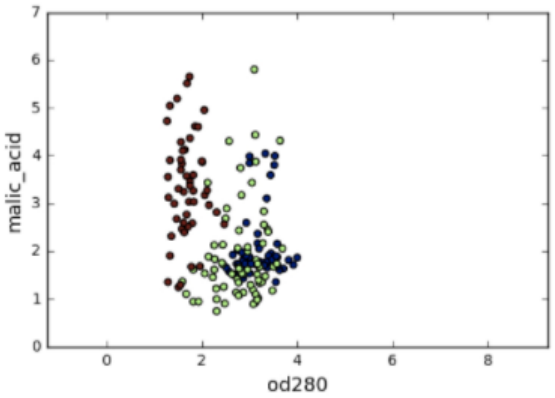
```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3)
labels = model.fit_predict(samples)
```

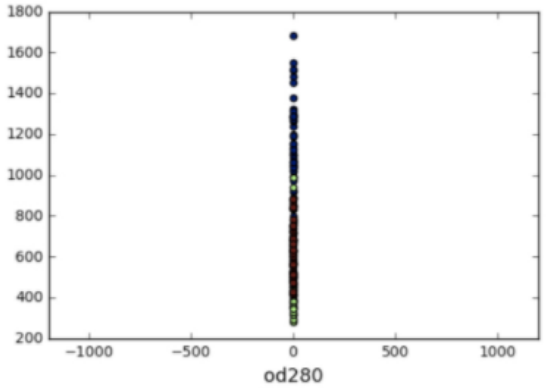
```
df = pd.DataFrame({'labels': labels,
                   'varieties': varieties})
ct = pd.crosstab(df['labels'], df['varieties'])
print(ct)
```

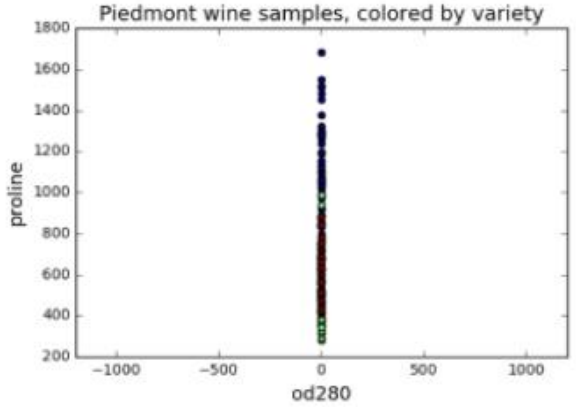
varieties	Barbera	Barolo	Grignolino
labels			
0	29	13	20
1	0	46	1
2	19	0	50

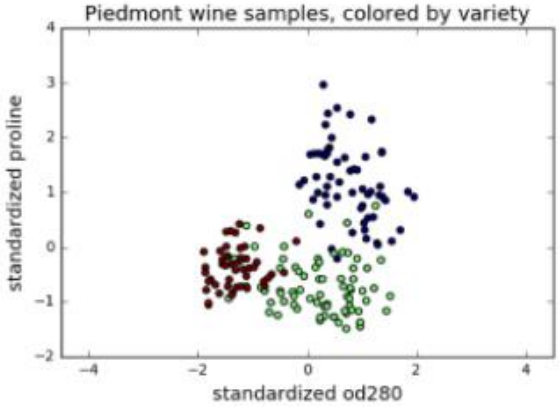
- The wine features have very different variances!
- Variance of a feature measures spread of its values

feature	variance
alcohol	0.65
malic_acid	1.24
...	
od280	0.50
proline	99166.71









In kmeans: feature variance = feature influence

StandardScaler (pre-processing) transforms each feature to have mean 0 and variance 1

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(samples)
StandardScaler(copy=True, with_mean=True, with_std=True)
samples_scaled = scaler.transform(samples)
```

Using pipeline to apply StandardScaler then KMeans

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
scaler = StandardScaler()
kmeans = KMeans(n_clusters=3)
from sklearn.pipeline import make_pipeline
pipeline = make_pipeline(scaler, kmeans)
pipeline.fit(samples)
```

```
Pipeline(steps=...)
```

```
labels = pipeline.predict(samples)
```

With feature standardization:

varieties	Barbera	Barolo	Grignolino
labels			
0	0	59	3
1	48	0	3
2	0	0	65

Without feature standardization

varieties	Barbera	Barolo	Grignolino
labels			
0	29	13	20
1	0	46	1
2	19	0	50

Scaling fish data for clustering

You are given an array `samples` giving measurements of fish. Each row represents an individual fish. The measurements, such as weight in grams, length in centimeters, and the percentage ratio of height to length, have very different scales. In order to cluster this data effectively, you'll need to standardize these features first. In this exercise, you'll build a pipeline to standardize and cluster the data.

These fish measurement data were sourced from the **Journal of Statistics Education**.

```
# Perform the necessary imports
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# Create scaler: scaler
scaler = StandardScaler()

# Create KMeans instance: kmeans
kmeans = KMeans(n_clusters=4)

# Create pipeline: pipeline
pipeline = make_pipeline(scaler, kmeans)
```

Now that you've built the pipeline, you'll use it in the next exercise to cluster the fish by their measurements.

Clustering the fish data

You'll now use your standardization and clustering pipeline from the previous exercise to cluster the fish by their measurements, and then create a cross-tabulation to compare the cluster labels with the fish species.

As before, `samples` is the 2D array of fish measurements. Your pipeline is available as `pipeline`, and the species of every fish sample is given by the list `species`.

```
# Import pandas
import pandas as pd
```

```python
# Fit the pipeline to samples
pipeline.fit(samples)

# Calculate the cluster labels: labels
labels = pipeline.predict(samples)

# Create a DataFrame with labels and species as columns: df
df = pd.DataFrame({'labels': labels, 'species': species})

# Create crosstab: ct
ct = pd.crosstab(df['labels'], df['species'])

# Display ct
print(ct)
```

```
<script.py> output:
    species  Bream  Pike  Roach  Smelt
    labels
    0            0     0      0     13
    1           33     0      1      0
    2            0    17      0      0
    3            1     0     19      1
```

It looks like the fish data separates really well into 4 clusters!

Clustering stocks using KMeans

In this exercise, you'll cluster companies using their daily stock price movements (i.e. the dollar difference between the closing and opening prices for each trading day). You are given a NumPy array `movements` of daily price movements from 2010 to 2015 (obtained from Yahoo! Finance), where each row corresponds to a company, and each column corresponds to a trading day.

Some stocks are more expensive than others. To account for this, include a `Normalizer` at the beginning of your pipeline. The Normalizer will separately transform each company's stock price to a relative scale before the clustering begins.

Note that `Normalizer()` is different to `StandardScaler()`, which you used in the previous exercise.
While `StandardScaler()` standardizes **features** (such as the features of the fish data from the previous exercise) by removing the mean and scaling to unit variance, `Normalizer()` rescales **each sample** - here, each company's stock price - independently of the other.

`KMeans` and `make_pipeline` have already been imported for you.

```
# Import Normalizer
from sklearn.preprocessing import Normalizer

# Create a normalizer: normalizer
normalizer = Normalizer()

# Create a KMeans model with 10 clusters: kmeans
kmeans = KMeans(n_clusters=10)

# Make a pipeline chaining normalizer and kmeans: pipeline
pipeline = make_pipeline(normalizer, kmeans)

# Fit pipeline to the daily price movements
pipeline.fit(movements)
```

Now that your pipeline has been set up, you can find out which stocks move together in the next exercise!

Which stocks move together?

In the previous exercise, you clustered companies by their daily stock price movements. So which company have stock prices that tend to change in the same way? You'll now inspect the cluster labels from your clustering to find out.

Your solution to the previous exercise has already been run. Recall that you constructed a Pipeline `pipeline` containing a `KMeans` model and fit it to the NumPy array `movements` of daily stock movements. In addition, a list `companies` of the company names is available.

```
# Import pandas
import pandas as pd

# Predict the cluster labels: labels
labels = pipeline.predict(movements)

# Create a DataFrame aligning labels and companies: df
```

```python
df = pd.DataFrame({'labels': labels, 'companies': companies})

# Display df sorted by cluster label
print(df.sort_values('labels'))
```

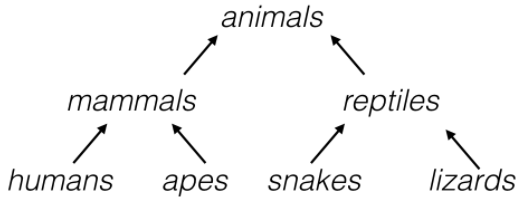
```
<script.py> output:
    labels                          companies
59       0                              Yahoo
15       0                               Ford
35       0                            Navistar
26       1                      JPMorgan Chase
16       1                   General Electrics
58       1                               Xerox
11       1                               Cisco
18       1                        Goldman Sachs
20       1                           Home Depot
5        1                      Bank of America
3        1                     American express
55       1                          Wells Fargo
1        1                                 AIG
38       2                               Pepsi
40       2                        Procter Gamble
28       2                             Coca Cola
27       2                         Kimberly-Clark
9        2                     Colgate-Palmolive
54       3                             Walgreen
36       3                      Northrop Grumman
29       3                       Lookheed Martin
4        3                                Boeing
0        4                                 Apple
47       4                               Symantec
33       4                             Microsoft
32       4                                    3M
31       4                             McDonalds
30       4                            MasterCard
50       4  Taiwan Semiconductor Manufacturing
14       4                                  Dell
17       4                        Google/Alphabet
24       4                                 Intel
23       4                                   IBM
2        4                                Amazon
51       4                   Texas instruments
43       4                                   SAP
45       5                                 Sony
48       5                                Toyota
21       5                                 Honda
22       5                                    HP
34       5                             Mitsubishi
7        5                                 Canon
56       6                             Wal-Mart
57       7                                 Exxon
44       7                           Schlumberger
8        7                            Caterpillar
10       7                          ConocoPhillips
12       7                                Chevron
13       7                      DuPont de Nemours
53       7                          Valero Energy
39       8                                 Pfizer
41       8                           Philip Morris
25       8                       Johnson & Johnson
49       9                                  Total
46       9                         Sanofi-Aventis
37       9                                Novartis
42       9                       Royal Dutch Shell
19       9                        GlaxoSmithKline
52       9                                Unilever
6        9                  British American Tobacco
```

Chapter 2. Visualization with hierarchical clustering and t-SNE

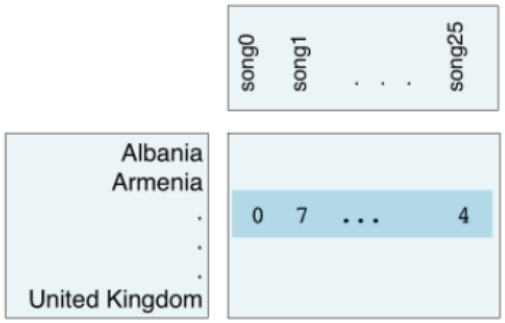
In this chapter, you'll learn about two unsupervised learning techniques for data visualization, hierarchical clustering and t-SNE. Hierarchical clustering merges the data samples into ever-coarser clusters, yielding a tree visualization of the resulting cluster hierarchy. t-SNE maps the data samples into 2d space so that the proximity of the samples to one another can be visualized.

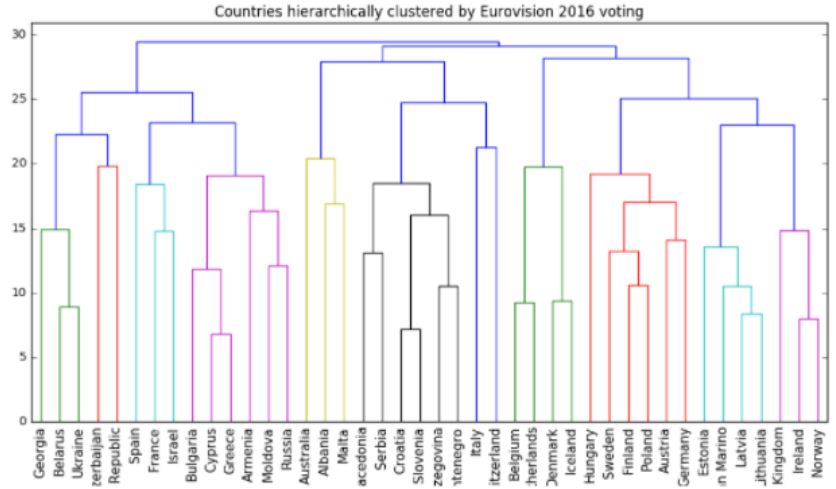
Visualizing hierarchies

A hierarchy of groups, eg. groups of living things can form a hierarchy
Clusters are contained in one another, ie,
hierarchical clustering arranges samples into a hierarchy of clusters



Eurovision scoring dataset →**dendrogram**





```
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram
mergings = linkage(samples, method='complete')
dendrogram(mergings,
           labels=country_names,
           leaf_rotation=90,
           leaf_font_size=6)
plt.show()
```

How many merges?

