UNSUPERVISED LEARNING IN PYTHON

Unsupervised learning

Unsupervised learning

- Unsupervised learning finds patterns in data
- E.g. *clustering* customers by their purchases
- Compressing the data using purchase patterns (*dimension reduction*)

Supervised vs unsupervised learning

- *Supervised* learning finds patterns for a prediction task
- E.g. classify tumors as benign or cancerous (*labels*)
- Unsupervised learning finds patterns in data
- … but *without* a specific prediction task in mind

Iris dataset

- Measurements of many iris plants
- 3 species of iris: setosa, versicolor, virginica
- Petal length, petal width, sepal length, sepal width (the *features* of the dataset)

Source: http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html

Arrays, features & samples

- 2D NumPy array
- Columns are measurements (the *features*)
- Rows represent iris plants (the *samples*)

Iris data is 4-dimensional

- Iris samples are points in 4 dimensional space
- Dimension = number of features
- Dimension too high to visualize!
- ... but unsupervised learning gives insight

k-means clustering

- Finds clusters of samples
- Number of clusters must be specified
- Implemented in **sklearn** ("scikit-learn")

k-means clustering with scikit-learn

```
In [1]: print(samples)
[[ 5.   3.3  1.4  0.2]
 [ 5.   3.5  1.3  0.3]
 [ 4.9  2.4  3.3  1. ]
 [ 6.3  2.8  5.1  1.5]
 ...
 [ 7.2  3.2  6.   1.8]]

In [2]: from sklearn.cluster import KMeans

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

In [4]: model.fit(samples)
Out[4]: KMeans(algorithm='auto', ...)

In [5]: labels = model.predict(samples)

In [6]: print(labels)
[0 0 1 1 0 1 2 1 0 1 ...]
```

Cluster labels for new samples

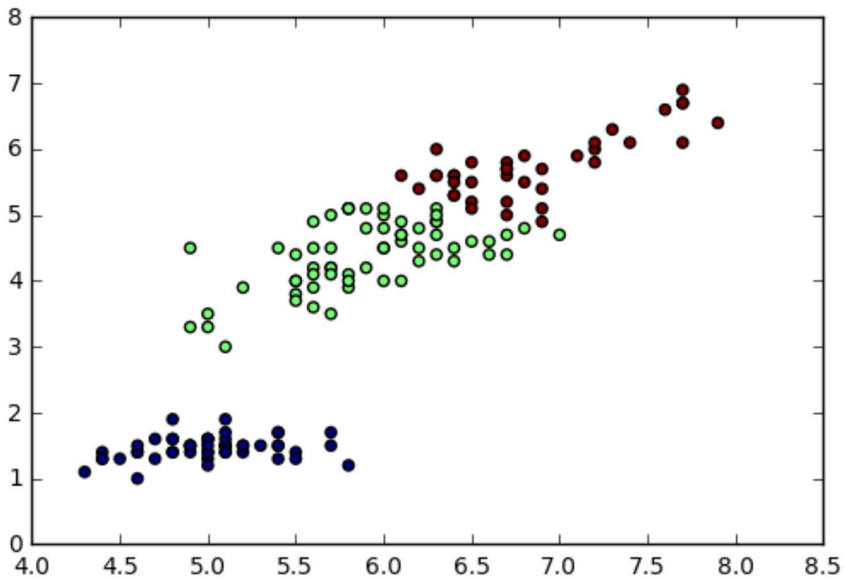
- New samples can be assigned to existing clusters
- k-means remembers the mean of each cluster (the "centroids")
- Finds the nearest centroid to each new sample

Cluster labels for new samples

```
In [7]: 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]]

In [8]: new_labels = model.predict(new_samples)

In [9]: print(new_labels)
[0 2 1]
```

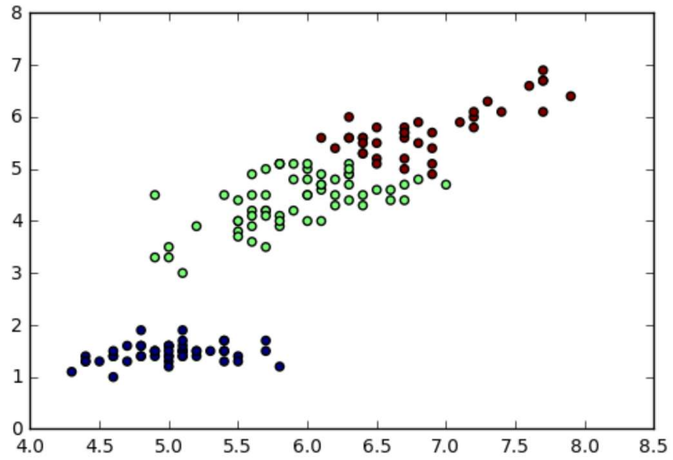
Scatter plots

- Scatter plot of sepal length vs petal length
- Each point represents an iris sample
- Color points by cluster labels
- PyPlot (**matplotlib.pyplot**)



Scatter plots

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

In [2]: xs = samples[:,0]

In [3]: ys = samples[:,2]

In [4]: plt.scatter(xs, ys, c=labels)

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



UNSUPERVISED LEARNING IN PYTHON

Evaluating a clustering

Evaluating a clustering

- Can check correspondence with e.g. iris species
- ... but what if there are no species to check against?
- Measure quality of a clustering
- Informs choice of how many clusters to look for

Iris: clusters vs species

- k-means found 3 clusters amongst the iris samples
- Do the clusters correspond to the species?

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

Cross tabulation with pandas

- Clusters vs species is a "cross-tabulation"
- Use the **pandas** library
- Given the species of each sample as a list **species**

```
In [1]: print(species)
['setosa', 'setosa', 'versicolor', 'virginica', ... ]
```

Aligning labels and species

```
In [2]: import pandas as pd

In [3]: df = pd.DataFrame({'labels': labels, 'species': species})

In [4]: print(df)
     labels     species
0         1      setosa
1         1      setosa
2         2  versicolor
3         2   virginica
4         1      setosa
...
```

Crosstab of labels and species

```
In [5]: ct = pd.crosstab(df['labels'], df['species'])

In [6]: print(ct)
species  setosa  versicolor  virginica
labels
0             0           2         36
1            50           0          0
2             0          48         14
```

How to evaluate a clustering, if there were no species information?

Measuring clustering quality

- Using only samples and their cluster labels
- A good clustering has tight clusters
- … and samples in each cluster bunched together

