Chapter 1)

1)

- mport `KMeans` from `sklearn.cluster`.
- Using `KMeans()`, create a `KMeans` instance called `model` to find `3`clusters. To specify the number of clusters, use the `n_clusters` keyword argument.
- Use the `.fit()` method of `model` to fit the model to the array of points `points`.
- Use the `.predict()` method of `model` to predict the cluster labels of `new_points`, assigning the result to `labels`.
- Hit 'Submit Answer' to see the cluster labels of `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)
```

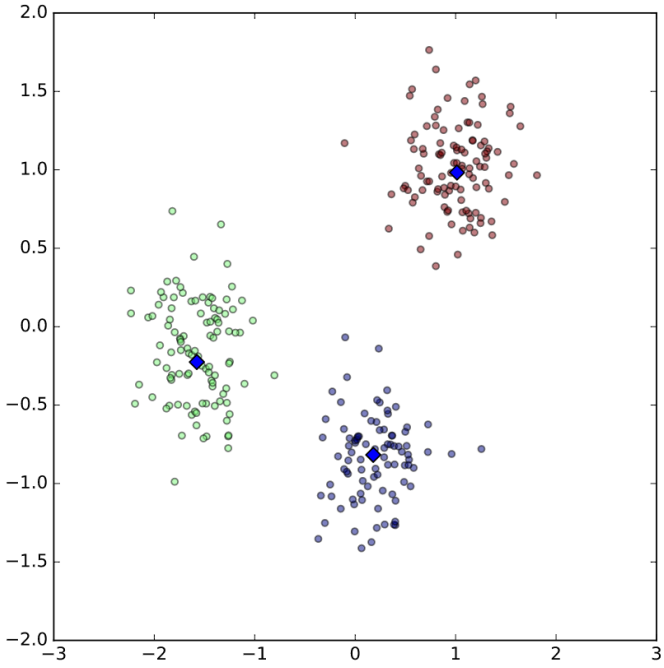
```
[0 2 1 0 2 0 2 2 2 1 0 2 2 1 1 2 1 1 2 2 1 2 0 2 0 1 2 1 1 0 0 2 2 2 1 0 2
 2 0 2 1 0 0 1 0 2 1 1 2 2 2 2 1 1 0 0 1 1 1 0 0 2 2 2 0 2 1 2 0 1 0 0 0 2
 0 1 1 0 2 1 0 1 0 2 1 2 1 0 2 2 2 0 2 2 0 1 1 1 1 0 2 0 1 1 0 0 2 0 1 1 0
 1 1 1 2 2 2 2 1 1 2 0 2 1 2 0 1 2 1 1 2 1 2 1 0 2 0 0 2 1 0 2 0 0 1 2 2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

2)

- import `matplotlib.pyplot` as `plt`.
- Assign column `0` of `new_points` to `xs`, and column `1` of `new_points` to `ys`.

- Make a scatter plot of `xs` and `ys`, specifying the `c=labels` keyword arguments to color the points by their cluster label. Also specify `alpha=0.5`.
- Compute the coordinates of the centroids using the `.cluster_centers_` attribute of `model`.
- Assign column `0` of `centroids` to `centroids_x`, and column `1` of `centroids` to `centroids_y`.
- Make a scatter plot of `centroids_x` and `centroids_y`, using `'D'` (a diamond) as a marker by specifying the `marker` parameter. Set the size of the markers to be `50` using `s=50`.

```
# Import pyplot
import matplotlib.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()
```



Note)

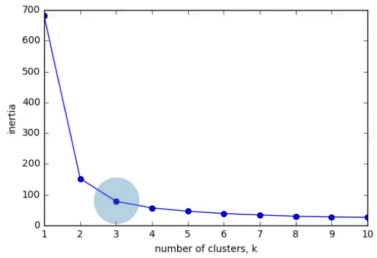
Inertia measures clustering quality

- Measures how spread out the clusters are (*lower* is better)
- Distance from each sample to centroid of its cluster
- After **fit()**, available as attribute **inertia_**
- k-means attempts to minimize the inertia when choosing clusters

```
In [1]: from sklearn.cluster import KMeans

In [2]: model = KMeans(n_clusters=3)

In [3]: model.fit(samples)

In [4]: print(model.inertia_)
78.9408414261
```

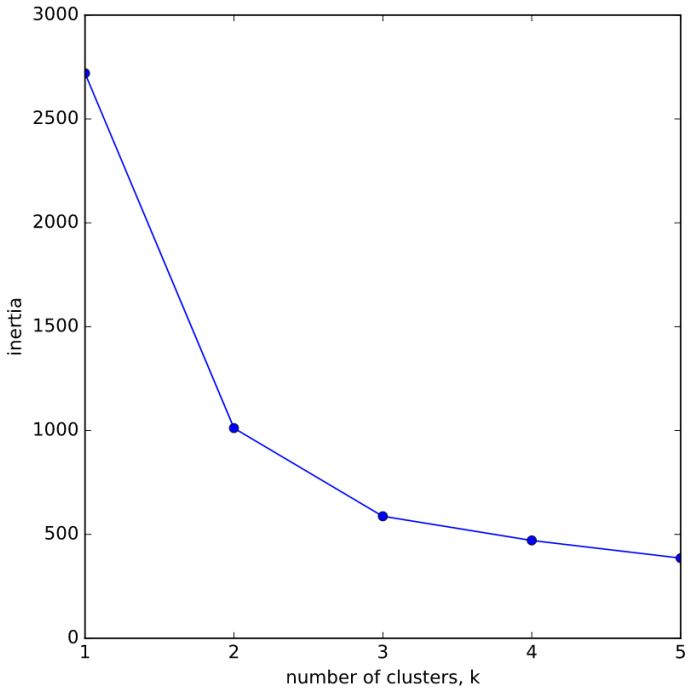
How many clusters to choose?