If there are 5 data samples, how many merge operations will occur in a hierarchical clustering?

Answer: With 5 data samples, there would be 4 merge operations, and with 6 data samples, there would be 5 merges, and so on.

Possible Answers

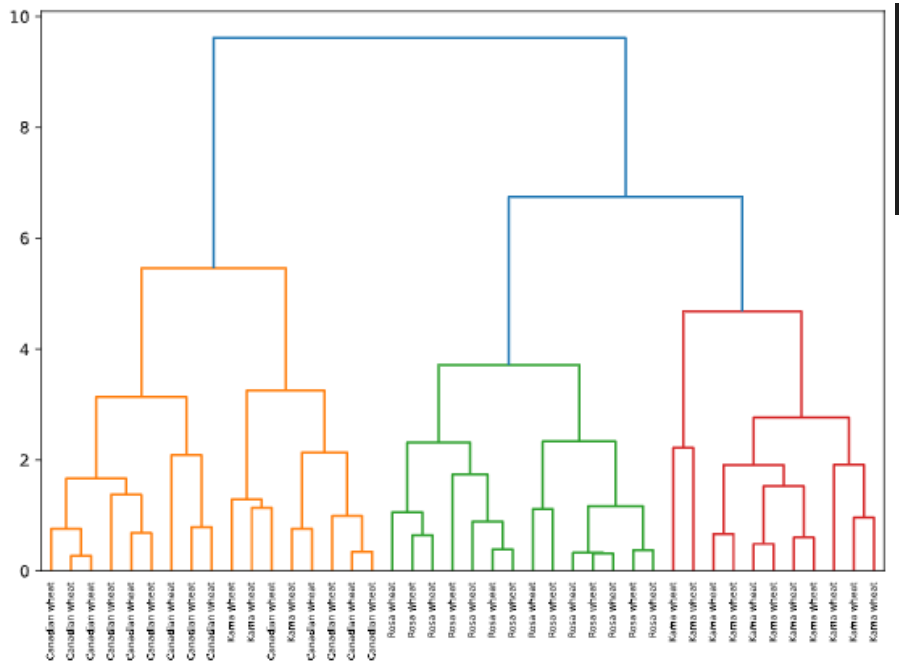
- (●) 4 merges.
- ( ) 3 merges.
- ( ) This can't be known in advance.

Hierarchical clustering of the grain data

You have learnt that the SciPy `linkage()` function performs hierarchical clustering on an array of samples. Use the `linkage()` function to obtain a hierarchical clustering of the grain samples, and use `dendrogram()` to visualize the result. A sample of the grain measurements is provided in the array `samples`, while the variety of each grain sample is given by the list `varieties`.

```
# Perform the necessary imports
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt

# Calculate the linkage: mergings
mergings = linkage(samples, method='complete')

# Plot the dendrogram, using varieties as labels
dendrogram(mergings,
```

```
            labels=varieties,
            leaf_rotation=90,
            leaf_font_size=6,
)
plt.show()
```



Dendrograms are a great way to illustrate the arrangement of the clusters produced by hierarchical clustering.

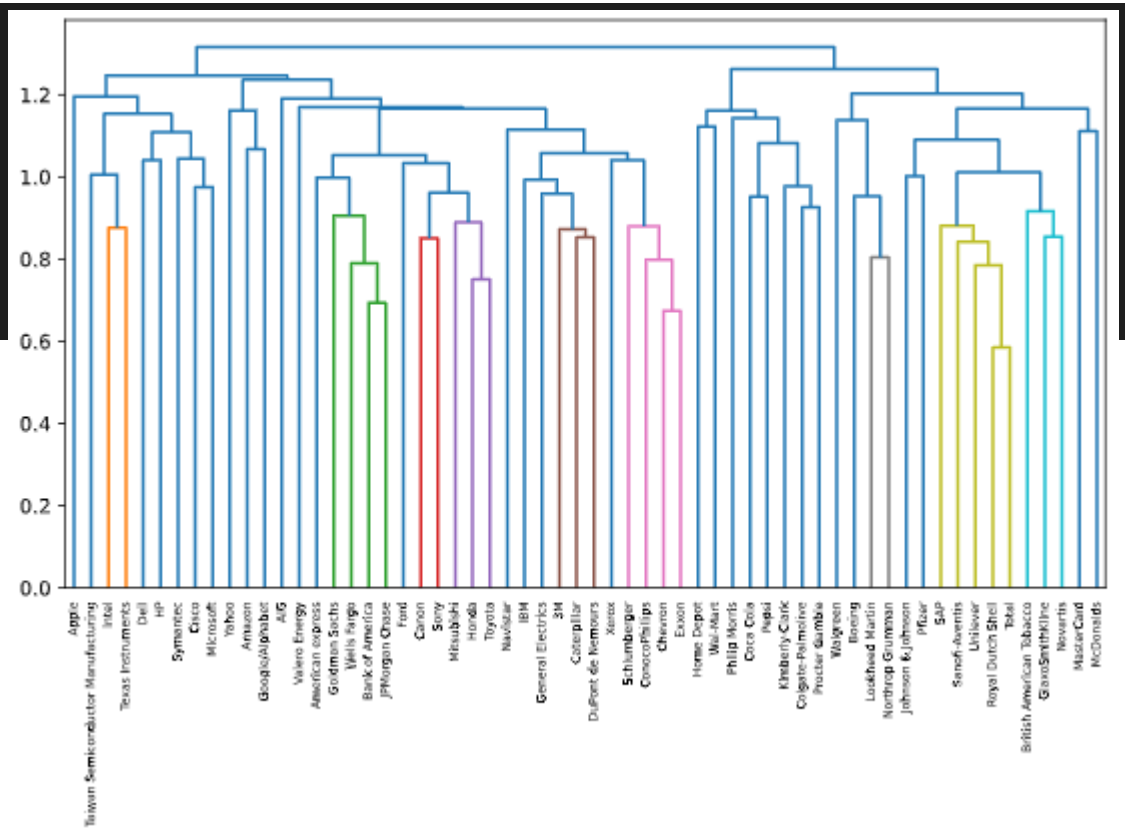
Hierarchies of stocks

In chapter 1, you used k-means clustering to cluster companies according to their stock price movements. Now, you'll perform hierarchical clustering of the companies. You are given a NumPy array of price movements `movements`, where the rows correspond to companies, and a list of the company names `companies`. SciPy hierarchical clustering doesn't fit into a sklearn pipeline, so you'll need to use the `normalize()` function from `sklearn.preprocessing` instead of `Normalizer`.

`linkage` and `dendrogram` have already been imported from `scipy.cluster.hierarchy`, and PyPlot has been imported as `plt`.

```
# Import normalize
from sklearn.preprocessing import normalize

# Normalize the movements: normalized_movements
normalized_movements = normalize(movements)

# Calculate the linkage: mergings
mergings = linkage(normalized_movements, method='complete')
```

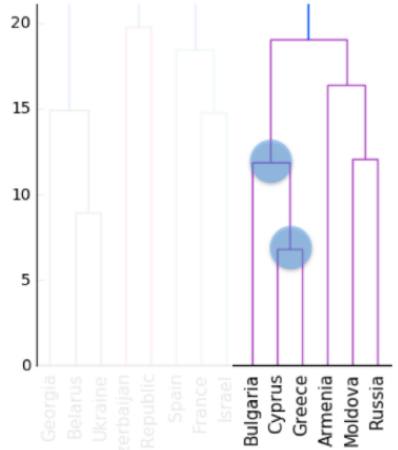
```
# Plot the dendrogram
dendrogram(
    mergings,
    labels=companies,
    leaf_rotation=90,
    leaf_font_size=6
)
plt.show()
```



You can produce great visualizations such as this with hierarchical clustering, but it can be used for more than just visualizations.

Cluster labels in hierarchical clustering

- Height on dendrogram = distance between merging clusters
- E.g. clusters with only Cyprus and Greece had distance approx. 6
- This new cluster distance approx. 12 from cluster with only Bulgaria



Height on dendrogram specifies an intermediate clustering and max distance between merging clusters.
Distance between clusters is defined by a “linkage method” = max distances between their samples.

```
from scipy.cluster.hierarchy import linkage
mergings = linkage(samples, method='complete')
from scipy.cluster.hierarchy import fcluster
labels = fcluster(mergings, 15, criterion='distance')
print(labels)
```

```
[ 9  8 11 20  2  1 17 14 ... ]
```

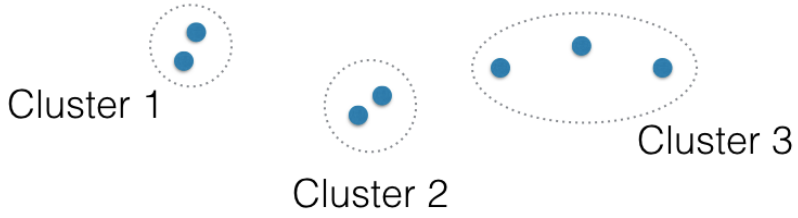
```
import pandas as pd
pairs = pd.DataFrame({'labels': labels, 'countries': country_names})
print(pairs.sort_values('labels'))
```

```
             countries  labels
5              Belarus       1
40             Ukraine       1
...
36               Spain       5
8             Bulgaria       6
19              Greece       6
10              Cyprus       6
28             Moldova       7
...
```

Which clusters are closest?

You have learnt that the linkage method defines how the distance between clusters is measured. In *complete* linkage, the distance between clusters is the distance between the *furthest* points of the clusters. In *single* linkage, the distance between clusters is the distance between the *closest* points of the clusters.

Consider the three clusters in the diagram. Which of the following statements are true?



A. In single linkage, Cluster 3 is the closest to Cluster 2.

B. In complete linkage, Cluster 1 is the closest to Cluster 2.

Possible Answers

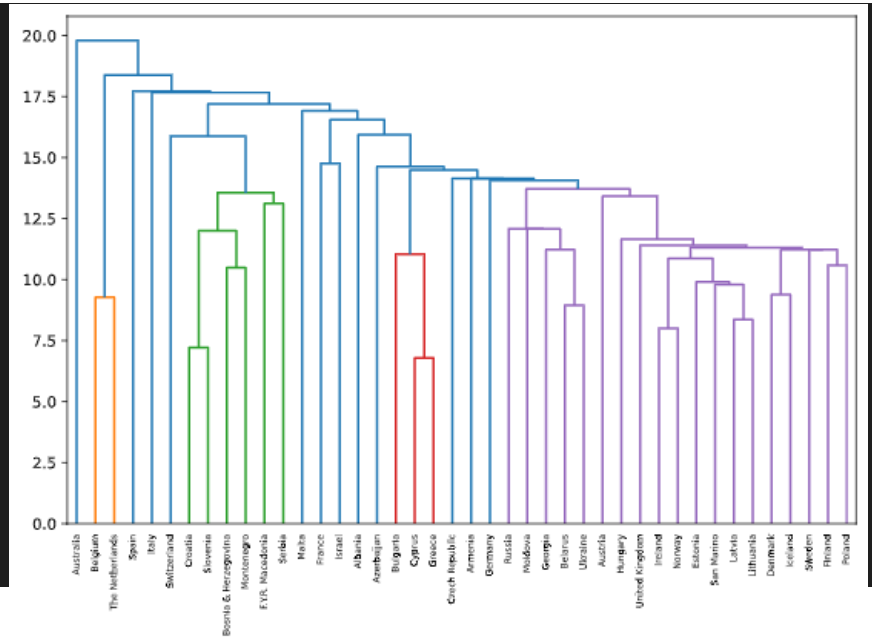
- ○ Neither A nor B.
- ○ A only.
- ◉ Both A and B.

Different linkage, different hierarchical clustering!

You have seen a hierarchical clustering of the voting countries at the Eurovision song contest using `'complete'` linkage. Now, perform a hierarchical clustering of the voting countries with `'single'` linkage, and compare the resulting dendrogram. Different linkage, different hierarchical clustering!