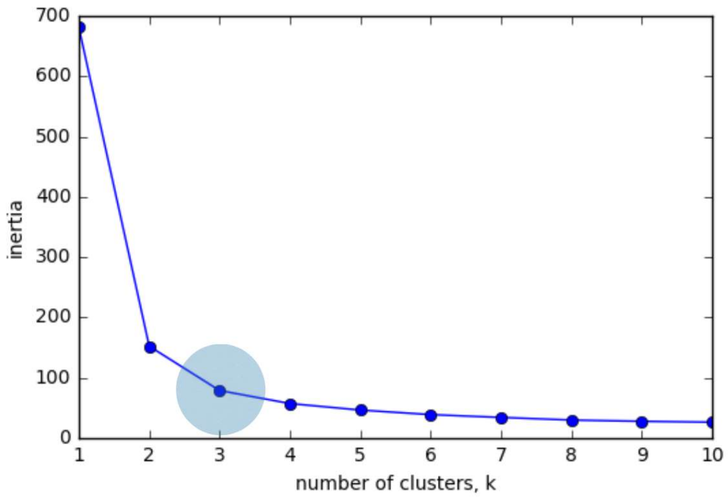
Inertia measures clustering quality

- 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
```

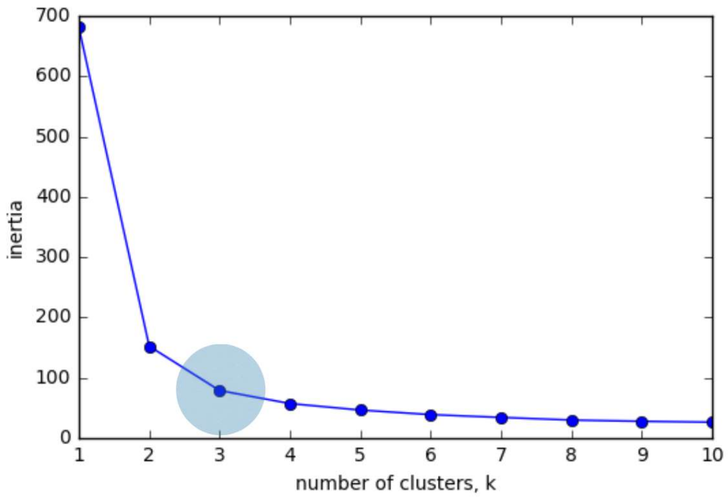
The number of clusters

- Clusterings of the iris dataset with different numbers of clusters
- More clusters means lower inertia
- What is the best number of clusters?



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



UNSUPERVISED LEARNING IN PYTHON

Transforming features for better clusterings

Piedmont wines dataset

- 178 samples from 3 distinct varieties of red wine: *Barolo*, *Grignolino* and *Barbera*
- Features measure chemical composition e.g. alcohol content
- ... also visual properties like “color intensity”

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

Clustering the wines

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

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

In [3]: labels = model.fit_predict(samples)
```

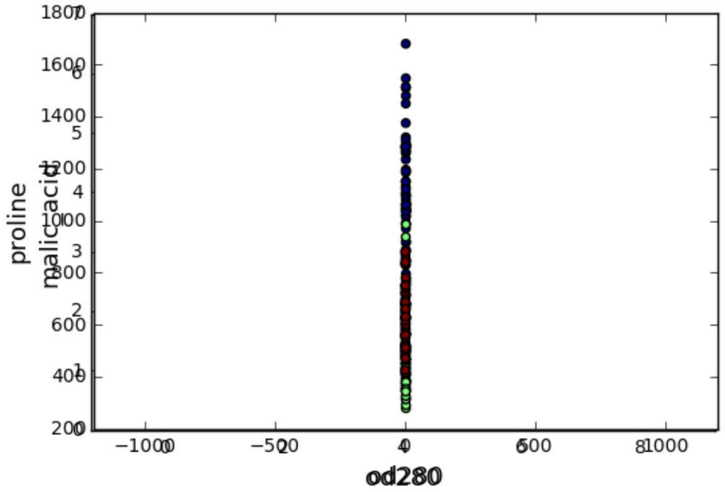
Clusters vs. varieties

```
In [4]: df = pd.DataFrame({'labels': labels,
   ...:                    'varieties': varieties})

In [5]: ct = pd.crosstab(df['labels'], df['varieties'])

In [6]: print(ct)
varieties  Barbera  Barolo  Grignolino
labels
0               29      13          20
1                0      46           1
2               19       0          50
```

Feature variances

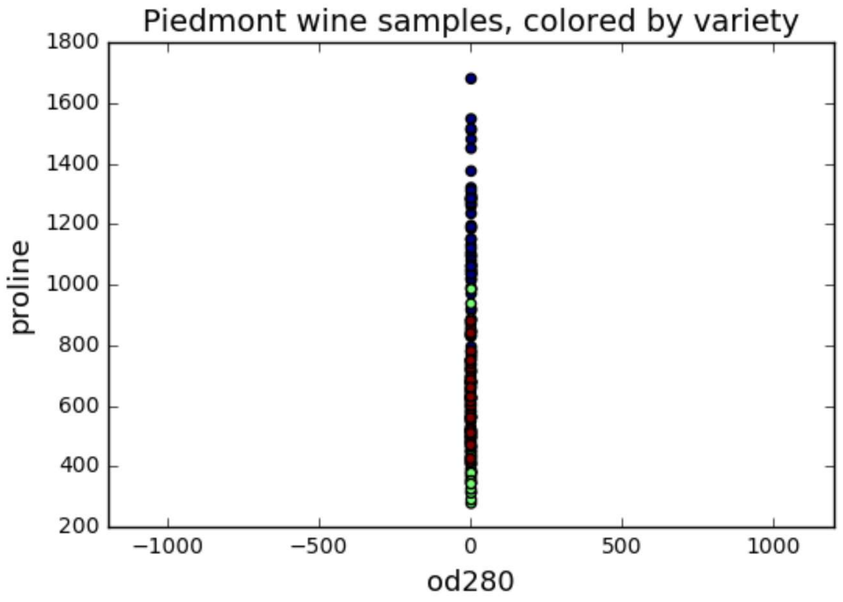
- The wine features have very different variances!
- Variance of a feature measures spread of its values

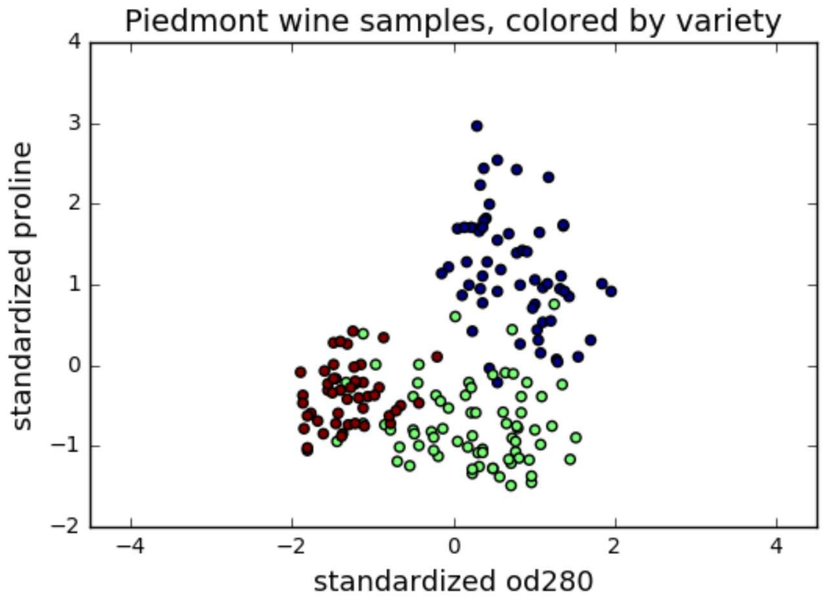
```
feature      variance
alcohol          0.65
malic_acid       1.24
...
od280            0.50
proline      99166.71
```



StandardScaler

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





sklearn StandardScaler

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

In [2]: scaler = StandardScaler()

In [3]: scaler.fit(samples)
Out[3]: StandardScaler(copy=True, with_mean=True, with_std=True)