- A good clustering has tight clusters (so low inertia)
- ... but not too many clusters!
- Choose an "elbow" in the inertia plot
- Where inertia begins to decrease more slowly
- E.g. for iris dataset, 3 is a good choice



3)

```
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()
```



4)

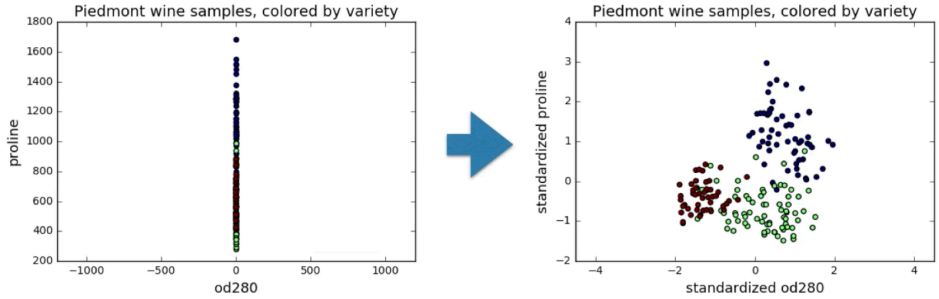
```
# Create a KMeans model with 3 clusters: model
model = KMeans(n_clusters=3)

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

# Create a DataFrame with 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)
```

varieties	Canadian wheat	Kama wheat	Rosa wheat
labels			
0	68	9	0
1	0	1	60
2	2	60	10

Note)

StandardScaler

- In kmeans: feature variance = feature influence
- **`StandardScaler`** transforms each feature to have mean 0 and variance 1
- Features are said to be "standardized"



Pipelines combine multiple steps

```
In [1]: from sklearn.preprocessing import StandardScaler

In [2]: from sklearn.cluster import KMeans

In [3]: scaler = StandardScaler()

In [4]: kmeans = KMeans(n_clusters=3)

In [5]: from sklearn.pipeline import make_pipeline

In [6]: pipeline = make_pipeline(scaler, kmeans)

In [7]: pipeline.fit(samples)
Out[7]: Pipeline(steps=...)

In [8]: labels = pipeline.predict(samples)
```

Feature standardization improves clustering

```
In [9]: df = pd.DataFrame({'labels': labels, 'varieties': varieties})

In [10]: ct = pd.crosstab(df['labels'], df['varieties'])

In [11]: print(ct)
varieties  Barbera  Barolo  Grignolino
labels
0                0      59           3
1               48       0           3
2                0       0          65
```

***Without* feature standardization was very bad:**

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

sklearn preprocessing steps

- **StandardScaler** is a "preprocessing" step
- **MaxAbsScaler** and **Normalizer** are other examples

5)

- Import:
  - `make_pipeline` from `sklearn.pipeline`.
  - `StandardScaler` from `sklearn.preprocessing`.
  - `KMeans` from `sklearn.cluster`.
- Create an instance of `StandardScaler` called `scaler`.
- Create an instance of `KMeans` with `4` clusters called `kmeans`.
- Create a pipeline called `pipeline` that chains `scaler` and `kmeans`. To do this, you just need to pass them in as arguments to `make_pipeline()`.

```
# 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)
```

5)

```
# Import pandas
import pandas as pd

# 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)
```

species	Bream	Pike	Roach	Smelt
labels				
0	33	0	1	0
1	0	0	0	13
2	0	17	0	0
3	1	0	19	1

6)

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.

```
# 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)
```

7)

```
# Import pandas
import pandas as pd

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

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

# Display df sorted by cluster label
print(df.sort_values('labels',ascending=True))
```

	companies	labels
59	Yahoo	0
2	Amazon	0
32	3M	1
23	IBM	1
39	Pfizer	1
44	Schlumberger	1
12	Chevron	1
10	ConocoPhillips	1
13	DuPont de Nemours	1
57	Exxon	1
53	Valero Energy	1
8	Caterpillar	1
49	Total	2
37	Novartis	2
52	Unilever	2
19	GlaxoSmithKline	2
41	Philip Morris	2
42	Royal Dutch Shell	2
46	Sanofi-Aventis	2
31	McDonalds	2
6	British American Tobacco	2
43	SAP	2
30	MasterCard	2
51	Texas instruments	3
33	Microsoft	3
47	Symantec	3
50	Taiwan Semiconductor Manufacturing	3
0	Apple	3
11	Cisco	3
24	Intel	3
14	Dell	3
29	Lookheed Martin	4
54	Walgreen	4
4	Boeing	4
36	Northrop Grumman	4
20	Home Depot	5
16	General Electrics	5
58	Xerox	5
28	Coca Cola	6
38	Pepsi	6
40	Procter Gamble	7
27	Kimberly-Clark	7
56	Wal-Mart	7
25	Johnson & Johnson	7
9	Colgate-Palmolive	7
1	AIG	8

18	Goldman Sachs	8
17	Google/Alphabet	8
55	Wells Fargo	8
35	Navistar	8
3	American express	8
5	Bank of America	8
26	JPMorgan Chase	8
21	Honda	9
22	HP	9
45	Sony	9
15	Ford	9
34	Mitsubishi	9
7	Canon	9
48	Toyota	9

Chapter 2)

Note)

Hierarchical clustering with SciPy

- Given **samples** (the array of scores), and **country_names**

```
In [1]: import matplotlib.pyplot as plt

In [2]: from scipy.cluster.hierarchy import linkage, dendrogram

In [3]: mergings = linkage(samples, method='complete')

In [4]: dendrogram(mergings,
   ...:            labels=country_names,
   ...:            leaf_rotation=90,
   ...:            leaf_font_size=6)

In [5]: plt.show()
```

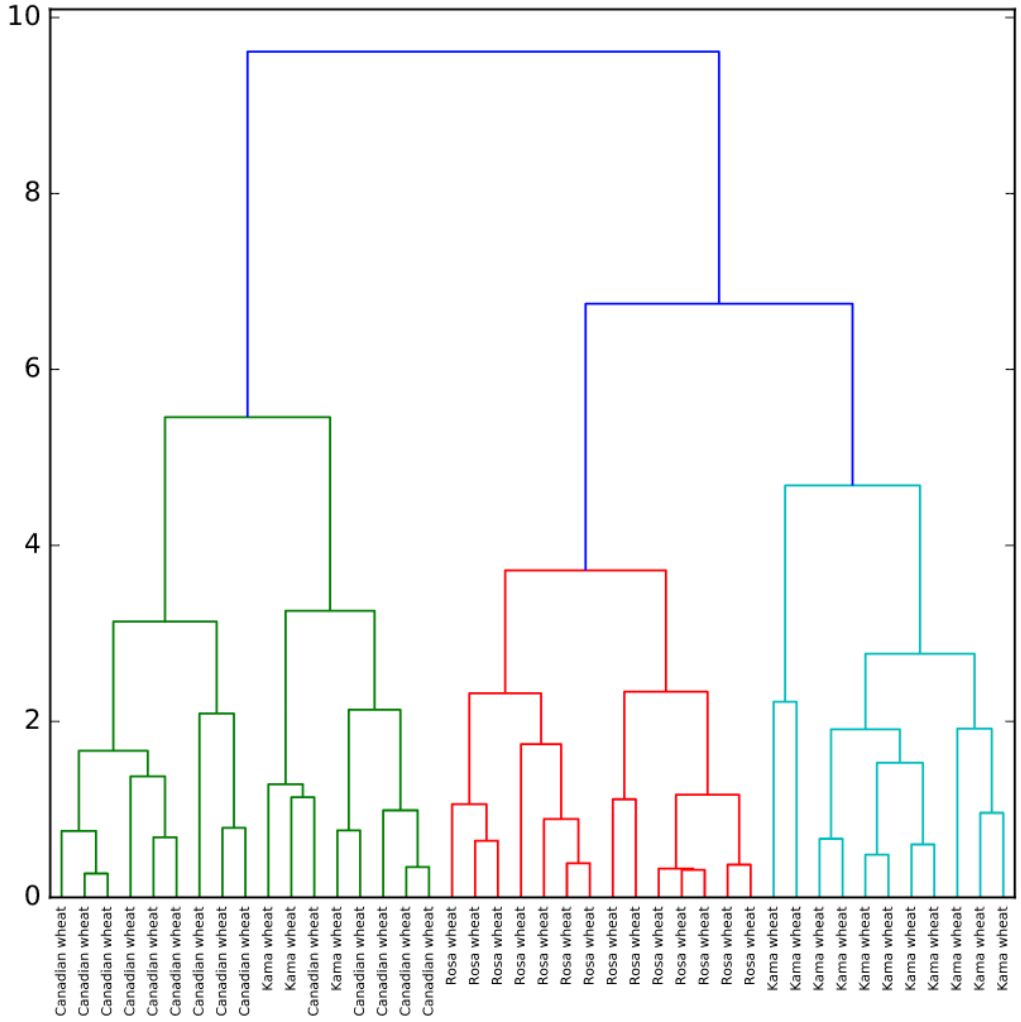
1)