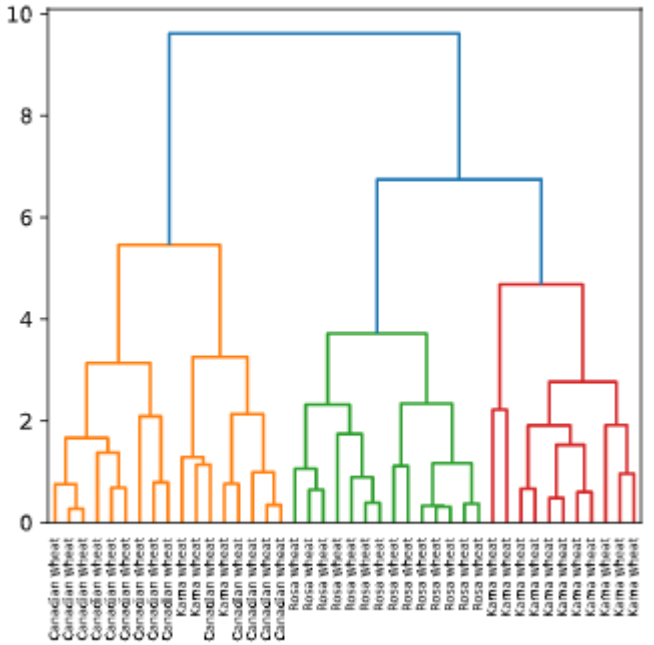
You are given an array `samples`. Each row corresponds to a voting country, and each column corresponds to a performance that was voted for. The list `country_names` gives the name of each voting country. This dataset was obtained from **Eurovision**.

```
# Perform the necessary imports
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram

# Calculate the linkage: mergings
mergings = linkage(samples, method='single')

# Plot the dendrogram
dendrogram(mergings,
           labels=country_names,
           leaf_rotation=90,
           leaf_font_size=6,
)
plt.show()
```



Intermediate clusterings

Displayed on the right is the dendrogram for the hierarchical clustering of the grain samples that you computed earlier. If the hierarchical clustering were stopped at height 6 on the dendrogram, how many clusters would there be?



Possible Answers

- 1.
- 3.
- As many as there were at the beginning.

Extracting the cluster labels

In the previous exercise, you saw that the intermediate clustering of the grain samples at height 6 has 3 clusters. Now, use the `fcluster()` function to extract the cluster labels for this intermediate clustering, and compare the labels with the grain varieties using a cross-tabulation.

The hierarchical clustering has already been performed and `mergings` is the result of the `linkage()` function. The list `varieties` gives the variety of each grain sample.

```
# Perform the necessary imports
import pandas as pd
from scipy.cluster.hierarchy import fcluster

# Use fcluster to extract labels: labels
labels = fcluster(mergings, 6, criterion='distance')

# Create a DataFrame with labels and varieties as columns: df
df = pd.DataFrame({'labels': labels, 'varieties': varieties})
```

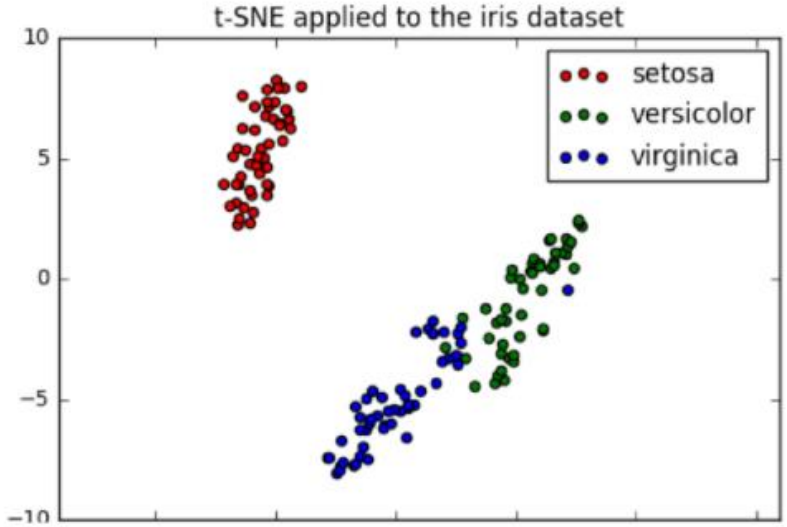
```
# Create crosstab: ct
ct = pd.crosstab(df['labels'], df['varieties'])

# Display ct
print(ct)
```

```
<script.py> output:
    varieties  Canadian wheat  Kama wheat  Rosa wheat
    labels
    1                      14           3           0
    2                       0           0          14
    3                       0          11           0
```

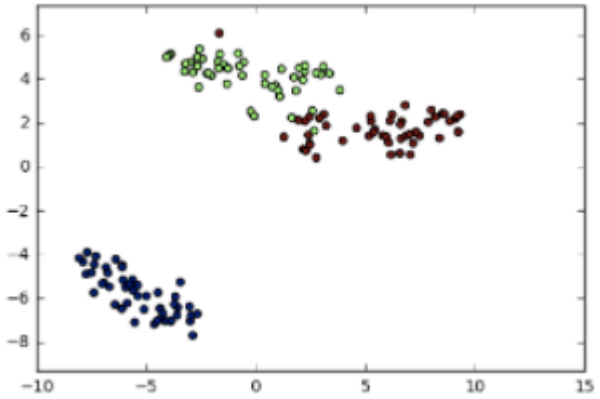
You've now mastered the fundamentals of k-Means and agglomerative hierarchical clustering. Next, you'll learn about t-SNE, which is a powerful tool for visualizing high dimensional data.

t-SNE for 2-dimensional maps

t-SNE = “t-distributed stochastic neighbour embedding”
maps samples from their high-dimensional space into 2D (or 3D) space, so they can be visualised

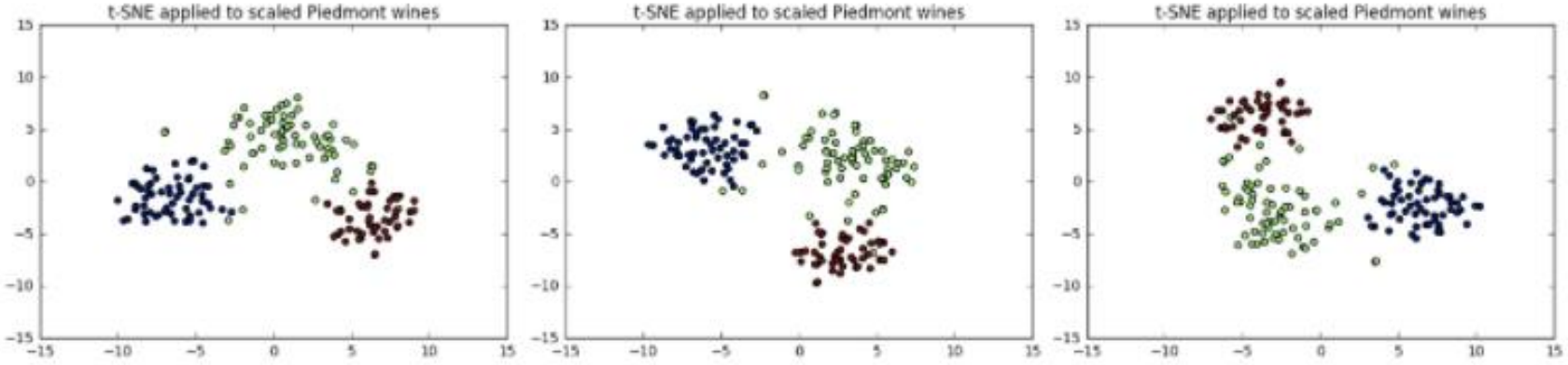


```
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
model = TSNE(learning_rate=100)
transformed = model.fit_transform(samples)
xs = transformed[:,0]
ys = transformed[:,1]
plt.scatter(xs, ys, c=species)
plt.show()
```



Note t-SNE characteristics:

- t-SNE has only fit_transform() method
- t-SNE has learning rate, try values between 50 and 200
- t-SNE plot axes do not have ay interpretable meaning: they are different every time t-SNE is applied, even on the same data using the same codes. Scaled Piedmont wine samples example:

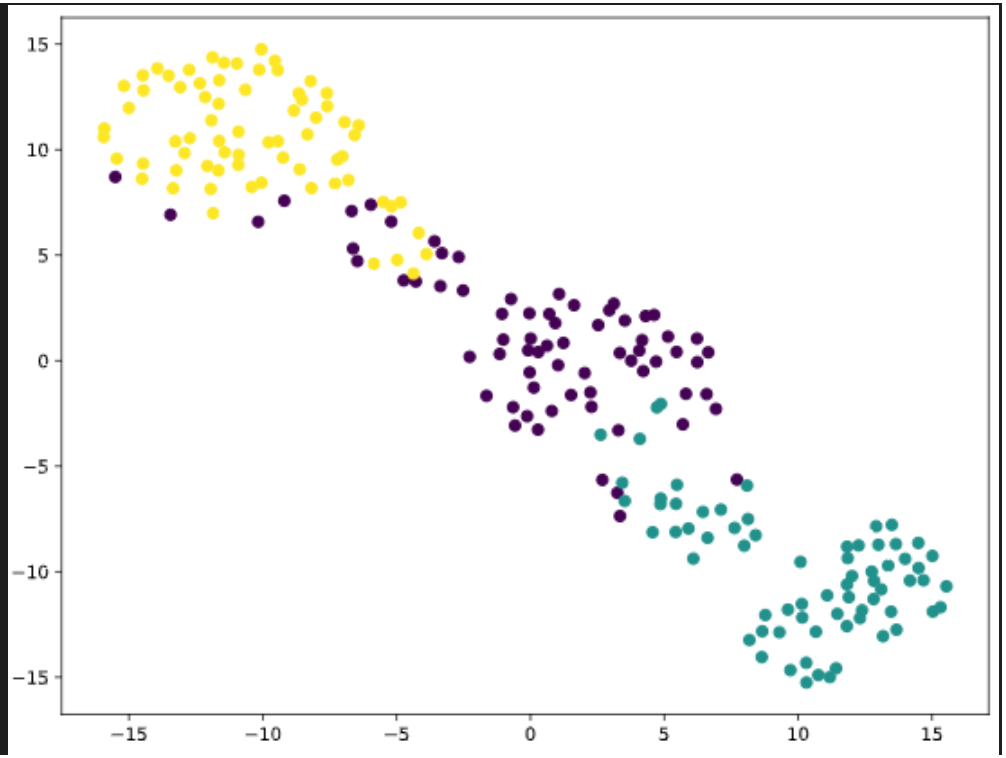


The 3 wine varieties (represented by 3 colors) have the same position relative to one another.

t-SNE visualization of grain dataset

You have seen t-SNE applied to the iris dataset. In this exercise, you'll apply t-SNE to the grain samples data and inspect the resulting t-SNE features using a scatter plot. You are given an array `samples` of grain samples and a list `variety_numbers` giving the variety number of each grain sample.

```
# Import TSNE
from sklearn.manifold import TSNE

# Create a TSNE instance: model
model = TSNE(learning_rate=200)

# Apply fit_transform to samples: tsne_features
tsne_features = model.fit_transform(samples)

# Select the 0th feature: xs
xs = tsne_features[:,0]

# Select the 1st feature: ys
ys = tsne_features[:,1]

# Scatter plot, coloring by variety_numbers
plt.scatter(xs, ys, c=variety_numbers)
plt.show()
```



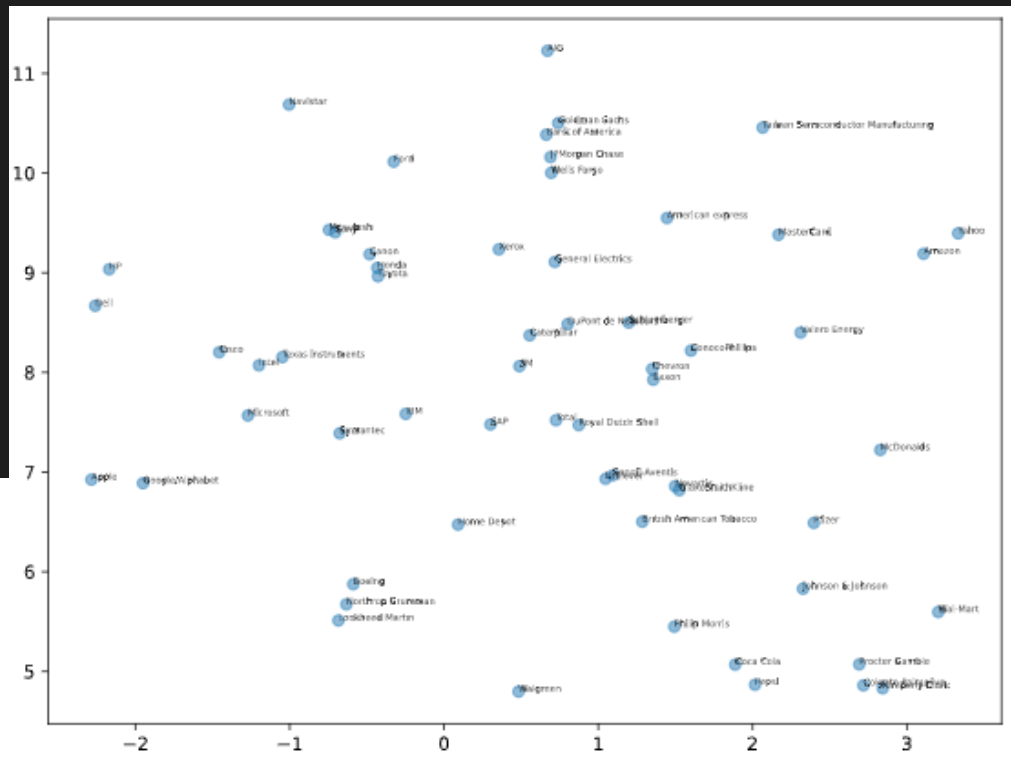
As you can see, the t-SNE visualization manages to separate the 3 varieties of grain samples. But how will it perform on the stock data? You'll find out in the next exercise!