In [4]: samples_scaled = scaler.transform(samples)
```

Similar methods

- **StandardScaler** and **KMeans** have similar methods
- Use **fit() / transform()** with **StandardScaler**
- Use **fit() / predict()** with **KMeans**

StandardScaler, then KMeans

- Need to perform two steps: **StandardScaler**, then **KMeans**
- Use sklearn pipeline to combine multiple steps
- Data flows from one step into the next

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

UNSUPERVISED LEARNING IN PYTHON

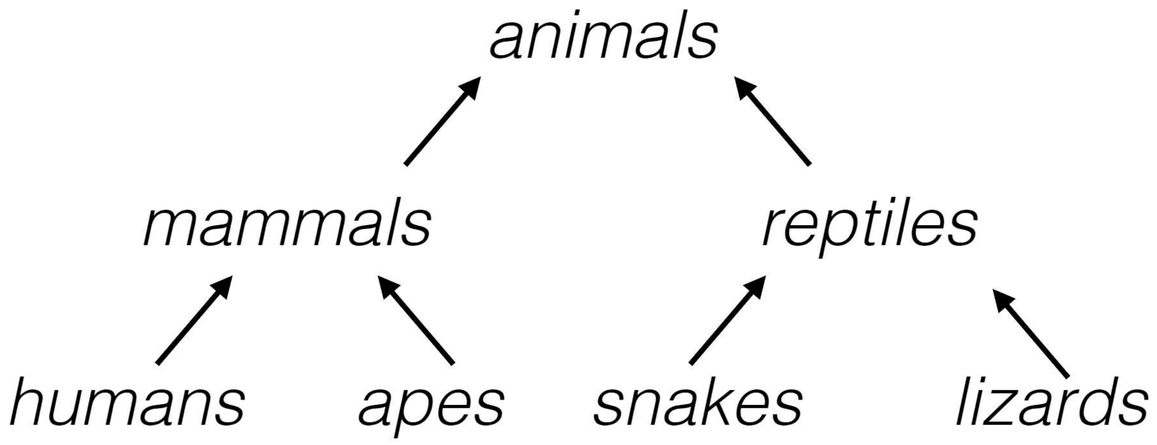
Visualizing hierarchies

Visualisations communicate insight

- "t-SNE" : Creates a 2D map of a dataset (later)
- "Hierarchical clustering" (this video)

A hierarchy of groups

- Groups of living things can form a hierarchy
- Clusters are contained in one another



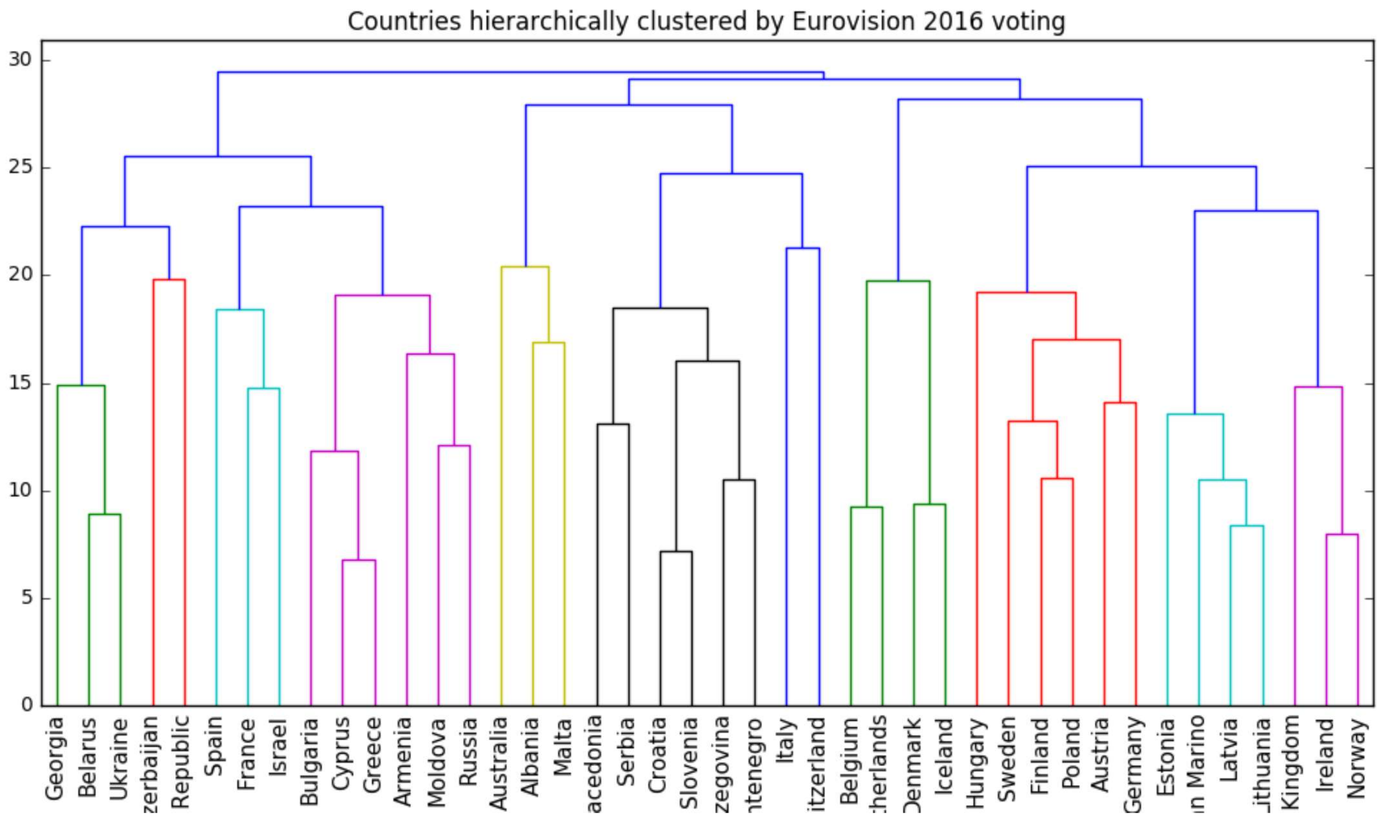
Eurovision scoring dataset

- Countries gave scores to songs performed at the Eurovision 2016