Hierarchical clustering of the grain data

In the video, you learned 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()
```



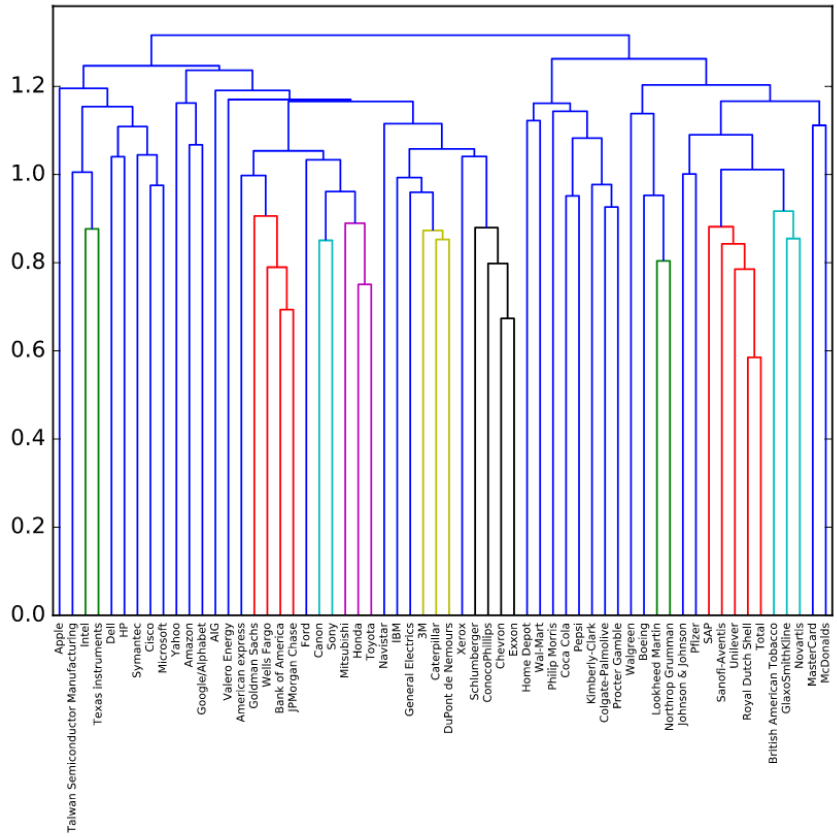
2)

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 `sklearn.cluster.hierarchy`, and PyPlot has been imported as `plt`.

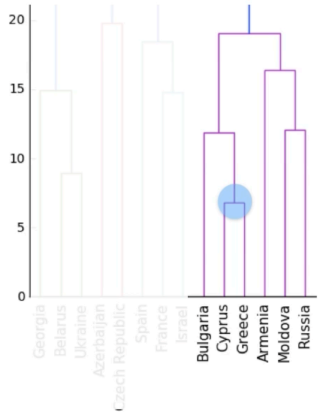
```
# Import normalize
from sklearn.preprocessing import normalize

# 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()
```



Note)

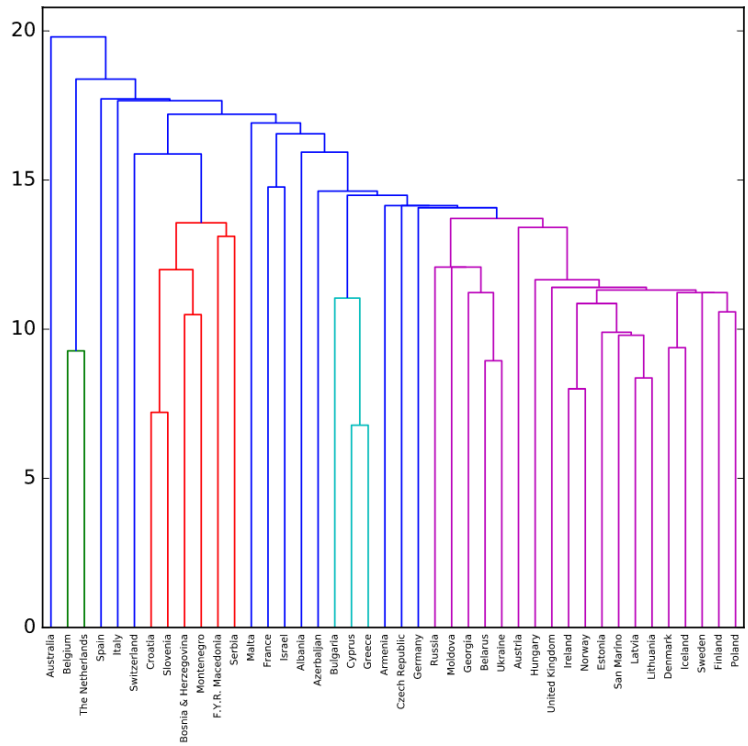
Dendrograms show cluster distances

- Height on dendrogram = distance between merging clusters
- E.g. clusters with only Cyprus and Greece had distance approx. 6



3)