A t-SNE map of the stock market

t-SNE provides great visualizations when the individual samples can be labeled. In this exercise, you'll apply t-SNE to the company stock price data. A scatter plot of the resulting t-SNE features, labeled by the company names, gives you a map of the stock market! The stock price movements for each company are available as the array `normalized_movements` (these have already been normalized for you).

The list `companies` gives the name of each company. PyPlot (`plt`) has been imported for you.

```
# Import TSNE
from sklearn.manifold import TSNE

# Create a TSNE instance: model
model = TSNE(learning_rate=50)

# Apply fit_transform to normalized_movements: tsne_features
tsne_features = model.fit_transform(normalized_movements)

# Select the 0th feature: xs
xs = tsne_features[:,0]

# Select the 1th feature: ys
ys = tsne_features[:,1]

# Scatter plot
plt.scatter(xs, ys, alpha=0.5)

# Annotate the points
for x, y, company in zip(xs, ys, companies):
    plt.annotate(company, (x, y), fontsize=5, alpha=0.75)
plt.show()
```



It's visualizations such as this that make t-SNE such a powerful tool for extracting quick insights from high dimensional data.

Chapter 3. Decorrelating your data and dimension reduction

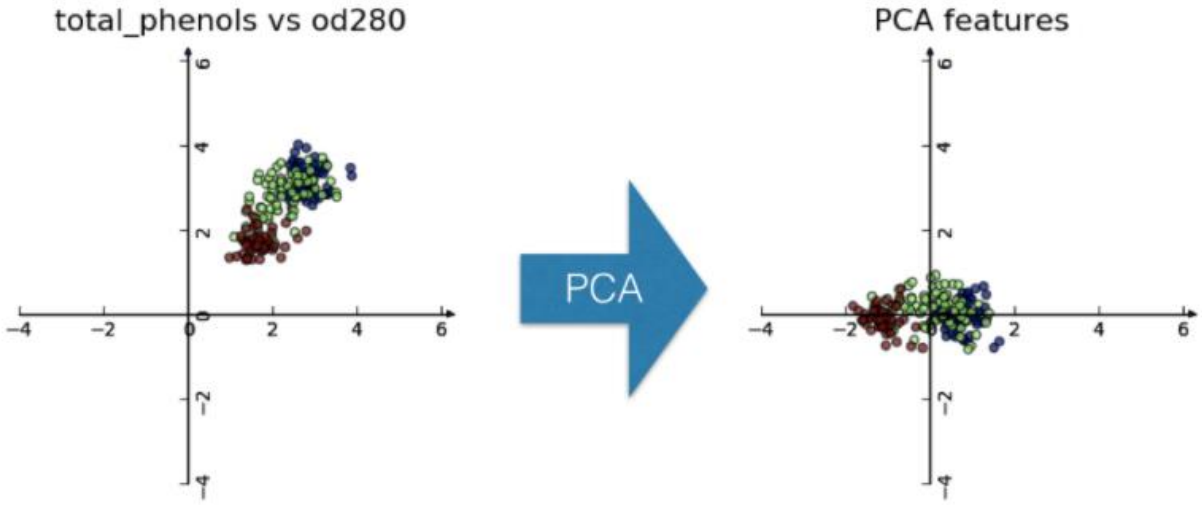
Dimension reduction summarizes a dataset using its common occuring patterns. In this chapter, you'll learn about the most fundamental of dimension reduction techniques, "**Principal Component Analysis**" ("**PCA**"). PCA is often used before supervised learning to improve model performance and generalization. It can also be useful for unsupervised learning. For example, you'll employ a variant of PCA will allow you to cluster Wikipedia articles by their content!

Visualizing the PCA transformation

Dimension reduction finds patterns in data, and uses these patterns to re-express it in a compressed form. It removes less-informative “noise” features, so that prediction (classification or regression) can be done better.

Principal Component Analysis

- Removes correlation
- Rotates the samples so that they are aligned with the coordinate axes
- Shifts data samples so that they have mean 0
- No information is lost



```
from sklearn.decomposition import PCA
model = PCA()
model.fit(samples)
```

```
PCA(copy=True, ...)
```

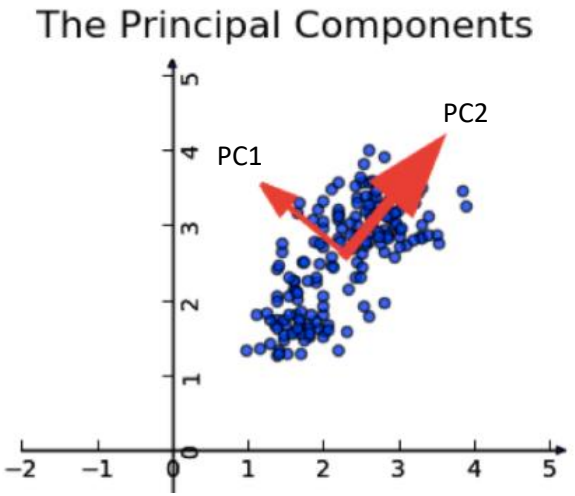
```
transformed = model.transform(samples)
```

```
print(transformed)
```

```
[[  1.32771994e+00   4.51396070e-01]
 [  8.32496068e-01   2.33099664e-01]
 ...
 [ -9.33526935e-01  -4.60559297e-01]]
```

```
print(model.components_)
```

```
[[ 0.64116665  0.76740167]
 [-0.76740167  0.64116665]]
```



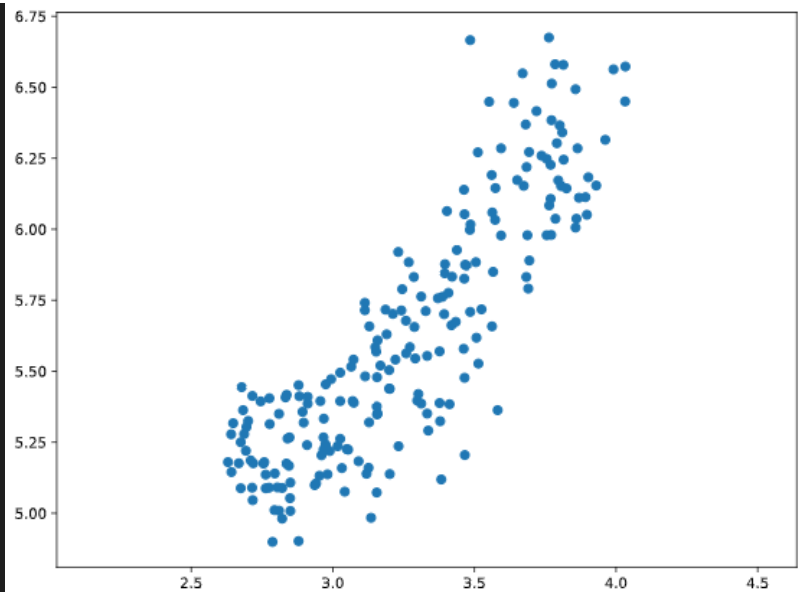
Correlated data in nature

You are given an array `grains` giving the width and length of samples of grain. You suspect that width and length will be correlated. To confirm this, make a scatter plot of width vs length and measure their Pearson correlation.

```
# Perform the necessary imports
import matplotlib.pyplot as plt
from scipy.stats import pearsonr

# Assign the 0th column of grains: width
width = grains[:,0]
```

```
# Assign the 1st column of grains: length
length = grains[:,1]

# Scatter plot width vs length
plt.scatter(width, length)
plt.axis('equal')
plt.show()

# Calculate the Pearson correlation
correlation, pvalue = pearsonr(width, length)

# Display the correlation
print(correlation)
```

```
<script.py> output:
    0.8604149377143466
```



As you would expect, the width and length of the grain samples are highly correlated.

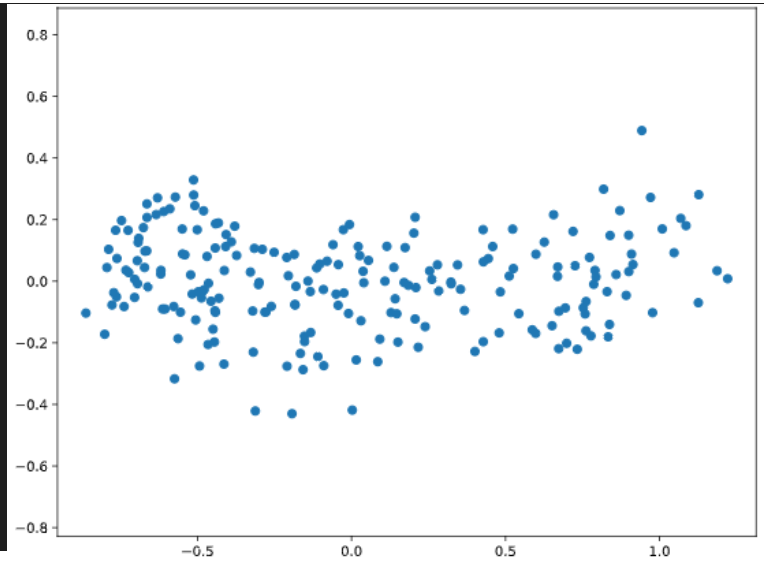
Decorrelating the grain measurements with PCA

You observed in the previous exercise that the width and length measurements of the grain are correlated. Now, you'll use PCA to decorrelate these measurements, then plot the decorrelated points and measure their Pearson correlation.

```
# Import PCA
from sklearn.decomposition import PCA

# Create PCA instance: model
model = PCA()

# Apply the fit_transform method of model to grains: pca_features
pca_features = model.fit_transform(grains)

# Assign 0th column of pca_features: xs
xs = pca_features[:,0]
```

```
# Assign 1st column of pca_features: ys
ys = pca_features[:,1]

# Scatter plot xs vs ys
plt.scatter(xs, ys)
plt.axis('equal')
plt.show()

# Calculate the Pearson correlation of xs and ys
correlation, pvalue = pearsonr(xs, ys)

# Display the correlation
print(correlation)
```

```
<script.py> output:
    2.5478751053409354e-17
```

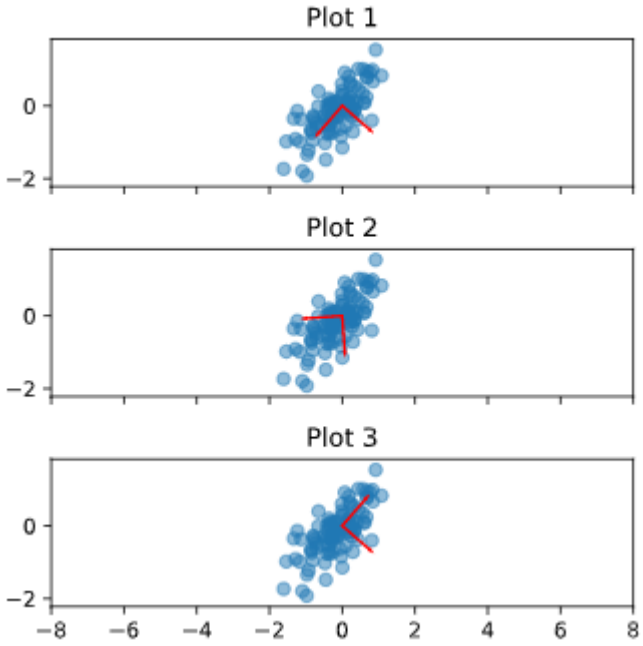


You've successfully decorrelated the grain measurements with PCA!

Principal components

On the right are three scatter plots of the same point cloud. Each scatter plot shows a different set of axes (in red). In which of the plots could the axes represent the principal components of the point cloud?

Recall that the principal components are the directions along which the the data varies.



Possible Answers

- None of them.
- Both plot 1 and plot 3.
- Plot 2.