- 2D array of scores
- Rows are countries, columns are songs

	song0	song1	. . .	song25
Albania				
Armenia				
.	0	7	. . .	4
.				
.				
United Kingdom				

Source: http://www.eurovision.tv/page/results

Hierarchical clustering of voting countries

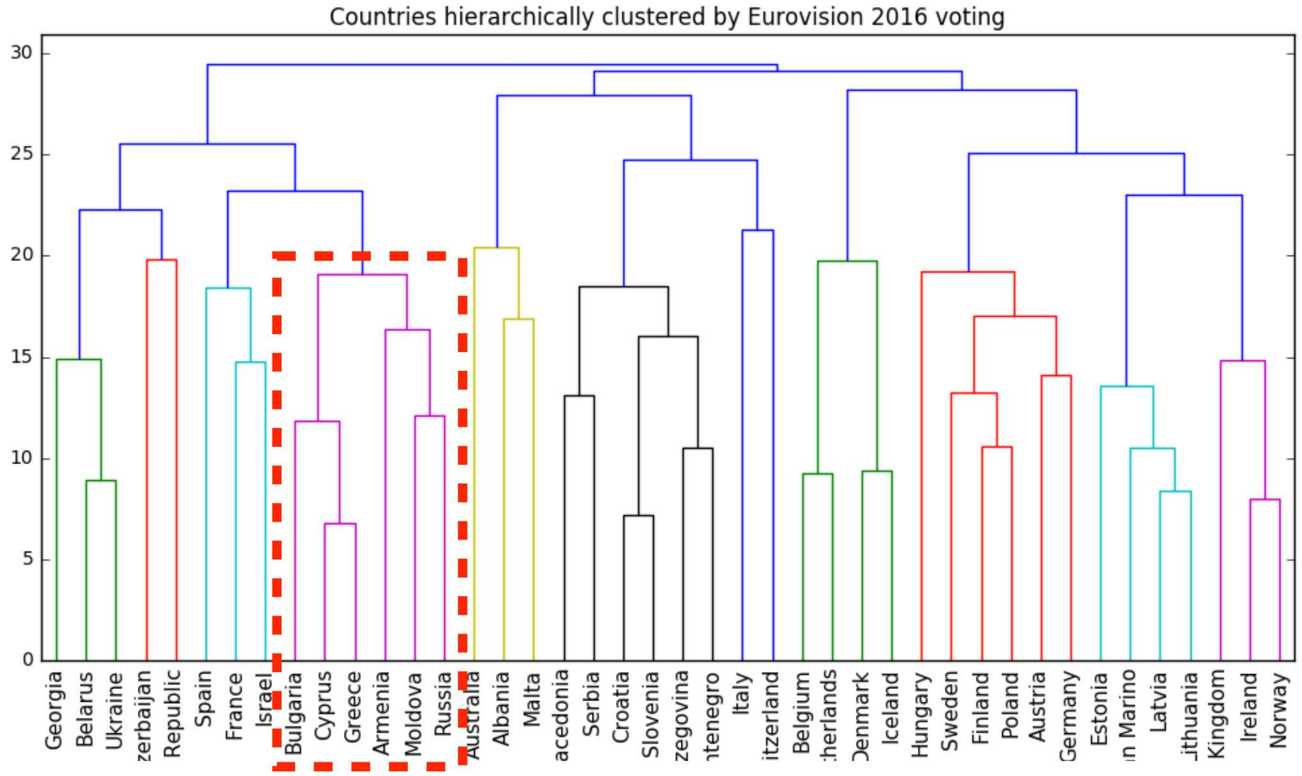


Hierarchical clustering

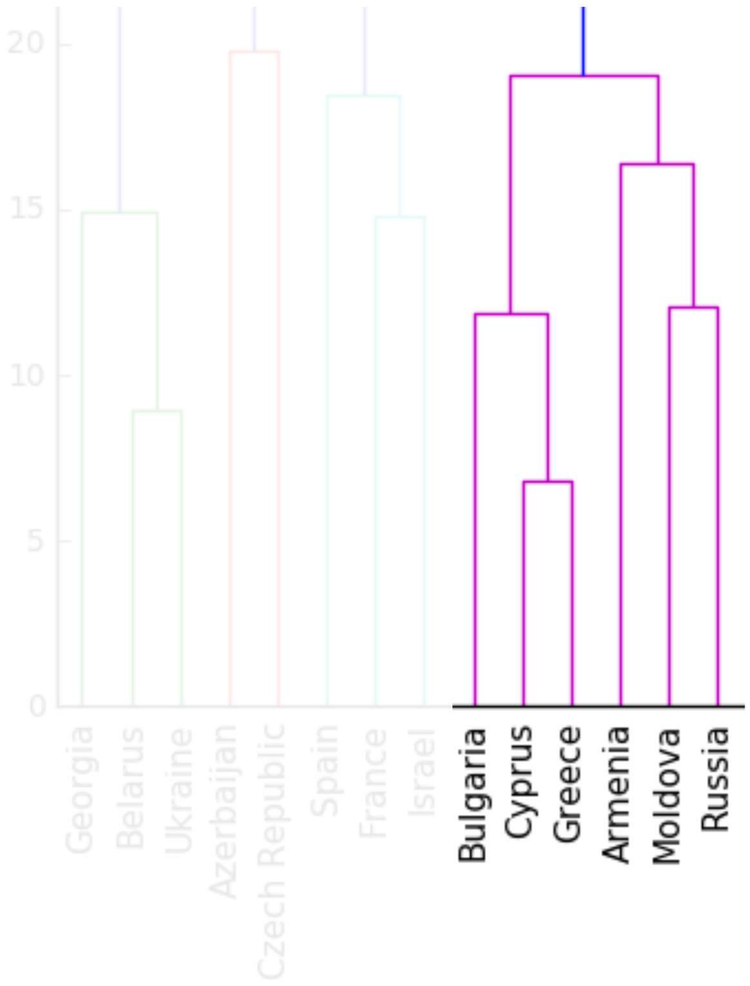
- Every country begins in a separate cluster
- At each step, the two closest clusters are merged
- Continue until all countries in a single cluster
- This is “agglomerative” hierarchical clustering

The dendrogram of a hierarchical clustering

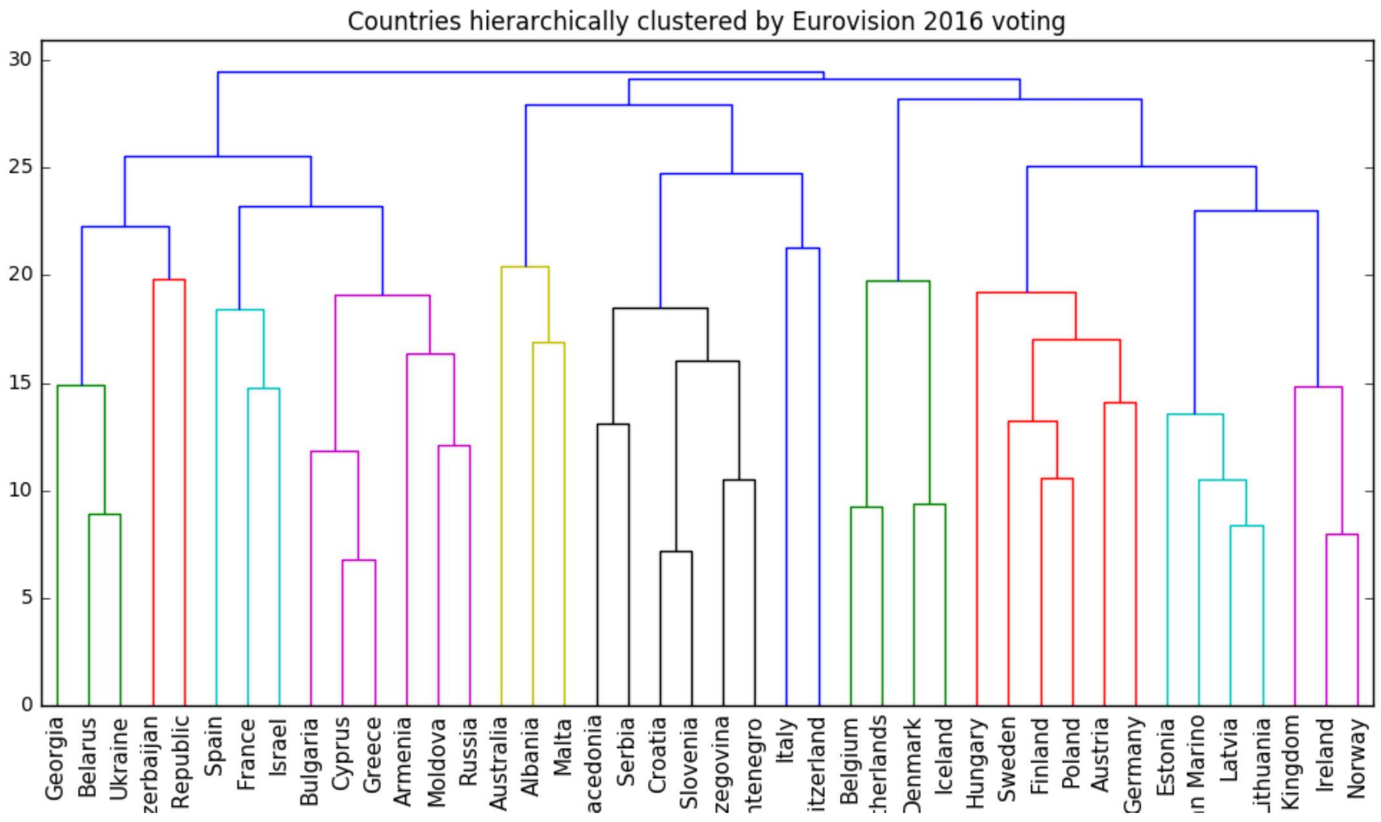
- Read from the bottom up
- Vertical lines represent clusters



Dendrograms, step-by-step



Dendrograms, step-by-step



Hierarchical clustering with SciPy

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

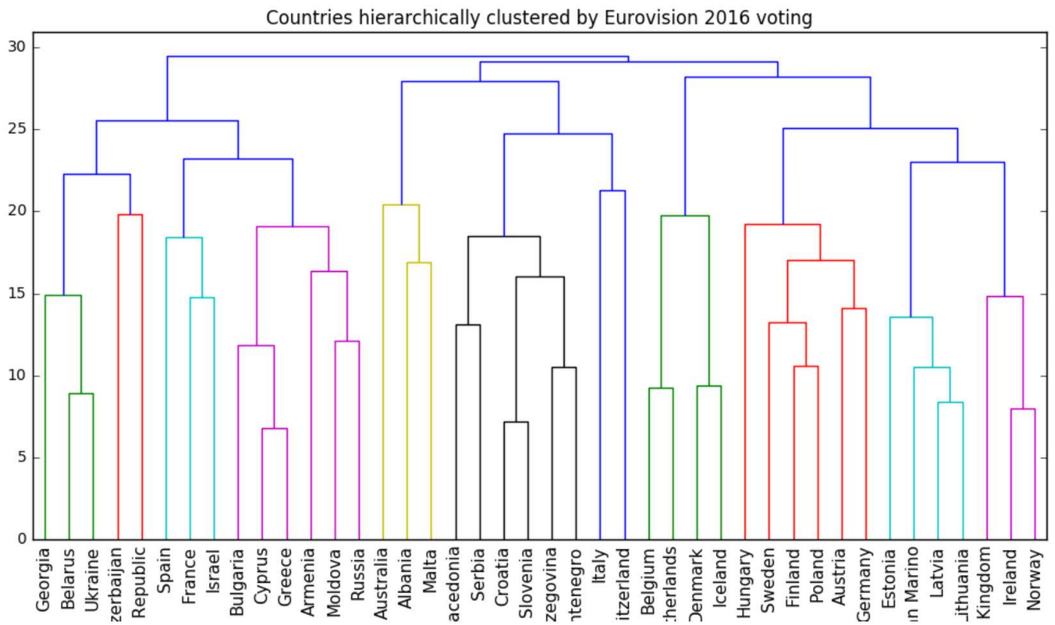
```
In [1]: import matplotlib.pyplot as plt