```
# 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()
```



4)

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.

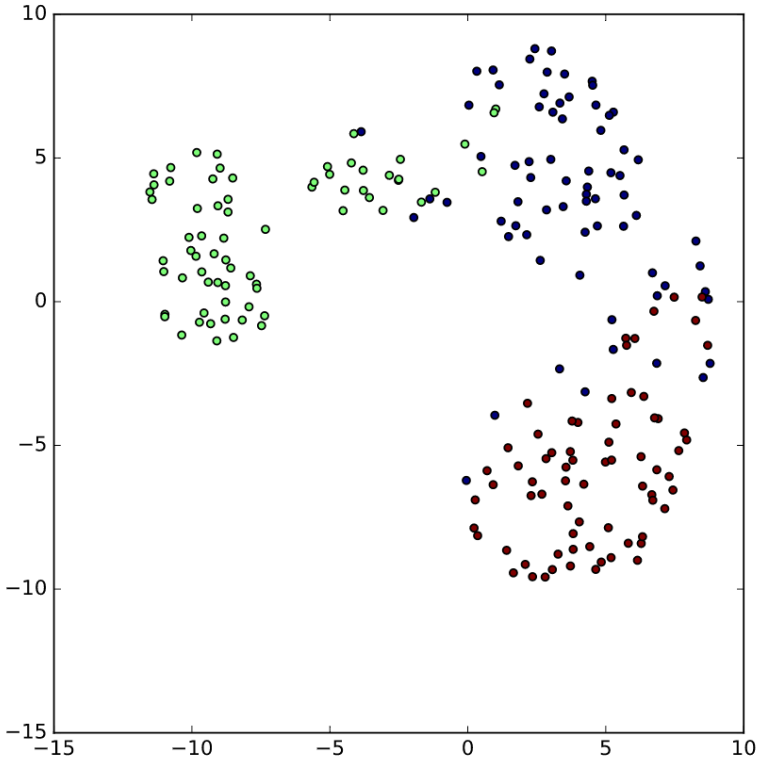
```
# Perform the necessary imports
import pandas as pd
from scipy.cluster.hierarchy import fcluster

# 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)
```

```
varieties  Canadian wheat  Kama wheat  Rosa wheat
labels
1                     14           3           0
2                      0           0          14
3                      0          11           0
```

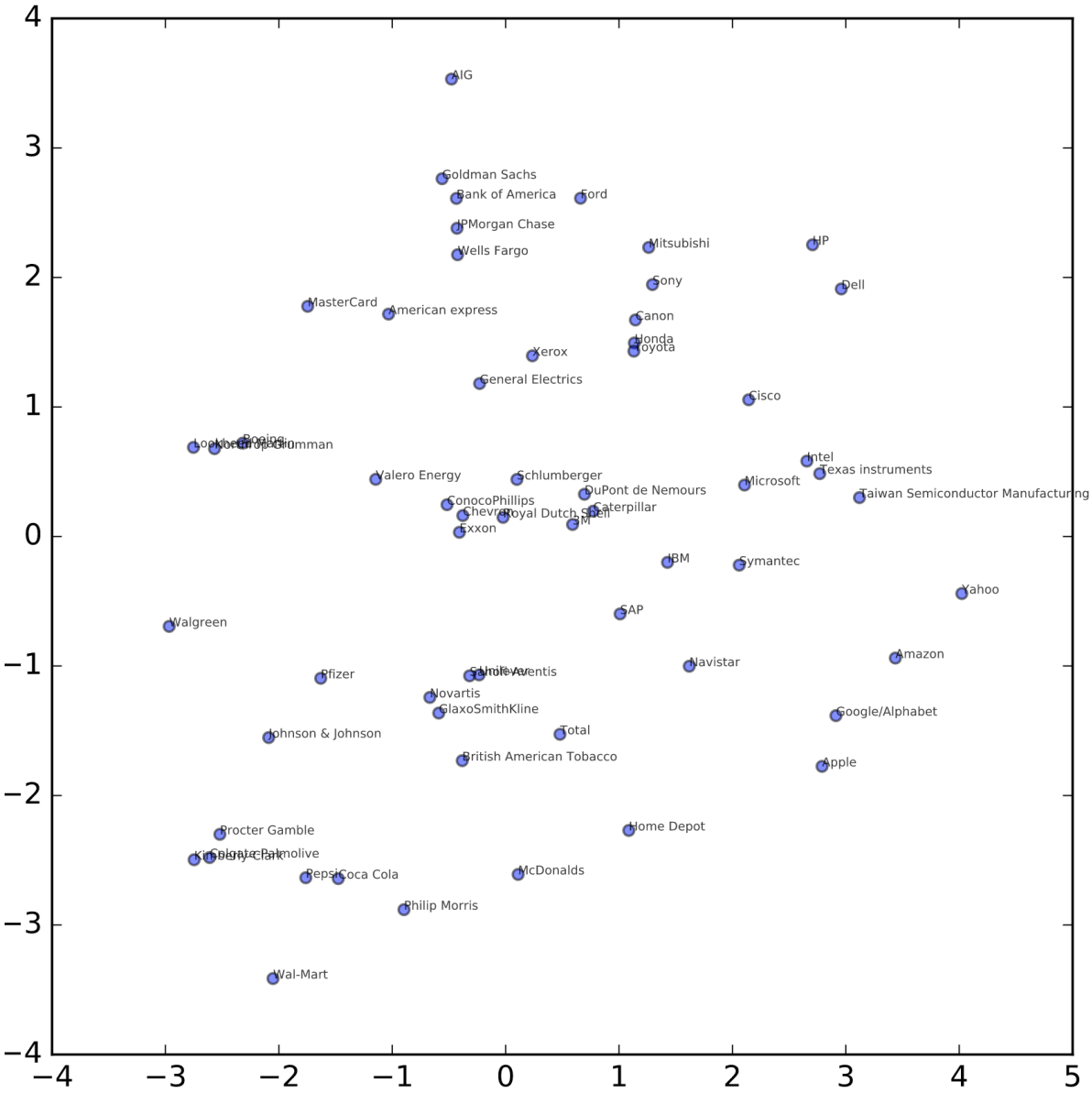
5)

```
# 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()
```



6)

```
# 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()
```



Chapter 3)

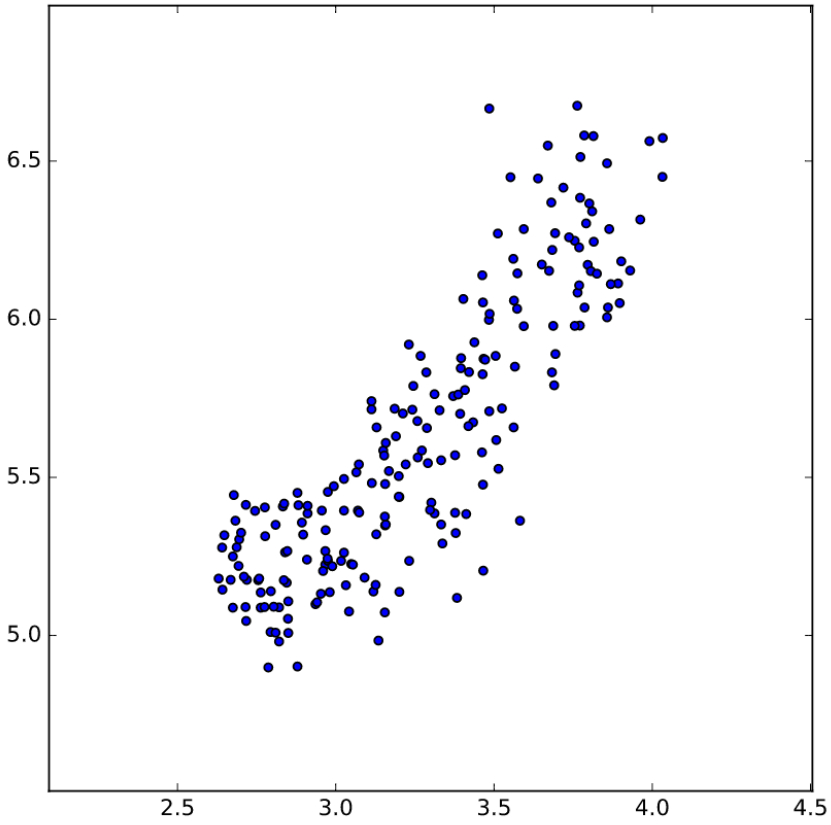
1)

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)
```