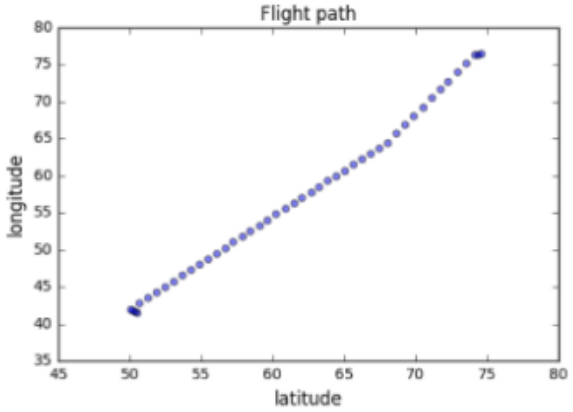
You've correctly inferred that the principal components have to align with the axes of the point cloud. This happens in both plot 1 and plot 3.

Intrinsic dimension

Intrinsic dimension = number of features needed to approximate the dataset, significant PCA features

- 2 features: longitude and latitude at points along a flight path
- Dataset *appears* to be 2-dimensional

latitude	longitude
50.529	41.513
50.360	41.672
50.196	41.835
...	

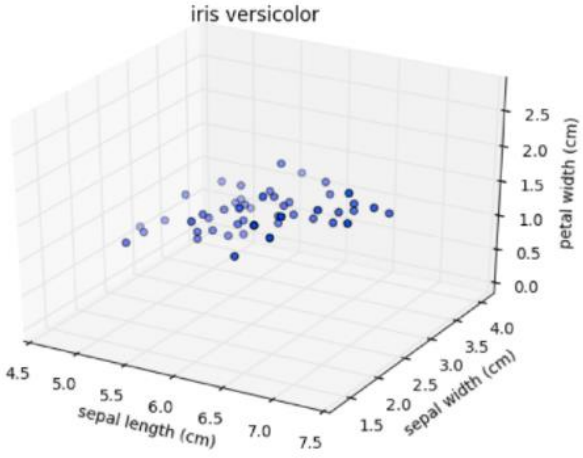


Flight path can be approximated using 1 dimension

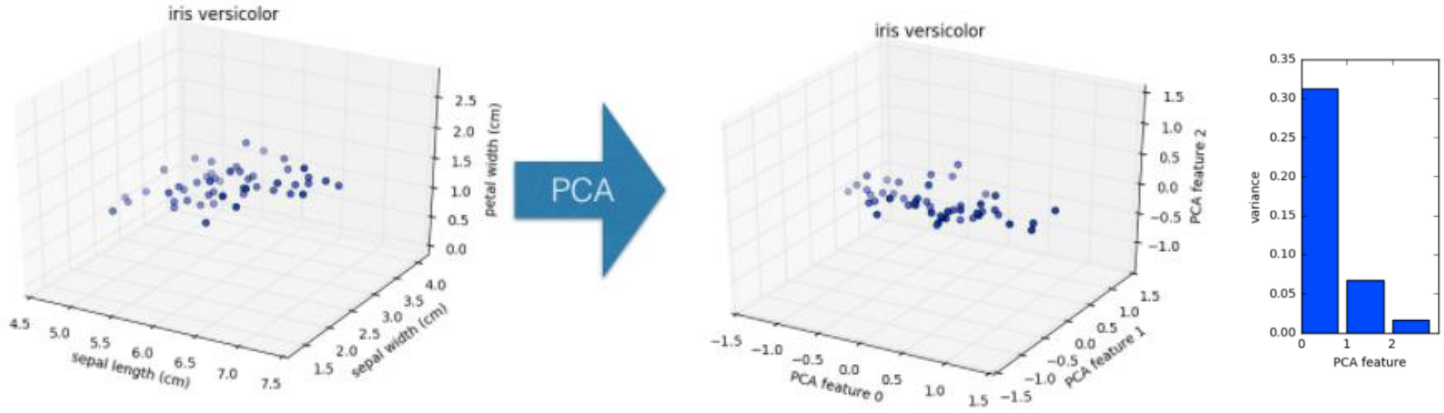
Iris species has 3 features:
sepal length, sepal width, petal width

Versicolor dataset has intrinsic dimension 2

Samples lie close to a flat 2-dimensional sheet



The intrinsic dimension can be identified by counting the number of PCA features with significant variance

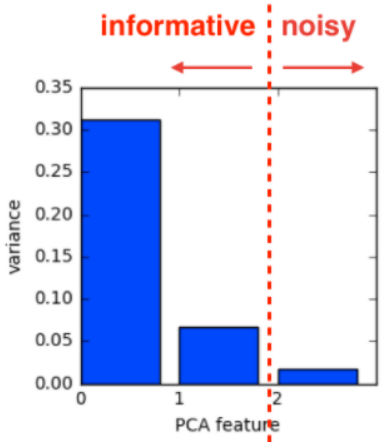


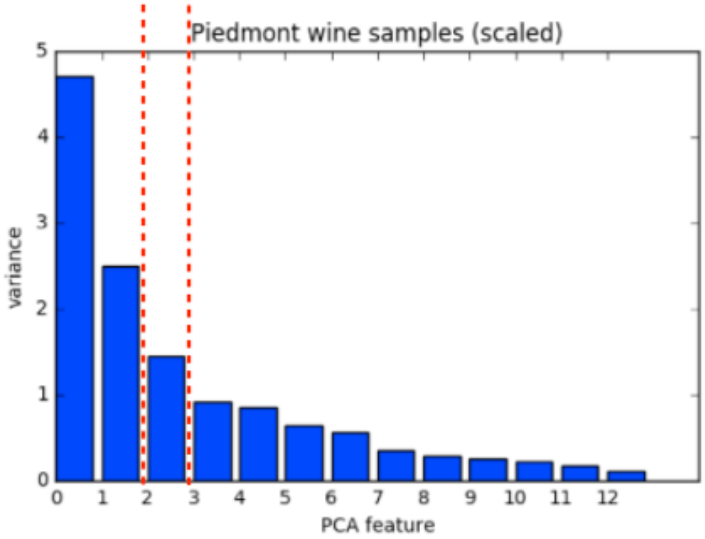
```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
pca = PCA()
pca.fit(samples)
```

```
PCA(copy=True, ... )
```

```
features = range(pca.n_components_)
```

```
plt.bar(features, pca.explained_variance_)
plt.xticks(features)
plt.ylabel('variance')
plt.xlabel('PCA feature')
plt.show()
```





Intrinsic dimension can be ambiguous, example Piedmont wines can have 2 or 3 intrinsic dimensions

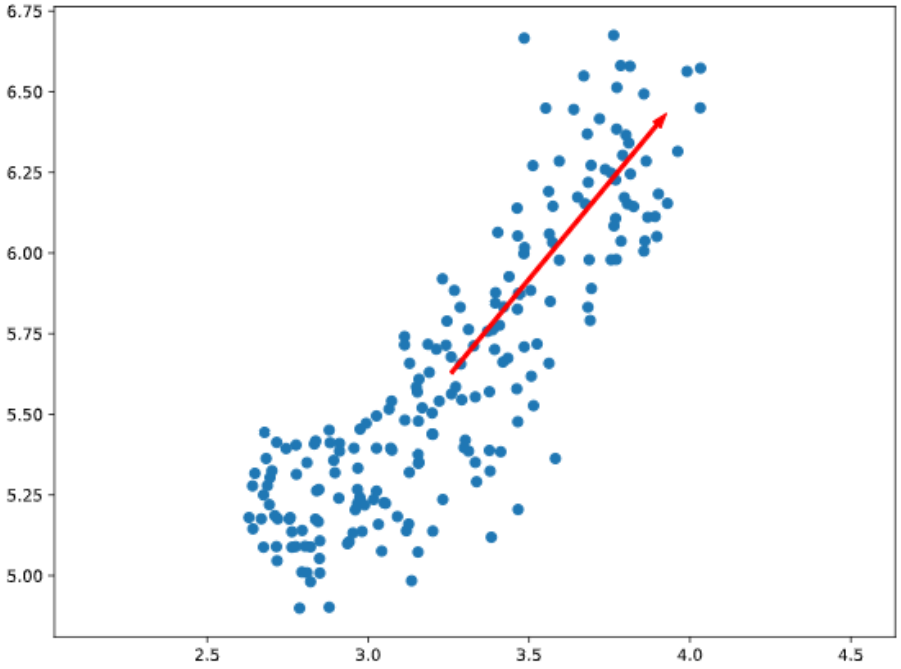
The first principal component

The first principal component of the data is the direction in which the data varies the most. In this exercise, your job is to use PCA to find the first principal component of the length and width measurements of the grain samples, and represent it as an arrow on the scatter plot.

The array `grains` gives the length and width of the grain samples. PyPlot (`plt`) and `PCA` have already been imported for you.

```
# Make a scatter plot of the untransformed points
plt.scatter(grains[:,0], grains[:,1])
```

```
# Create a PCA instance: model
model = PCA()

# Fit model to points
model.fit(grains)

# Get the mean of the grain samples: mean
mean = model.mean_

# Get the first principal component: first_pc
first_pc = model.components_[0,:]

# Plot first_pc as an arrow, starting at mean
plt.arrow(mean[0], mean[1], first_pc[0], first_pc[1], color='red', width=0.01)

# Keep axes on same scale
plt.axis('equal')
plt.show()
```

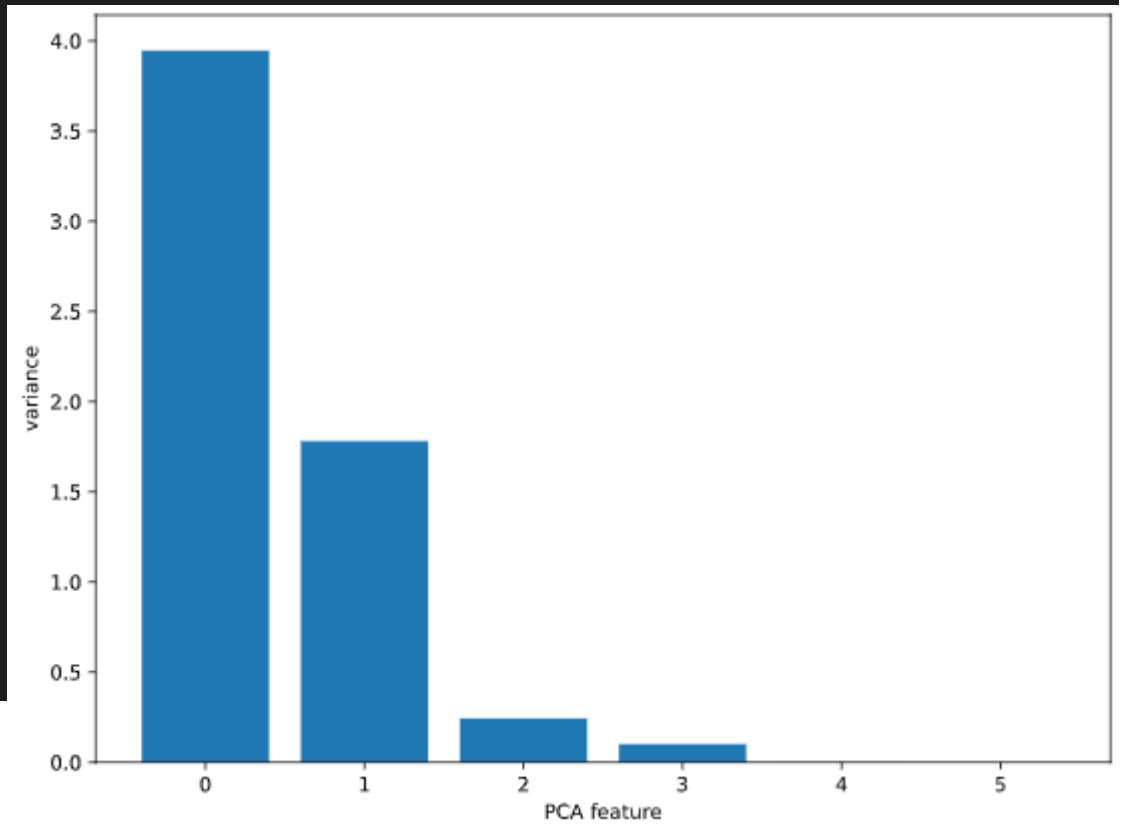


This is the direction in which the grain data varies the most.

Variance of the PCA features

The fish dataset is 6-dimensional. But what is its *intrinsic* dimension? Make a plot of the variances of the PCA features to find out. As before, `samples` is a 2D array, where each row represents a fish. You'll need to standardize the features first.

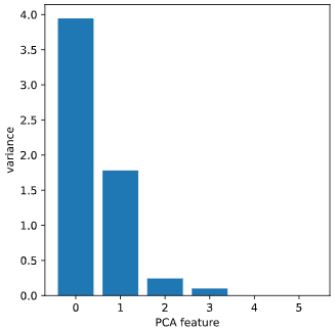
```
# Perform the necessary imports
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import matplotlib.pyplot as plt

# Create scaler: scaler
scaler = StandardScaler()

# Create a PCA instance: pca
pca = PCA()

# Create pipeline: pipeline
pipeline = make_pipeline(scaler, pca)

# Fit the pipeline to 'samples'
pipeline.fit(samples)

# Plot the explained variances
features = range(pca.n_components_)
plt.bar(features, pca.explained_variance_)
plt.xlabel('PCA feature')
plt.ylabel('variance')
plt.xticks(features)
plt.show()
```



It looks like PCA features 0 and 1 have significant variance.