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()
```

UNSUPERVISED LEARNING IN PYTHON

Cluster labels in hierarchical clustering

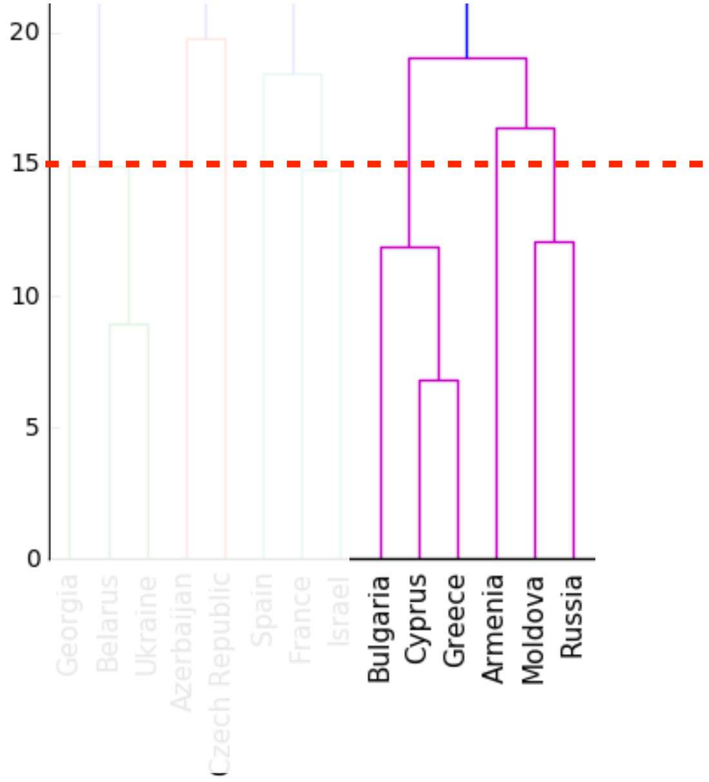
Cluster labels in hierarchical clustering

- Not only a visualisation tool!
- Cluster labels at any intermediate stage can be recovered
- For use in e.g. cross-tabulations



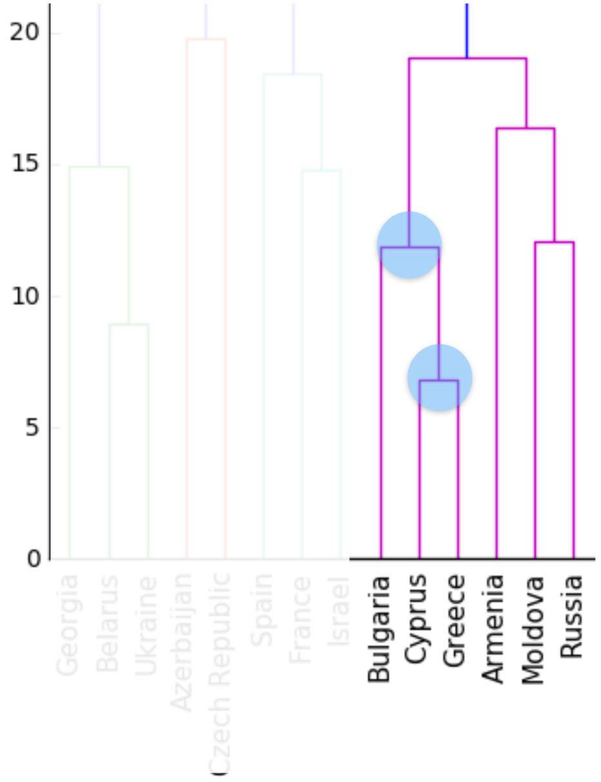
Intermediate clusterings & height on dendrogram

- E.g. at height 15: Bulgaria, Cyprus, Greece are one cluster
- Russia and Moldova are another
- Armenia in a cluster on its own



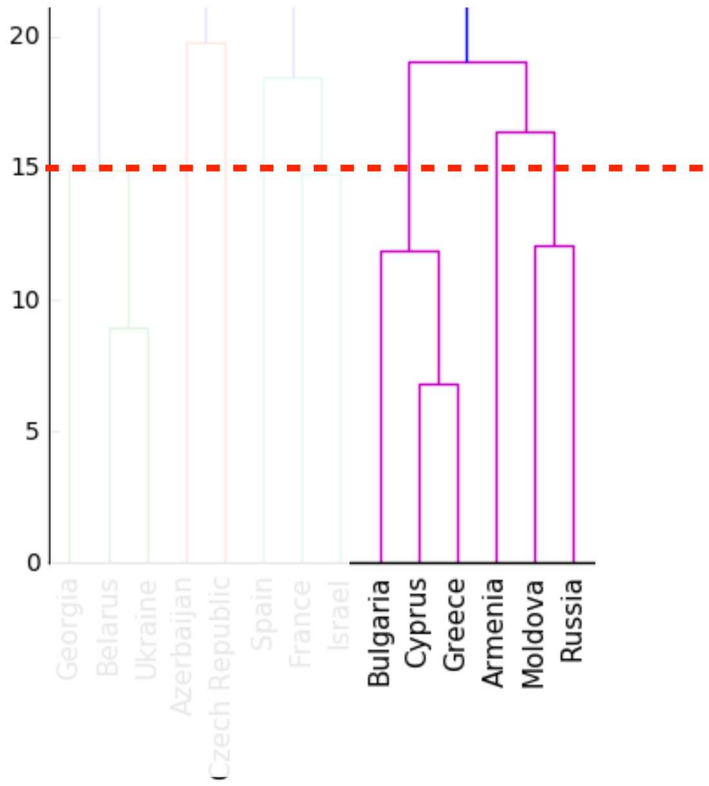
Dendrograms show cluster distances

- 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



Intermediate clusterings & height on dendrogram

- Height on dendrogram specifies max. distance between merging clusters
- Don't merge clusters further apart than this (e.g. 15)

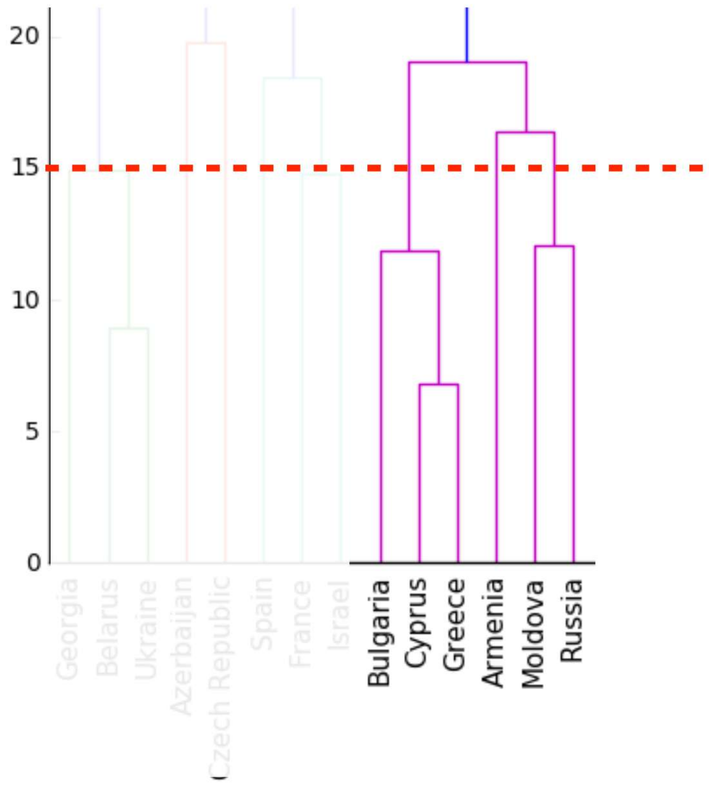


Distance between clusters

- Defined by a "linkage method"
- Specified via method parameter, e.g. **linkage(samples, method="complete")**
- In "complete" linkage: distance between clusters is max. distance between their samples
- Different linkage method, different hierarchical clustering!

Extracting cluster labels

- Use the **fcluster** method
- Returns a NumPy array of cluster labels



Extracting cluster labels using fcluster