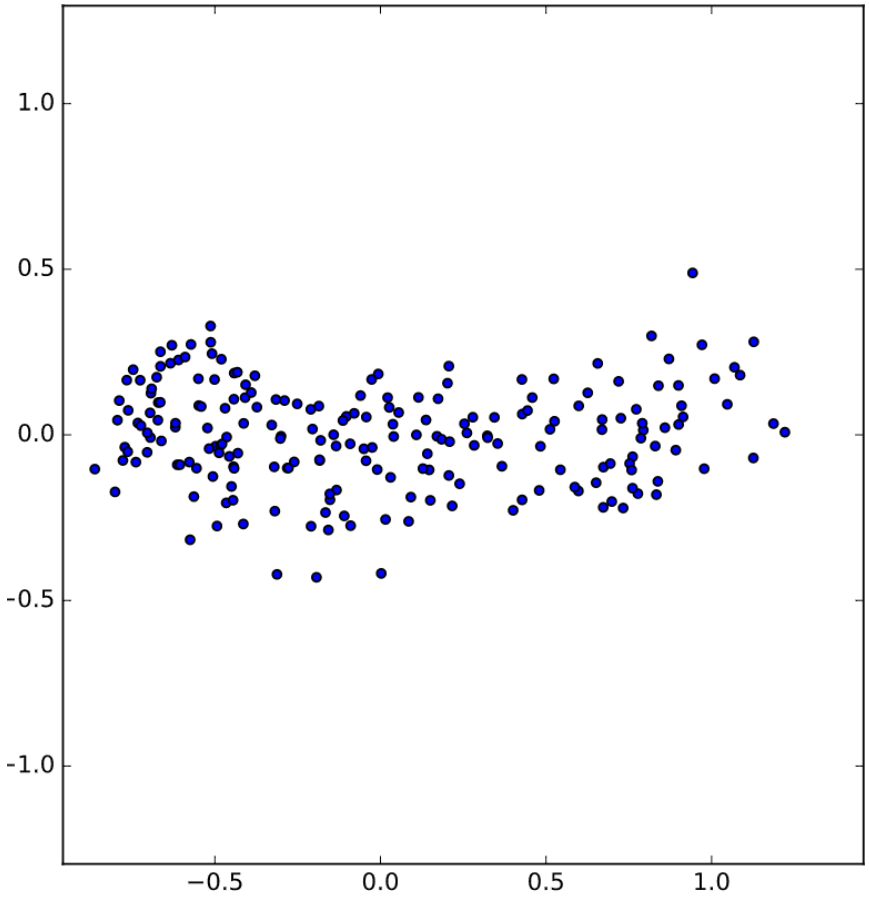


2)

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.



3)

Using custom trainControl

Now that you have a custom `trainControl` object, it's easy to fit `caret`models that use AUC rather than accuracy to tune and evaluate the model. You can just pass your custom `trainControl` object to the `train()` function via the `trControl` argument, e.g.:

```
train(<standard arguments here>, trControl = myControl)
```

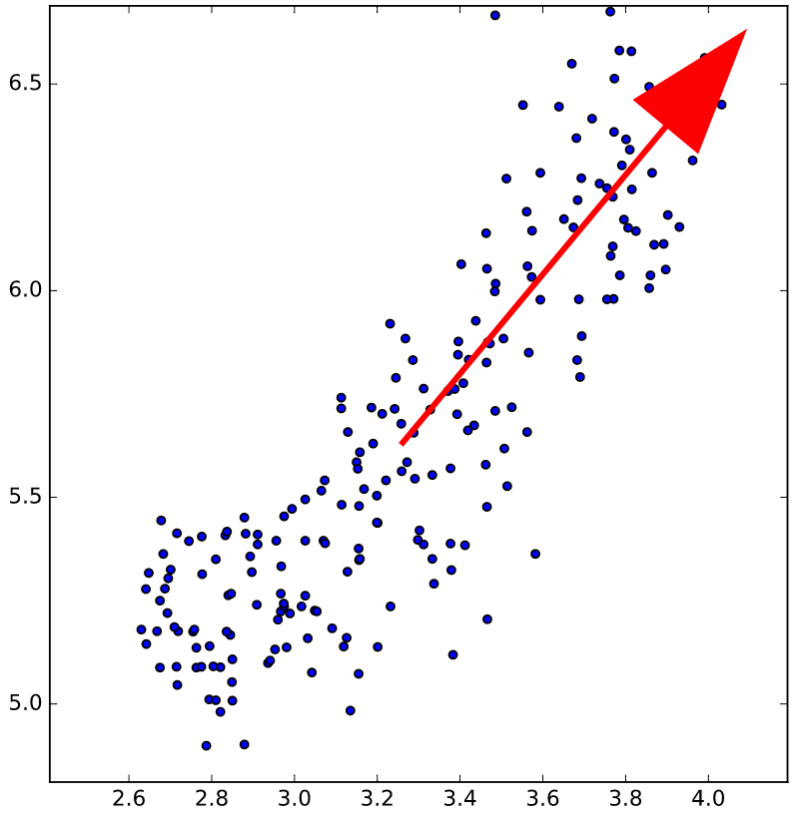
This syntax gives you a convenient way to store a lot of custom modeling parameters and then use them across multiple different calls to `train()`. You will make extensive use of this trick in Chapter 5.

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.