Intrinsic dimension of the fish data

In the previous exercise, you plotted the variance of the PCA features of the fish measurements. Looking again at your plot, what do you think would be a reasonable choice for the "intrinsic dimension" of the fish measurements? Recall that the intrinsic dimension is the number of PCA features with significant variance.



Possible Answers

- 1
- 2
- 5

Since PCA features 0 and 1 have significant variance, the intrinsic dimension of this dataset appears to be 2.

Dimension reduction with PCA

Represents the same data with less features (eg. from 4 to 2)

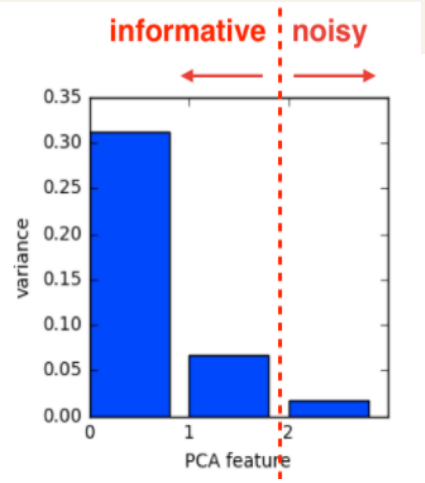
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(samples)
```

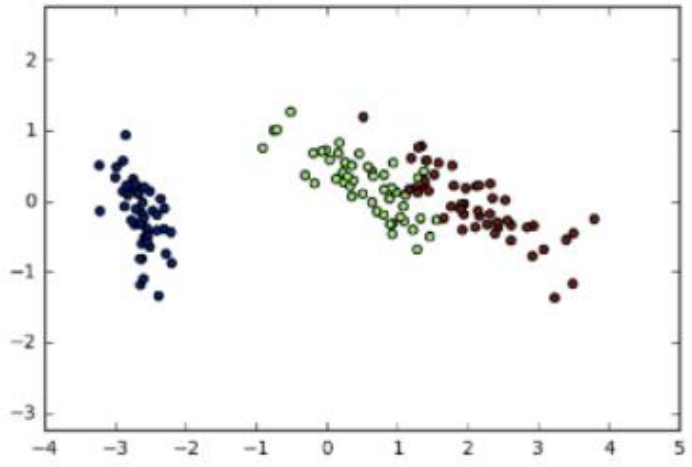
```
PCA(copy=True, ... )
```

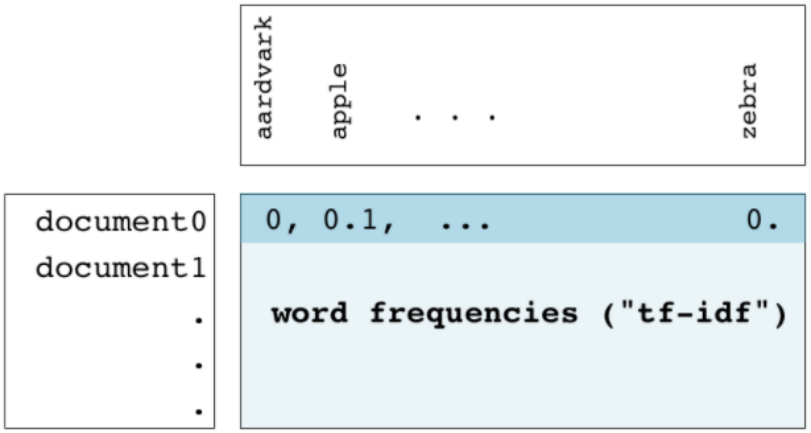
```
transformed = pca.transform(samples)
print(transformed.shape)
```

```
(150, 2)
```

```
import matplotlib.pyplot as plt
xs = transformed[:,0]
ys = transformed[:,1]
plt.scatter(xs, ys, c=species)
plt.show()
```







use scipy.sparse.csr_matrix (instead of numpy array)

```
from sklearn.decomposition import TruncatedSVD
model = TruncatedSVD(n_components=3)
model.fit(documents)  # documents is csr_matrix
TruncatedSVD(algorithm='randomized', ... )
transformed = model.transform(documents)
```

Dimension reduction of the fish measurements

In a previous exercise, you saw that `2` was a reasonable choice for the "intrinsic dimension" of the fish measurements. Now use PCA for dimensionality reduction of the fish measurements, retaining only the 2 most important components.

The fish measurements have already been scaled for you, and are available as `scaled_samples`.

```
# Import PCA
from sklearn.decomposition import PCA

# Create a PCA instance with 2 components: pca
pca = PCA(n_components=2)

# Fit the PCA instance to the scaled samples
pca.fit(scaled_samples)
```

```
# Transform the scaled samples: pca_features
pca_features = pca.transform(scaled_samples)

# Print the shape of pca_features
print(pca_features.shape)
```

```
<script.py> output:
    (85, 2)
```

You've successfully reduced the dimensionality from 6 to 2.

A tf-idf word-frequency array

In this exercise, you'll create a tf-idf word frequency array for a toy collection of documents. For this, use the `TfidfVectorizer` from sklearn. It transforms a list of documents into a word frequency array, which it outputs as a csr_matrix. It has `fit()` and `transform()` methods like other sklearn objects.

You are given a list `documents` of toy documents about pets. Its contents have been printed in the IPython Shell.

```
# Import TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Create a TfidfVectorizer: tfidf
tfidf = TfidfVectorizer()

# Apply fit_transform to document: csr_mat
csr_mat = tfidf.fit_transform(documents)

# Print result of toarray() method
print(csr_mat.toarray())

# Get the words: words
words = tfidf.get_feature_names()

# Print words
print(words)
```

```
<script.py> output:
    [[0.51785612 0.         0.         0.68091856 0.51785612 0.        ]
     [0.         0.         0.51785612 0.         0.51785612 0.68091856]
     [0.51785612 0.68091856 0.51785612 0.         0.         0.        ]]
    ['cats', 'chase', 'dogs', 'meow', 'say', 'woof']
```

Clustering Wikipedia part I

You have learnt that `TruncatedSVD` is able to perform PCA on sparse arrays in csr_matrix format, such as word-frequency arrays. Combine your knowledge of TruncatedSVD and k-means to cluster some popular pages from Wikipedia. In this exercise, build the pipeline. In the next exercise, you'll apply it to the word-frequency array of some Wikipedia articles.

Create a Pipeline object consisting of a TruncatedSVD followed by KMeans. (This time, we've precomputed the word-frequency matrix for you, so there's no need for a TfidfVectorizer).

The Wikipedia dataset you will be working with was obtained from **here**.

```
# Perform the necessary imports
from sklearn.decomposition import TruncatedSVD
from sklearn.cluster import KMeans
from sklearn.pipeline import make_pipeline

# Create a TruncatedSVD instance: svd
svd = TruncatedSVD(n_components=50)

# Create a KMeans instance: kmeans
kmeans = KMeans(n_clusters=6)

# Create a pipeline: pipeline
pipeline = make_pipeline(svd, kmeans)
```

Now that you have set up your pipeline, you will use it in the next exercise to cluster the articles.

Clustering Wikipedia part II

It is now time to put your pipeline from the previous exercise to work! You are given an array `articles` of tf-idf word-frequencies of some popular Wikipedia articles, and a list `titles` of their titles. Use your pipeline to cluster the Wikipedia articles.

A solution to the previous exercise has been pre-loaded for you, so a Pipeline `pipeline` chaining TruncatedSVD with KMeans is available.

```
# Import pandas
import pandas as pd

# Fit the pipeline to articles
pipeline.fit(articles)

# Calculate the cluster labels: labels
labels = pipeline.predict(articles)

# Create a DataFrame aligning labels and titles: df
df = pd.DataFrame({'label': labels, 'article': titles})

# Display df sorted by cluster label
print(df.sort_values('label'))
```

Take a look at the cluster labels and see if you can identify any patterns!

0 = music

1 = football

2 = celebrities

3 = news

4 = IT

5 = medical

```
<script.py> output:
    label                                    article
59      0                                Adam Levine
57      0                    Red Hot Chili Peppers
56      0                                  Skrillex
55      0                            Black Sabbath
54      0                          Arctic Monkeys
53      0                           Stevie Nicks
52      0                              The Wanted
51      0                              Nate Ruess
50      0                           Chad Kroeger
58      0                                  Sepsis
30      1          France national football team
31      1                       Cristiano Ronaldo
32      1                           Arsenal F.C.
33      1                         Radamel Falcao
37      1                               Football
35      1        Colombia national football team
36      1      2014 FIFA World Cup qualification
38      1                                 Neymar
39      1                          Franck Ribéry
34      1                      Zlatan Ibrahimović
26      2                             Mila Kunis
28      2                          Anne Hathaway
27      2                         Dakota Fanning
25      2                          Russell Crowe
29      2                        Jennifer Aniston
23      2                    Catherine Zeta-Jones
22      2                       Denzel Washington
21      2                      Michael Fassbender
20      2                         Angelina Jolie
24      2                           Jessica Biel
10      3                         Global warming
11      3  Nationally Appropriate Mitigation Action
13      3                       Connie Hedegaard
14      3                         Climate change
12      3                           Nigel Lawson
16      3                                 350.org
17      3  Greenhouse gas emissions by the United States
18      3  2010 United Nations Climate Change Conference
19      3  2007 United Nations Climate Change Conference
15      3                         Kyoto Protocol
8       4                                Firefox
1       4                          Alexa Internet
2       4                       Internet Explorer
3       4                             HTTP cookie
4       4                           Google Search
5       4                                  Tumblr
6       4          Hypertext Transfer Protocol
7       4                          Social search
49      4                                Lymphoma
42      4                              Doxycycline
47      4                                   Fever
46      4                              Prednisone
44      4                                    Gout
43      4                                Leukemia
9       4                                LinkedIn
48      4                              Gabapentin
0       4                                HTTP 404
45      5                           Hepatitis C
41      5                           Hepatitis B
40      5                           Tonsillitis
```

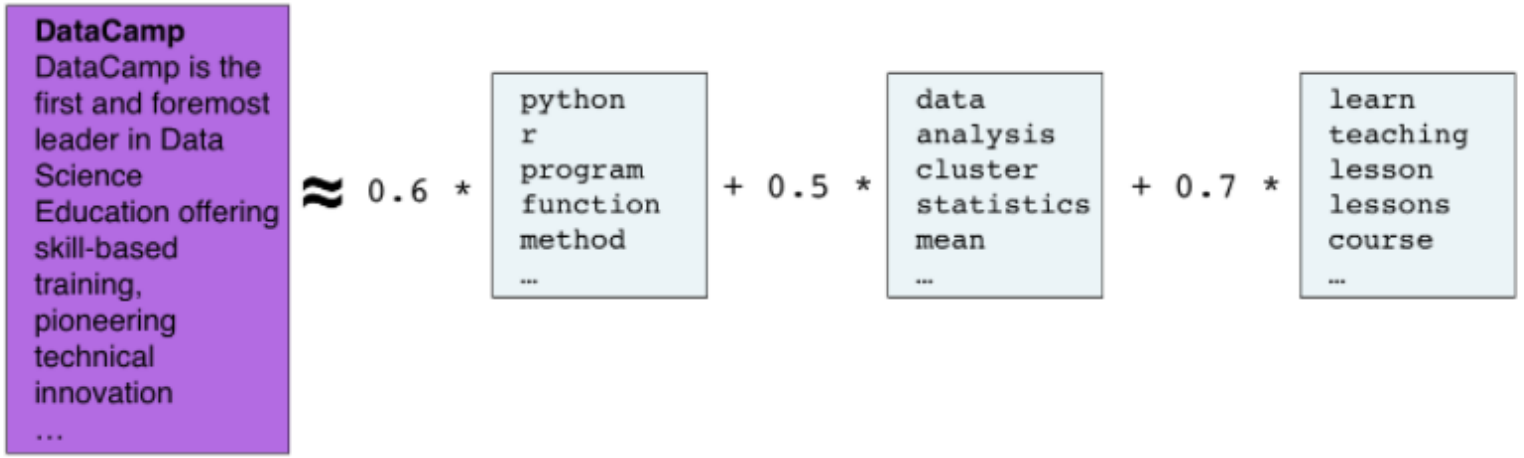
Chapter 4. Discovering interpretable features

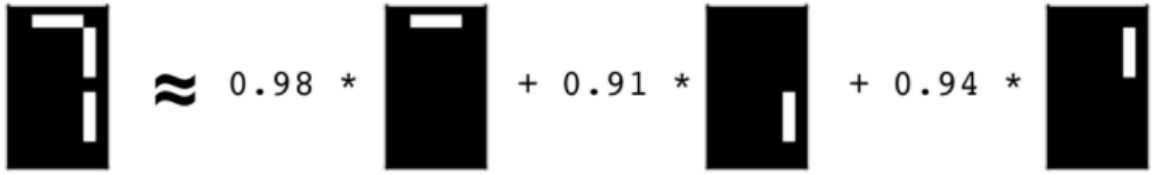
In this chapter, you'll learn about a dimension reduction technique called "**Non-negative matrix factorization**" ("**NMF**") that expresses samples as combinations of interpretable parts. For example, it expresses documents as combinations of topics, and images in terms of commonly occurring visual patterns. You'll also learn to use NMF to build recommender systems that can find you similar articles to read, or musical artists that match your listening history!

Non-negative matrix factorization (NMF)

Dimension reduction technique, interpretable (unlike PCA), easier to understand and explain
NMF cannot be applied to every dataset, sample features required to be “non-negative” (eg. word frequency)

NMF achieves its interpretability by decomposing samples as sums of their parts, expresses dataset as combinations of common themes/patterns