```
In [1]: from scipy.cluster.hierarchy import linkage

In [2]: mergings = linkage(samples, method='complete')

In [3]: from scipy.cluster.hierarchy import fcluster

In [4]: labels = fcluster(mergings, 15, criterion='distance')

In [5]: print(labels)
[ 9  8 11 20  2  1 17 14 ... ]
```

Aligning cluster labels with country names

- Given a list of strings **country_names**:

```
In [1]: import pandas as pd

In [2]: pairs = pd.DataFrame({'labels': labels,
   ...:                       'countries': country_names})

In [3]: print(pairs.sort_values('labels'))
              countries  labels
5               Belarus       1
40              Ukraine       1
17              Georgia       1
...
36                Spain       5
8              Bulgaria       6
19               Greece       6
10               Cyprus       6
28              Moldova       7
...
```

UNSUPERVISED LEARNING IN PYTHON

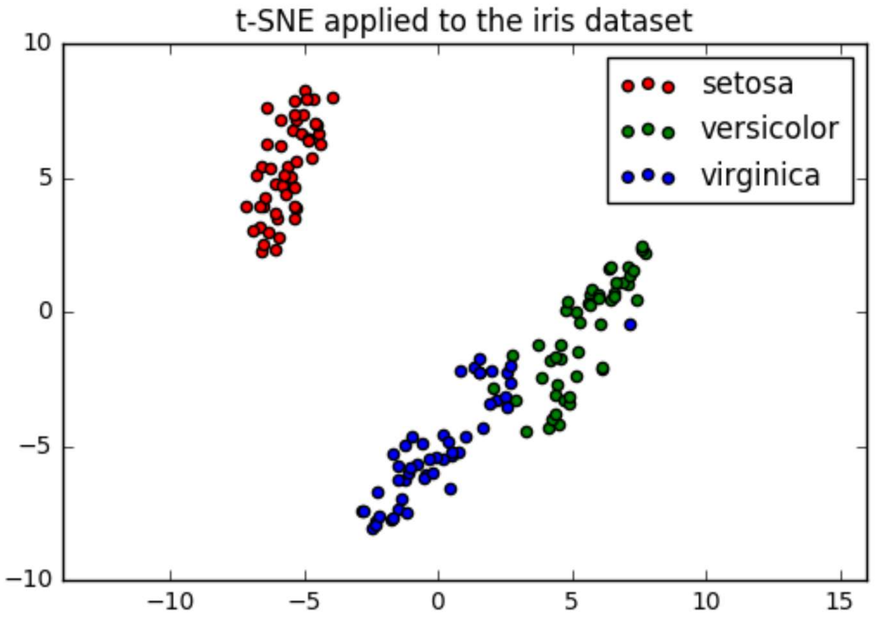
t-SNE for 2-dimensional maps

t-SNE for 2-dimensional maps

- t-SNE = “t-distributed stochastic neighbor embedding”
- Maps samples to 2D space (or 3D)
- Map approximately preserves nearness of samples
- Great for inspecting datasets

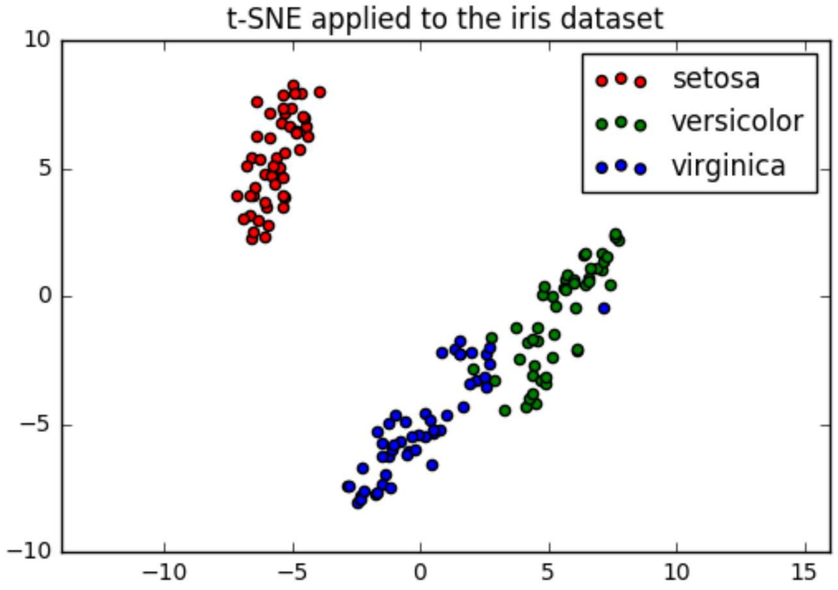
t-SNE on the iris dataset

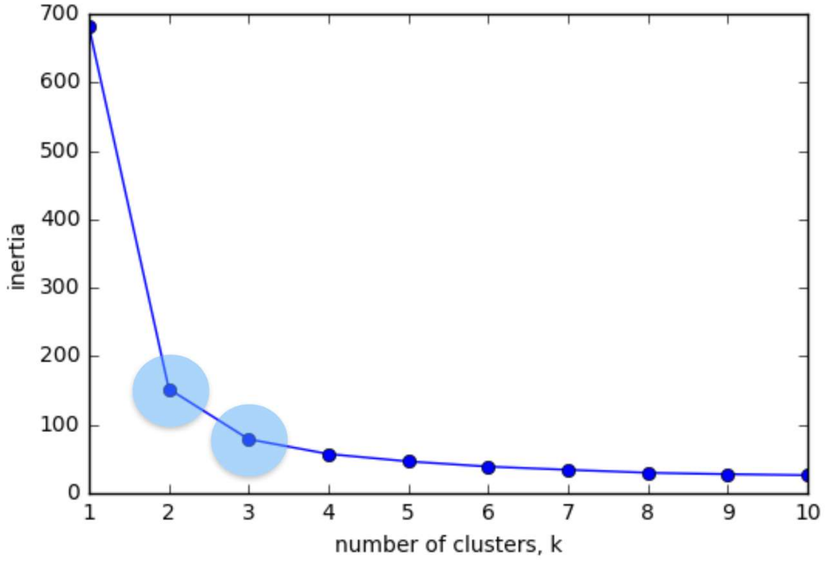
- Iris dataset has 4 measurements, so samples are 4-dimensional
- t-SNE maps samples to 2D space
- t-SNE didn't know that there were different species
- ... yet kept the species mostly separate



Interpreting t-SNE scatter plots

- "versicolor" and "virginica" harder to distinguish from one another
- Consistent with k-means inertia plot: could argue for 2 clusters, or for 3





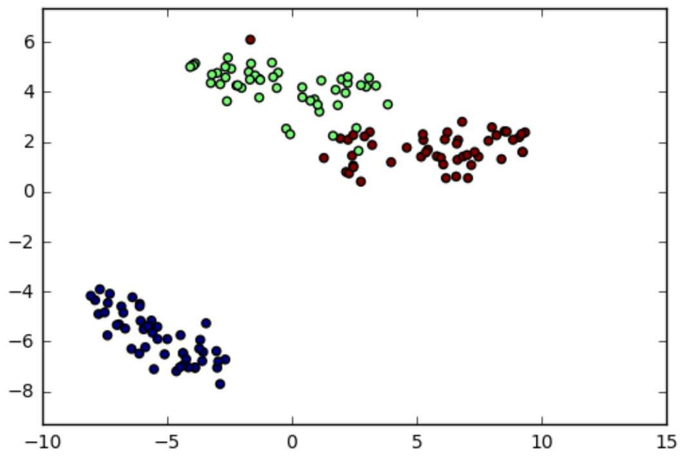
t-SNE in sklearn

- 2D NumPy array **samples**
- List **species** giving species of labels as number (0, 1, or 2)