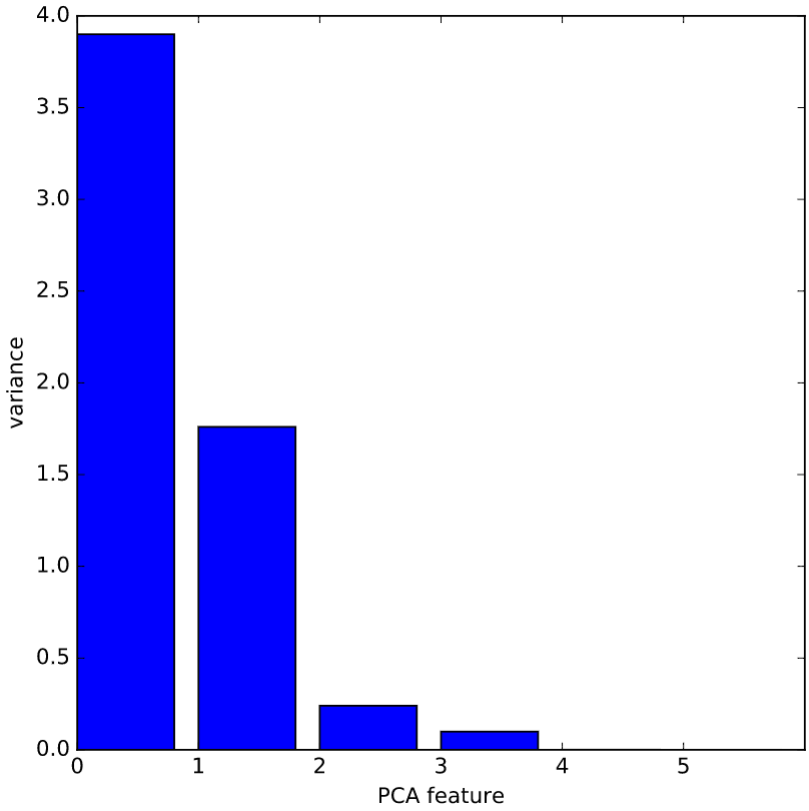
```
# Make a scatter plot of the untransformed points
plt.scatter(grains[:,0], grains[:,1])

# 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()
```



4)
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.

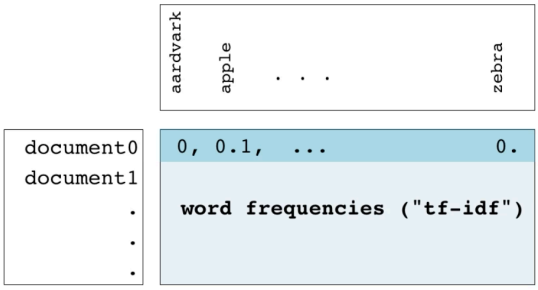
```
# Perform the necessary imports
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
import matplotlib.pyplot as plt

# 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()
```



Note)

Sparse arrays and csr_matrix

- Array is "sparse": most entries are zero
- Can use **scipy.sparse.csr_matrix** instead of NumPy array
- **csr_matrix** remembers only the non-zero entries (saves space!)



5)

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 model 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)
```

```
(85, 2)
```

6)

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())
```

```
[[ 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.        ]]
```

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

# Print words
print(words)
```

```
['cats', 'chase', 'dogs', 'meow', 'say', 'woof']
```

7)

Clustering Wikipedia part I

You saw in the video 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)
```

8)
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'))
```

	article	label
49	Lymphoma	0
47	Fever	0
46	Prednisone	0
45	Hepatitis C	0
44	Gout	0
43	Leukemia	0
42	Doxycycline	0
41	Hepatitis B	0
40	Tonsillitis	0
48	Gabapentin	0
39	Franck Ribéry	1
31	Cristiano Ronaldo	1
32	Arsenal F.C.	1
33	Radamel Falcao	1
34	Zlatan Ibrahimović	1
35	Colombia national football team	1
36	2014 FIFA World Cup qualification	1
37	Football	1
30	France national football team	1
38	Neymar	1
19	2007 United Nations Climate Change Conference	2
17	Greenhouse gas emissions by the United States	2
16	350.org	2
15	Kyoto Protocol	2
14	Climate change	2
13	Connie Hedegaard	2
12	Nigel Lawson	2
11	Nationally Appropriate Mitigation Action	2
10	Global warming	2
18	2010 United Nations Climate Change Conference	2
57	Red Hot Chili Peppers	3
56	Skrillex	3
55	Black Sabbath	3
54	Arctic Monkeys	3
50	Chad Kroeger	3
52	The Wanted	3
51	Nate Ruess	3
58	Sepsis	3
53	Stevie Nicks	3
59	Adam Levine	3
29	Jennifer Aniston	4
27	Dakota Fanning	4
26	Mila Kunis	4
25	Russell Crowe	4
24	Jessica Biel	4
23	Catherine Zeta-Jones	4

22	Denzel Washington	4
21	Michael Fassbender	4
20	Angelina Jolie	4
28	Anne Hathaway	4
1	Alexa Internet	5
2	Internet Explorer	5
3	HTTP cookie	5
4	Google Search	5
8	Firefox	5
6	Hypertext Transfer Protocol	5
7	Social search	5
9	LinkedIn	5
5	Tumblr	5
0	HTTP 404	5

Chapter 4)

Note)

Non-negative matrix factorization

- NMF = "non-negative matrix factorization"
- Dimension reduction technique
- NMF models are *interpretable* (unlike PCA)
- Easy to interpret means easy to explain!
- However, all sample features must be non-negative (>= 0)

1)

NMF applied to Wikipedia articles

In the video, you saw NMF 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)
[[ 0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
     0.00000000e+00   4.40520646e-01]
   [ 0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
     0.00000000e+00   5.66676272e-01]
    ....
```

2)

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. In the next video, you'll see 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 (df.head())
# Print the row for 'Anne Hathaway'
print(df.loc[['Anne Hathaway'],:])
                    0    1    2         3    4    5
  Anne Hathaway  0.003845  0.0  0.0  0.575711  0.0  0.0
# Print the row for 'Denzel Washington'
print(df.loc[['Denzel Washington'],:])
                      0         1    2         3    4    5
  Denzel Washington  0.0  0.005601  0.0  0.42238  0.0  0.0
```

3)

NMF learns topics of documents

In the video, you learned 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)
```

```
(6, 13125)
```

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

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

```
film       0.627877
award      0.253131
starred    0.245284
role       0.211451
actress    0.186398
Name: 3, dtype: float64
```

4)

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.