```
from sklearn.decomposition import NMF
model = NMF(n_components=2)
model.fit(samples)
```

```
NMF(alpha=0.0, ... )
```

```
nmf_features = model.transform(samples)
```

```
print(model.components_)
```

```
[[ 0.01  0.    2.13  0.54]
 [ 0.99  1.47  0.    0.5 ]]
```

```
print(nmf_features)
```

```
[[ 0.    0.2 ]
 [ 0.19  0.  ]
  ...
 [ 0.15  0.12]]
```

```
print(samples[i,:])
```

```
[ 0.12  0.18  0.32  0.14]
```

```
print(nmf_features[i,:])
```

```
[ 0.15  0.12]
```

	course	datacamp	potato	the
document0	0.2,	0.3,	0.0,	0.1
document1	0.0,	0.0,	0.4,	0.1
...		...		

matrix multiplication

```
  0.15 *
+ 0.12 *
```

```
[[ 0.01   0.     2.13   0.54 ]
 [ 0.99   1.47   0.     0.5  ]]
```

model.components_

```
[ 0.1203  0.1764  0.3195  0.141 ]
```

reconstruction of sample

Non-negative data

Possible Answers

Which of the following 2-dimensional arrays are examples of non-negative data?

1. A tf-idf word-frequency array.
2. An array daily stock market price movements (up and down), where each row represents a company.
3. An array where rows are customers, columns are products and entries are 0 or 1, indicating whether a customer has purchased a product.

- ○ 1 only
- ○ 2 and 3
- ◉ 1 and 3

Stock prices can go down as well as up, so an array of daily stock market price movements is not an example of non-negative data.

NMF applied to Wikipedia articles

NMF can be applied to transform a toy word-frequency array. Now it's your turn to apply NMF, this time using the tf-idf word-frequency array of Wikipedia articles, given as a csr matrix `articles`. Here, fit the model and transform the articles. In the next exercise, you'll explore the result.

```
# Import NMF
from sklearn.decomposition import NMF

# Create an NMF instance: model
model = NMF(n_components=6)

# Fit the model to articles
model.fit(articles)

# Transform the articles: nmf_features
nmf_features = model.transform(articles)

# Print the NMF features
print(nmf_features.round(2))
```

```
<script.py> output:
    [[0.   0.   0.   0.   0.   0.44]
     [0.   0.   0.   0.   0.   0.57]
     [0.   0.   0.   0.   0.   0.4 ]
     [0.   0.   0.   0.   0.   0.38]
     [0.   0.   0.   0.   0.   0.49]
     [0.01 0.01 0.01 0.03 0.   0.33]
     [0.   0.   0.02 0.   0.01 0.36]
     [0.   0.   0.   0.   0.   0.49]
     [0.02 0.01 0.   0.02 0.03 0.48]
     [0.01 0.03 0.03 0.07 0.02 0.34]
     [0.   0.   0.53 0.   0.03 0.  ]
     [0.   0.   0.36 0.   0.   0.  ]
     [0.01 0.01 0.31 0.06 0.01 0.02]
     [0.   0.01 0.34 0.01 0.   0.  ]
     [0.   0.   0.43 0.   0.04 0.  ]
     [0.   0.   0.48 0.   0.   0.  ]
     [0.01 0.02 0.38 0.03 0.   0.01]
     [0.   0.   0.48 0.   0.   0.  ]
```

Let's explore the meaning of these features in the next exercise!

NMF features of the Wikipedia articles

Now you will explore the NMF features you created in the previous exercise. A solution to the previous exercise has been pre-loaded, so the array `nmf_features` is available. Also available is a list `titles` giving the title of each Wikipedia article.

When investigating the features, notice that for both actors, the NMF feature 3 has by far the highest value. This means that both articles are reconstructed using mainly the 3rd NMF component. This is the reason why: NMF components represent topics (for instance, acting!).

```
# Import pandas
import pandas as pd

# Create a pandas DataFrame: df
df = pd.DataFrame(nmf_features, index=titles)

# Print the row for 'Anne Hathaway'
print(df.loc['Anne Hathaway'])

# Print the row for 'Denzel Washington'
print(df.loc['Denzel Washington'])
```

```
<script.py> output:
    0    0.003845
    1    0.000000
    2    0.000000
    3    0.575711
    4    0.000000
    5    0.000000
    Name: Anne Hathaway, dtype: float64
    0    0.000000
    1    0.005601
    2    0.000000
    3    0.422380
    4    0.000000
    5    0.000000
    Name: Denzel Washington, dtype: float64
```

Notice that for both actors, the NMF feature 3 has by far the highest value. This means that both articles are reconstructed using mainly the 3rd NMF component. NMF components represent topics (for instance, acting!).

NMF reconstructs samples

In this exercise, you'll check your understanding of how NMF reconstructs samples from its components using the NMF feature values. On the right are the components of an NMF model. If the NMF feature values of a sample are `[2, 1]`, then which of the following is *most likely* to represent the original sample? A pen and paper will help here!

```
[[1.  0.5 0. ]
 [0.2 0.1 2.1]]
```

Possible Answers

- (●) `[2.2, 1.1, 2.1]`.
- ( ) `[0.5, 1.6, 3.1]`.
- ( ) `[-4.0, 1.0, -2.0]`.

NMF learns interpretable parts

```
print(articles.shape)
```

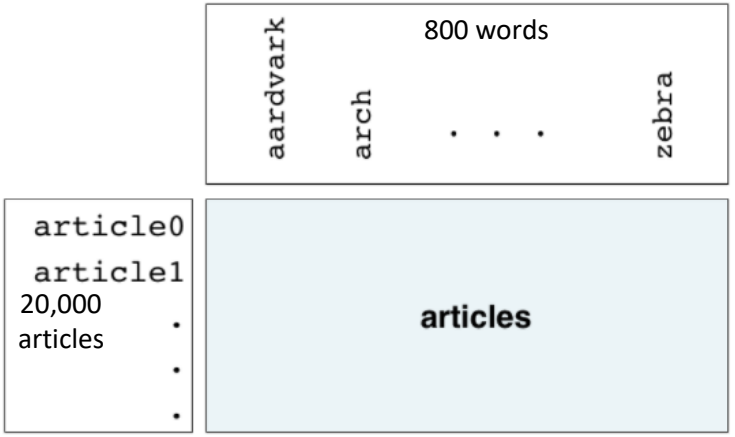
```
(20000, 800)
```

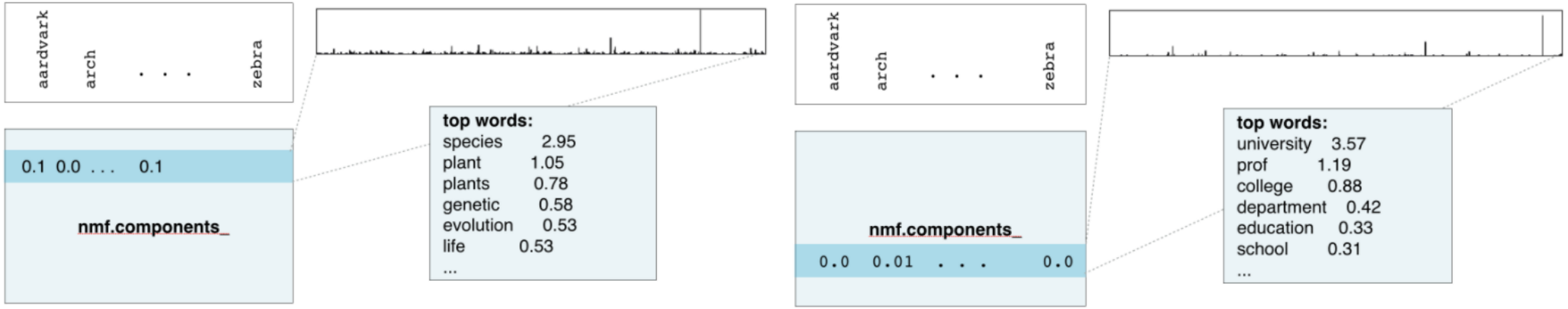
```
from sklearn.decomposition import NMF
nmf = NMF(n_components=10)
nmf.fit(articles)
```

```
NMF(alpha=0.0, ... )
```

```
print(nmf.components_.shape)
```

```
(10, 800)
```



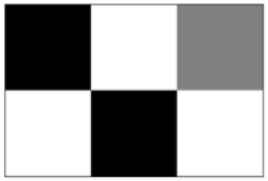


NMF components represent topics

NMF features combine topics into documents

For images, NMF components are parts of images (eg. corners, edges, lines, curves, etc)

Grayscale image



Convert to 2D array

```
[[ 0.   1.   0.5]
 [ 1.   0.   1. ]]
```

Flatten to 1D array

```
[ 0.   1.   0.5   1.   0.   1. ]
```

```
[[ 0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.7  0.8  0.   0.   0. ]
 [ 0.   0.   0.8  0.8  0.9  1.   0.   0. ]
 [ 0.   0.7  0.9  0.9  1.   1.   1.   0. ]
 [ 0.   0.8  0.9  1.   1.   1.   1.   0. ]
 [ 0.   0.   0.9  1.   1.   1.   0.   0. ]
 [ 0.   0.   0.   0.9  1.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0. ]]
```

```
print(sample)
```

```
[ 0.   1.   0.5  1.   0.   1. ]
```

```
bitmap = sample.reshape((2, 3))
print(bitmap)
```

```
[[ 0.   1.   0.5]
 [ 1.   0.   1. ]]
```

```
from matplotlib import pyplot as plt
plt.imshow(bitmap, cmap='gray', interpolation='nearest')
plt.show()
```

NMF learns topics of documents

You have learnt that when NMF is applied to documents, the components correspond to topics of documents, and the NMF features reconstruct the documents from the topics. Verify this for yourself for the NMF model that you built earlier using the Wikipedia articles. Previously, you saw that the 3rd NMF feature value was high for the articles about actors Anne Hathaway and Denzel Washington. In this exercise, identify the topic of the corresponding NMF component.

The NMF model you built earlier is available as `model`, while `words` is a list of the words that label the columns of the word-frequency array.

After you are done, take a moment to recognise the topic that the articles about Anne Hathaway and Denzel Washington have in common!

```
# Import pandas
import pandas as pd

# Create a DataFrame: components_df
components_df = pd.DataFrame(model.components_, columns=words)

# Print the shape of the DataFrame
print(components_df.shape)

# Select row 3: component
component = components_df.iloc[3]

# Print result of nlargest
print(component.nlargest())
```

```
<script.py> output:
    (6, 13125)
    film       0.627877
    award      0.253131
    starred    0.245284
    role       0.211451
    actress    0.186398
    Name: 3, dtype: float64
```

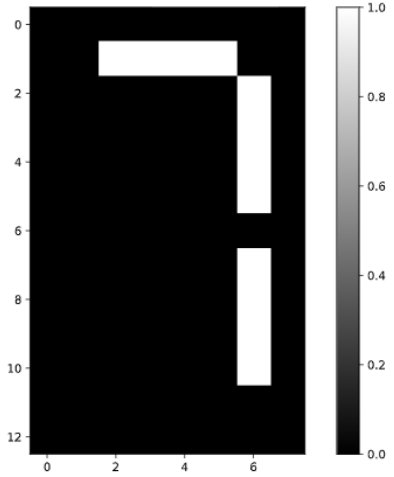
Take a moment to recognise the topics that the articles about Anne Hathaway and Denzel Washington have in common!

Explore the LED digits dataset

In the following exercises, you'll use NMF to decompose grayscale images into their commonly occurring patterns. Firstly, explore the image dataset and see how it is encoded as an array. You are given 100 images as a 2D array `samples`, where each row represents a single 13x8 image. The images in your dataset are pictures of a LED digital display.