```
In [1]: print(samples)
[[ 5.   3.3  1.4  0.2]
 [ 5.   3.5  1.3  0.3]
 [ 4.9  2.4  3.3  1. ]
 [ 6.3  2.8  5.1  1.5]
 ...
 [ 4.9  3.1  1.5  0.1]]

In [2]: print(species)
[0, 0, 1, 2, ..., 0]
```

t-SNE in sklearn

```
In [3]: import matplotlib.pyplot as plt

In [4]: from sklearn.manifold import TSNE

In [5]: model = TSNE(learning_rate=100)

In [5]: transformed = model.fit_transform(samples)

In [6]: xs = transformed[:,0]

In [7]: ys = transformed[:,1]

In [8]: plt.scatter(xs, ys, c=species)

In [9]: plt.show()
```



t-SNE has only fit_transform()

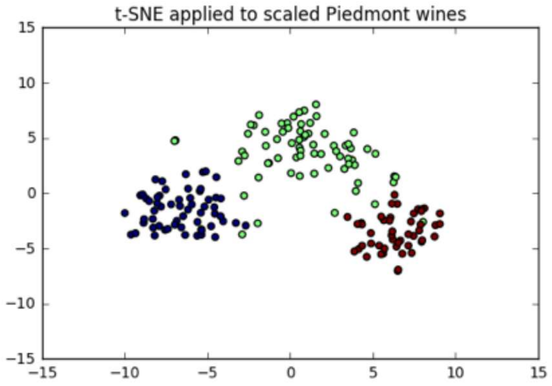
- Has a **fit_transform()** method
- Simultaneously fits the model and transforms the data
- Has no separate **fit()** or **transform()** methods
- Can't extend the map to include new data samples
- Must start over each time!

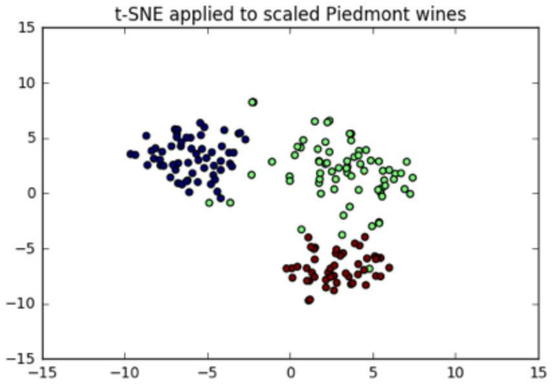
t-SNE learning rate

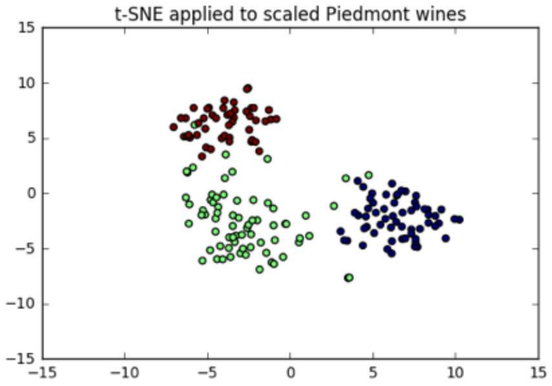
- Choose learning rate for the dataset
- Wrong choice: points bunch together
- Try values between 50 and 200

Different every time

- t-SNE features are different every time
- Piedmont wines, 3 runs, 3 different scatter plots!
- ... however: The wine varieties (=colors) have same position relative to one another







UNSUPERVISED LEARNING IN PYTHON

Visualizing the PCA transformation

Dimension reduction

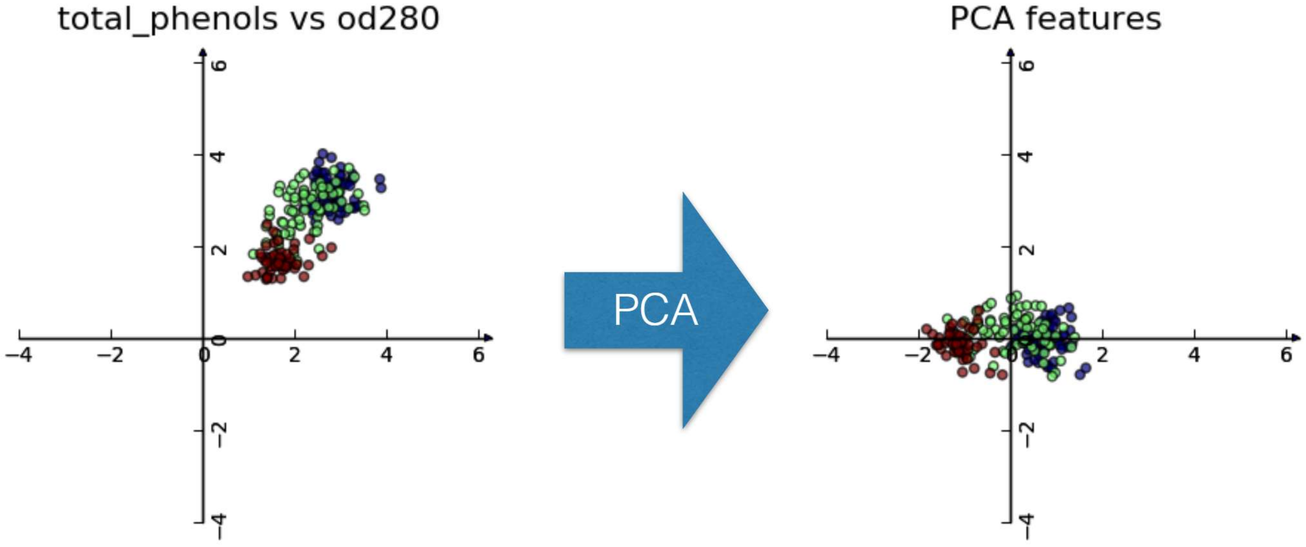
- More efficient storage and computation
- Remove less-informative "noise" features
- ... which cause problems for prediction tasks, e.g. classification, regression

Principal Component Analysis

- PCA = "Principal Component Analysis"
- Fundamental dimension reduction technique
- First step "decorrelation" (considered here)
- Second step reduces dimension (considered later)

PCA aligns data with axes

- Rotates data samples to be aligned with axes
- Shifts data samples so they have mean 0
- No information is lost



PCA follows the fit/transform pattern

- **PCA** a scikit-learn component like **KMeans** or **StandardScaler**
- **fit()** learns the transformation from given data
- **transform()** applies the learned transformation
- **transform()** can also be applied to new data

Using scikit-learn PCA

- **samples** = array of two wine features (total_phenols & od280)