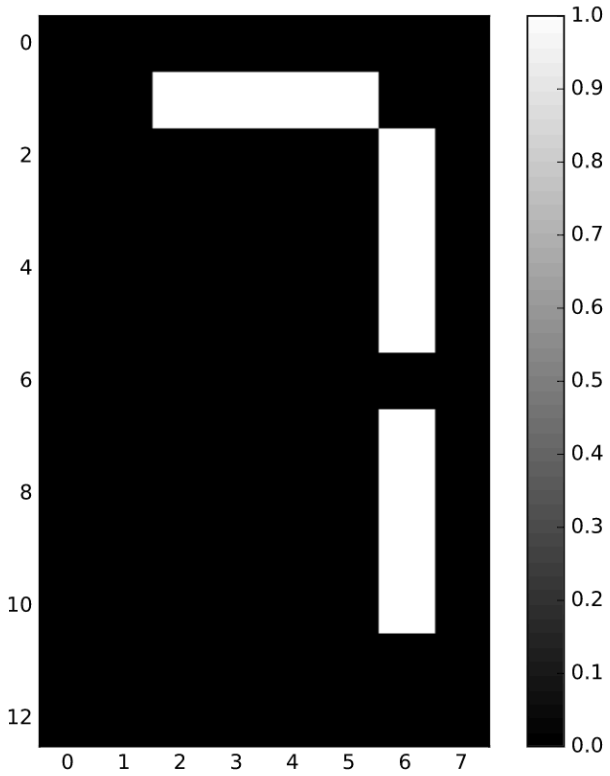
```
# Import pyplot
from matplotlib import pyplot as plt

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

# Print digit
print(digit)
```

```
 [ 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.]
```

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

# Print bitmap
print(bitmap)
```

```
 [[ 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.]]
```

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



5)

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:

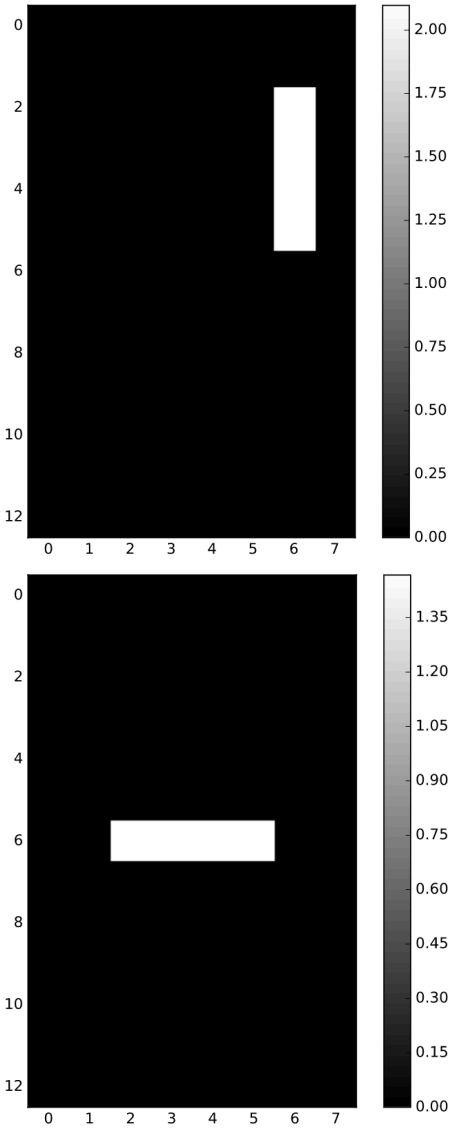
```
def show_as_image(sample):
    bitmap = sample.reshape((13, 8))
    plt.figure()
    plt.imshow(bitmap, cmap='gray', interpolation='nearest')
    plt.colorbar()
    plt.show()
```

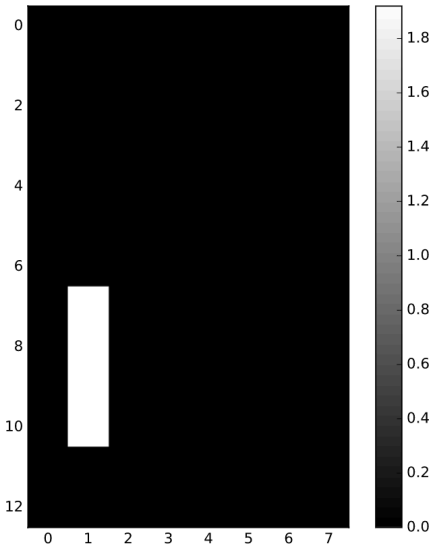
After you are done, take a moment to look through the plots and notice how NMF has expressed the digit as a sum of the components!

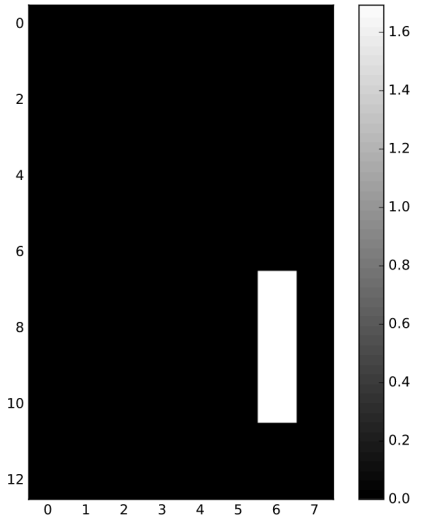
```
# Import NMF
from sklearn.decomposition import NMF

# 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
```

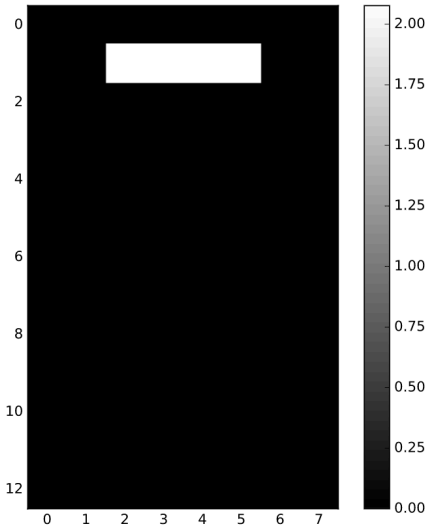
```
for component in model.components_:
    show_as_image(component)