```python
# Import pyplot
from matplotlib import pyplot as plt

# Select the 0th row: digit
digit = samples[0,:]

# Print digit
print(digit)

# Reshape digit to a 13x8 array: bitmap
bitmap = digit.reshape((13, 8))

# Print bitmap
print(bitmap)

# Use plt.imshow to display bitmap
plt.imshow(bitmap, cmap='gray', interpolation='nearest')
plt.colorbar()
plt.show()
```

```
<script.py> output:
    [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0.
     0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0.
     0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0.
     0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
     0. 0. 0. 0. 0. 0. 0. 0.]
    [[0. 0. 0. 0. 0. 0. 0. 0.]
     [0. 0. 1. 1. 1. 1. 0. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 0. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 1. 0.]
     [0. 0. 0. 0. 0. 0. 0. 0.]
     [0. 0. 0. 0. 0. 0. 0. 0.]]
```



You'll explore this dataset further in the next exercise and see for yourself how NMF can learn the parts of images.

NMF learns the parts of images

Now use what you've learned about NMF to decompose the digits dataset. You are again given the digit images as a 2D array `samples`. This time, you are also provided with a function `show_as_image()` that displays the image encoded by any 1D array:

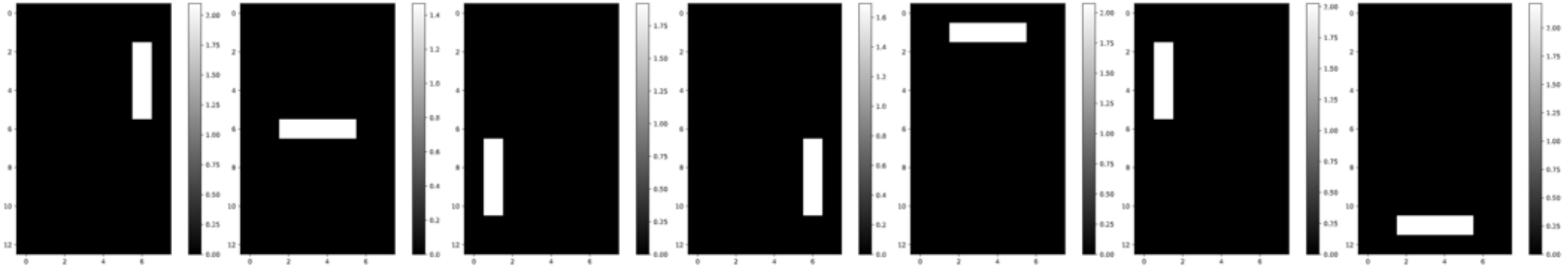
```
def show_as_image(sample):
    bitmap = sample.reshape((13, 8))
    plt.figure()
    plt.imshow(bitmap, cmap='gray', interpolation='nearest')
    plt.colorbar()
    plt.show()
```

After you are done, take a moment to look through the plots and notice how NMF has expressed the digit as a sum of the components!

```
# Import NMF
from sklearn.decomposition import NMF

# Create an NMF model: model
model = NMF(n_components=7)

# Apply fit_transform to samples: features
features = model.fit_transform(samples)

# Call show_as_image on each component
for component in model.components_:
    show_as_image(component)

# Select the 0th row of features: digit_features
digit_features = features[0,:]

# Print digit_features
print(digit_features)
```

```
<script.py> output:
    [4.76823559e-01 0.00000000e+00 0.00000000e+00 5.90605054e-01
     4.81559442e-01 0.00000000e+00 7.37557191e-16]
```

Take a moment to look through the plots and notice how NMF has expressed the digit as a sum of the components!



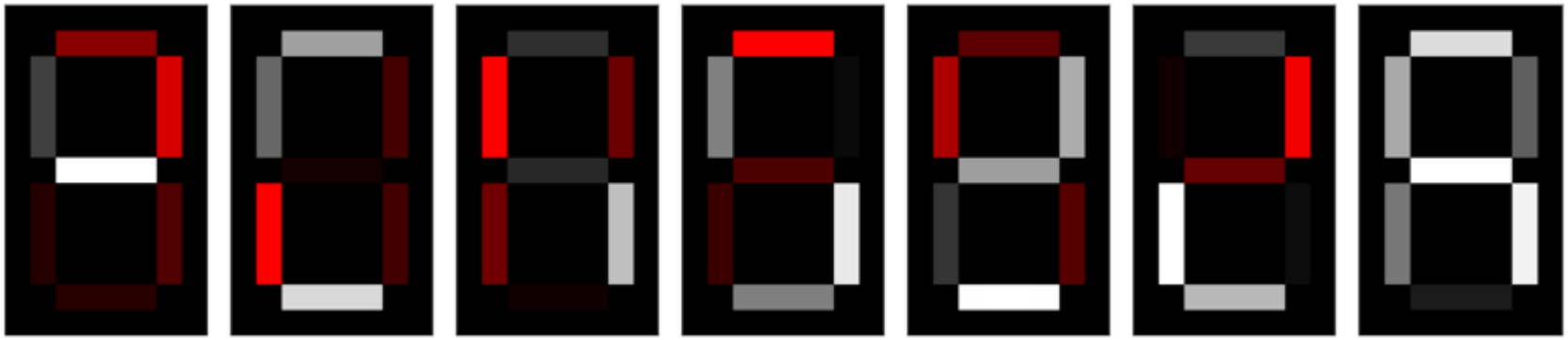
PCA doesn't learn parts

Unlike NMF, PCA *doesn't* learn the parts of things. Its components do not correspond to topics (in the case of documents) or to parts of images, when trained on images. Verify this for yourself by inspecting the components of a PCA model fit to the dataset of LED digit images from the previous exercise. The images are available as a 2D array `samples`. Also available is a modified version of the `show_as_image()` function which colors a pixel red if the value is negative.

After submitting the answer, notice that the components of PCA do not represent meaningful parts of images of LED digits!

```
# Import PCA
from sklearn.decomposition import PCA

# Create a PCA instance: model
model = PCA(n_components=7)

# Apply fit_transform to samples: features
features = model.fit_transform(samples)

# Call show_as_image on each component
for component in model.components_:
    show_as_image(component)
```

Great work! Notice that the components of PCA do not represent meaningful parts of images of LED digits!

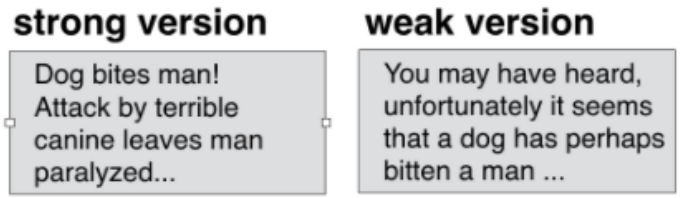


Building recommender systems using NMF

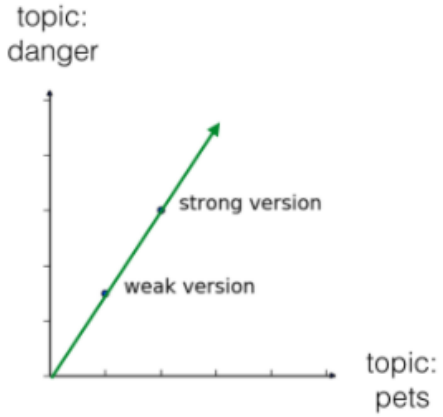
To recommend articles that are similar to the article currently being read by a customer. Similar articles should have similar topics.

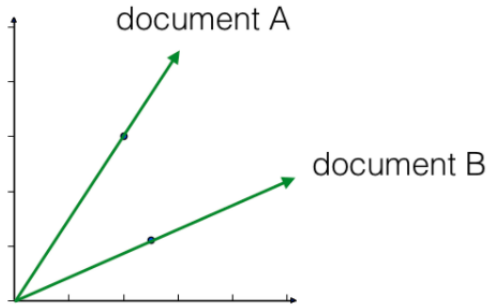
Apply NMF to the word-frequency array, and to use the resulting NMF features (which are topic mixture of an article).

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=6)
nmf_features = nmf.fit_transform(articles)
```



Similar topic
cosine similarity → 1





Different topics
cosine similarity → 0

```
from sklearn.preprocessing import normalize
norm_features = normalize(nmf_features)
# if has index 23
current_article = norm_features[23,:]
similarities = norm_features.dot(current_article)
print(similarities)
```

```
[ 0.7150569   0.26349967 ..., 0.20323616  0.05047817]
```

- Titles given as a list: `titles`

```
import pandas as pd
norm_features = normalize(nmf_features)
df = pd.DataFrame(norm_features, index=titles)
current_article = df.loc['Dog bites man']
similarities = df.dot(current_article)
```

```
print(similarities.nlargest())
```

```
Dog bites man                    1.000000
Hound mauls cat                  0.979946
Pets go wild!                    0.979708
Dachshunds are dangerous         0.949641
Our streets are no longer safe   0.900474
dtype: float64
```

Which articles are similar to 'Cristiano Ronaldo'?

You have learnt how to use NMF features and the cosine similarity to find similar articles. Apply this to your NMF model for popular Wikipedia articles, by finding the articles most similar to the article about the footballer Cristiano Ronaldo. The NMF features you obtained earlier are available as `nmf_features`, while `titles` is a list of the article titles.

```
# Perform the necessary imports
import pandas as pd
from sklearn.preprocessing import normalize

# Normalize the NMF features: norm_features
norm_features = normalize(nmf_features)
```

```python
# Create a DataFrame: df
df = pd.DataFrame(norm_features, index=titles)

# Select the row corresponding to 'Cristiano Ronaldo': article
article = df.loc['Cristiano Ronaldo']

# Compute the dot products: similarities
similarities = df.dot(article)

# Display those with the largest cosine similarity
print(similarities.nlargest())
```

```
<script.py> output:
    Cristiano Ronaldo                1.000000
    Franck Ribéry                    0.999972
    Radamel Falcao                   0.999942
    Zlatan Ibrahimović               0.999942
    France national football team    0.999923
    dtype: float64
```

Although you may need to know a little about football (or soccer, depending on where you're from!) to be able to evaluate for yourself the quality of the computed similarities!

Recommend musical artists part I

In this exercise and the next, you'll use what you've learned about NMF to recommend popular music artists! You are given a sparse array `artists` whose rows correspond to artists and whose columns correspond to users. The entries give the number of times each artist was listened to by each user.

In this exercise, build a pipeline and transform the array into normalized NMF features. The first step in the pipeline, `MaxAbsScaler`, transforms the data so that all users have the same influence on the model, regardless of how many different artists they've listened to. In the next exercise, you'll use the resulting normalized NMF features for recommendation!

```python
# Perform the necessary imports
from sklearn.decomposition import NMF
from sklearn.preprocessing import Normalizer, MaxAbsScaler
from sklearn.pipeline import make_pipeline

# Create a MaxAbsScaler: scaler
scaler = MaxAbsScaler()

# Create an NMF model: nmf
```

```
nmf = NMF(n_components=20)

# Create a Normalizer: normalizer
normalizer = Normalizer()

# Create a pipeline: pipeline
pipeline = make_pipeline(scaler, nmf, normalizer)

# Apply fit_transform to artists: norm_features
norm_features = pipeline.fit_transform(artists)
```

Now that you've computed the normalized NMF features, you'll use them in the next exercise to recommend musical artists!

Recommend musical artists part II

Suppose you were a big fan of Bruce Springsteen - which other musicial artists might you like? Use your NMF features from the previous exercise and the cosine similarity to find similar musical artists. A solution to the previous exercise has been run, so `norm_features` is an array containing the normalized NMF features as rows. The names of the musical artists are available as the list `artist_names`.

```
# Import pandas
import pandas as pd

# Create a DataFrame: df
df = pd.DataFrame(norm_features, index=artist_names)

# Select row of 'Bruce Springsteen': artist
artist = df.loc['Bruce Springsteen']

# Compute cosine similarities: similarities
similarities = df.dot(artist)

# Display those with highest cosine similarity
print(similarities.nlargest())
```

```
<script.py> output:
    Bruce Springsteen    1.000000
    Neil Young           0.955896
    Van Morrison         0.872452
    Leonard Cohen        0.864763
    Bob Dylan            0.859047
    dtype: float64
```

Course completed!

Recap topics covered:

- Characteristics of unsupervised learning
- Apply unsupervised techniques to real-world datasets
- Building knowledge of coding with python
- Using scikit-learn and scipy for unsupervised learning challenges
- Clustering techniques
- Dimension reduction techniques
- Clustering Wikipedia documents by the words they contain
- Recommending musical artists to consumers

Happy learning!