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

# Print digit_features
print(digit_features)
```

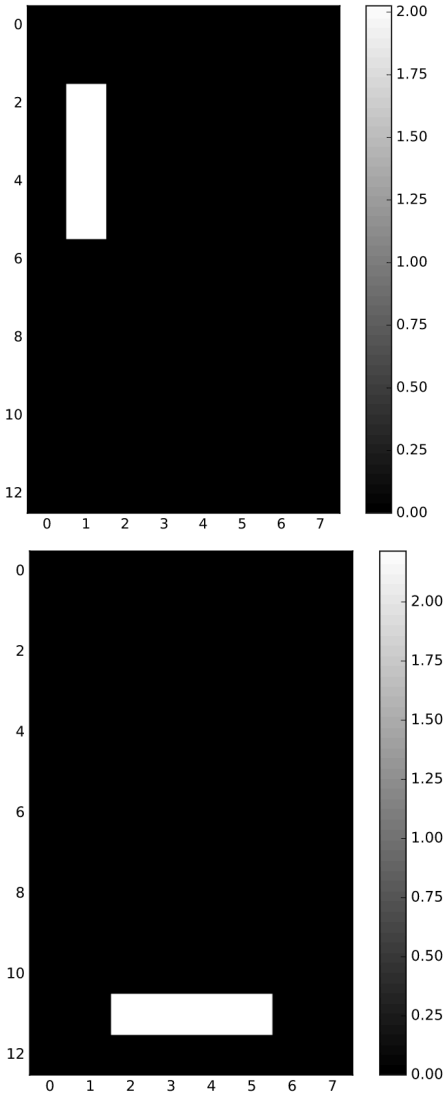
```
[ 4.76823559e-01  0.00000000e+00  0.00000000e+00  5.90605054e-01
  4.81559442e-01  0.00000000e+00  7.37568241e-16]
```




0
2
4
6
8
10
12
0 1 2 3 4 5 6 7
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0


0
2
4
6
8
10
12
0 1 2 3 4 5 6 7
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0


0
2
4
6
8
10
12
0 1 2 3 4 5 6 7
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00



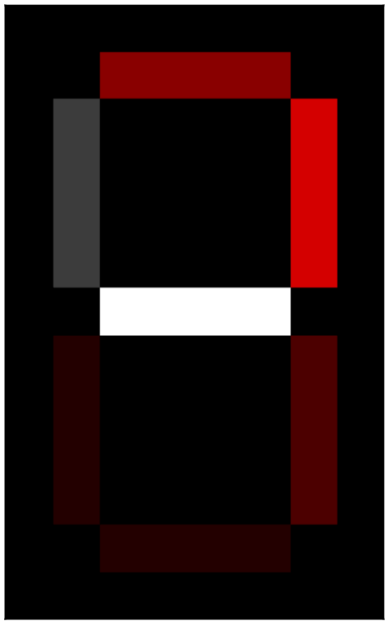
6)
PCA doesn't learn parts

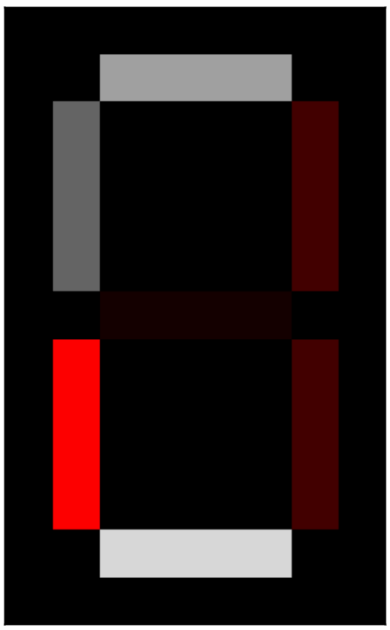
Unlike NMF, PCA *doesn't* learn the parts of things. Its components do not correspond to topics (in the case of documents) or to parts of images, when trained on images. Verify this for yourself by inspecting the components of a PCA model fit to the dataset of LED digit images from the previous exercise. 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!

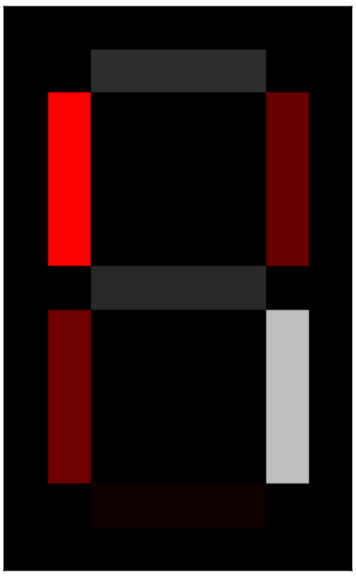
# Import PCA
from sklearn.decomposition import PCA

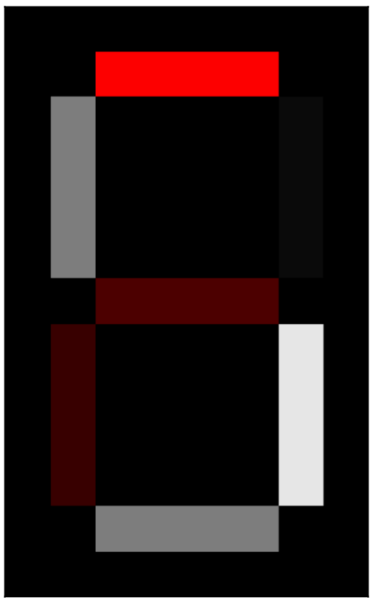
Create a PCA instance: model

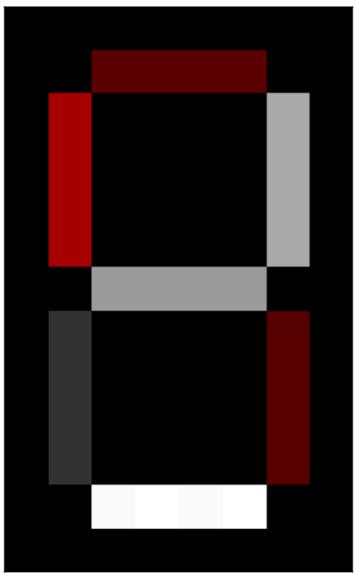
```
model = PCA(n_components=7)

# 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)
```

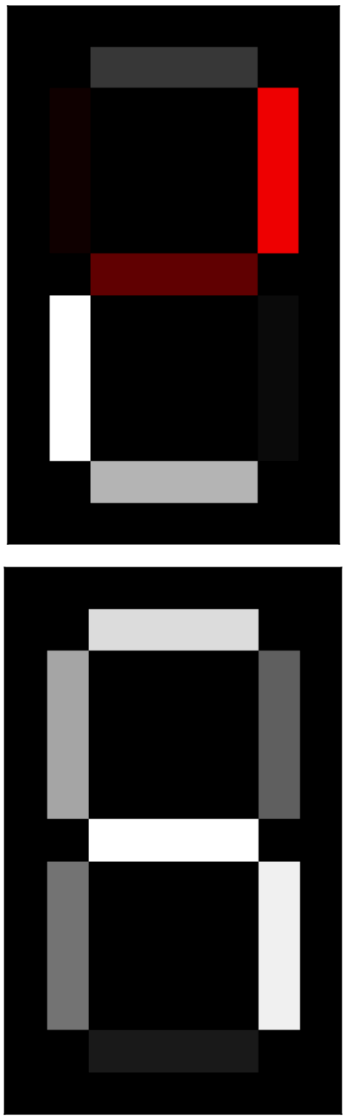












7)

Which articles are similar to 'Cristiano Ronaldo'?

In the video, you learned 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)
```

```
# 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())
```

```
Cristiano Ronaldo              1.000000
Franck Ribéry                  0.999972
Radamel Falcao                 0.999942
Zlatan Ibrahimović             0.999942
France national football team  0.999923
dtype: float64
```

8)

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 column 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!
This data is part of a larger dataset available **here**.

```
# 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)
```

9)
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())
```

```
Bruce Springsteen    1.000000
Neil Young           0.958064
Leonard Cohen        0.916500
Van Morrison         0.871899
Bob Dylan            0.862235
dtype: float64
```