```
In [1]: print(samples)
[[ 2.8   3.92]
 [ 2.65  3.4 ]
 ...
 [ 2.05  1.6 ]]

In [2]: from sklearn.decomposition import PCA

In [3]: model = PCA()

In [4]: model.fit(samples)
Out[4]: PCA(copy=True, ...)

In [5]: transformed = model.transform(samples)
```

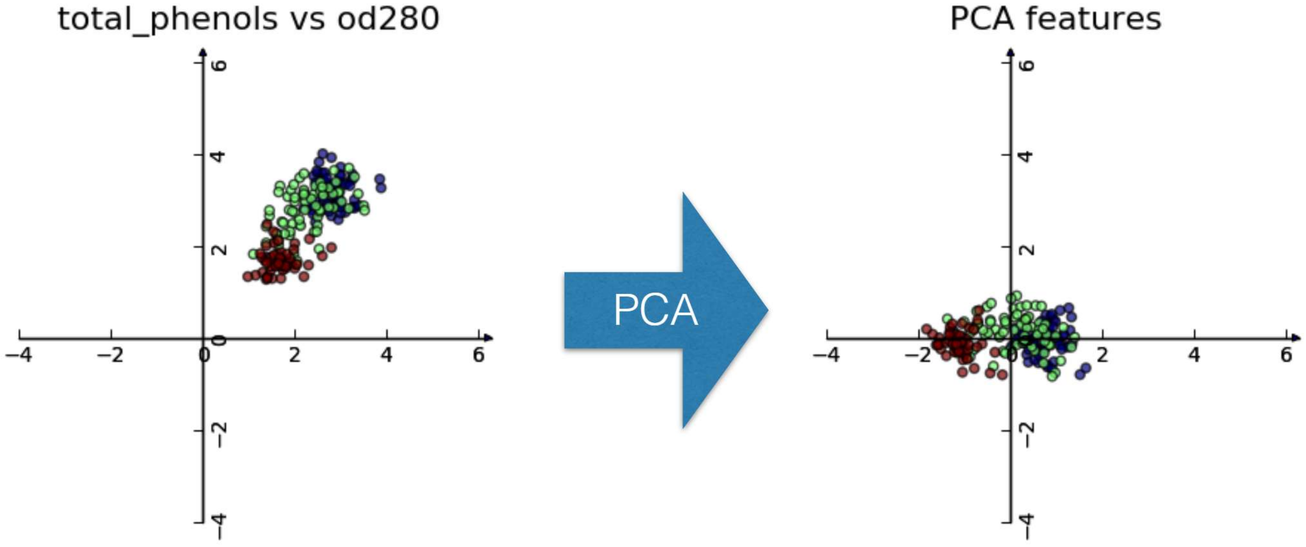
PCA features

- Rows of **transformed** correspond to samples
- Columns of **transformed** are the "PCA features"
- Row gives PCA feature values of corresponding sample

```
In [6]: print(transformed)
[[  1.32771994e+00   4.51396070e-01]
 [  8.32496068e-01   2.33099664e-01]
 ...
 [ -9.33526935e-01  -4.60559297e-01]]
```

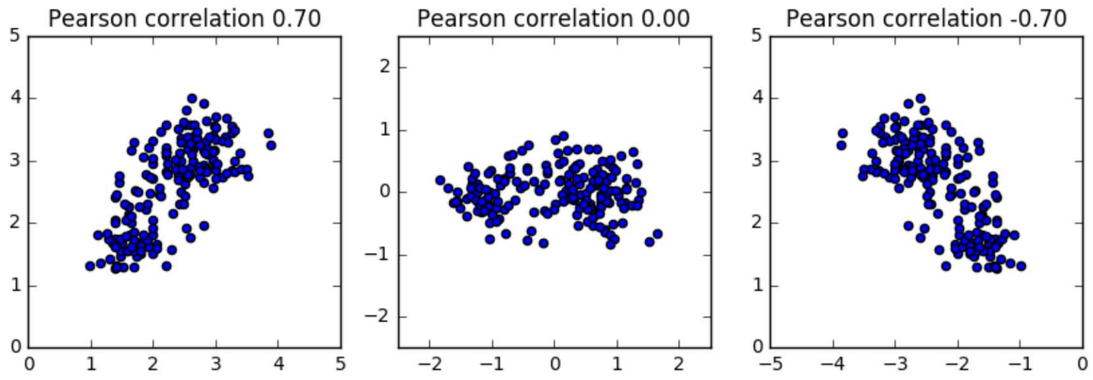
PCA features are not correlated

- Features of dataset are often correlated, e.g. total_phenols and od280
- PCA aligns the data with axes
- Resulting PCA features are not linearly correlated ("decorrelation")



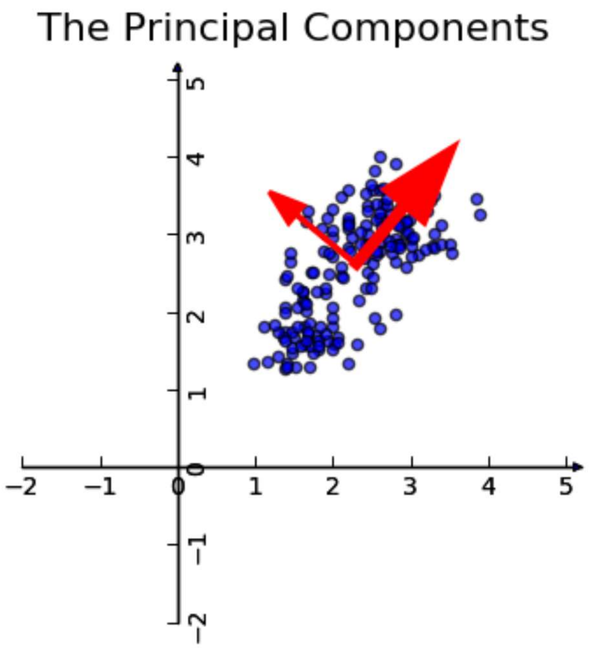
Pearson correlation

- Measures linear correlation of features
- Value between -1 and 1
- Value of 0 means no linear correlation



Principal components

- "Principal components" = directions of variance
- PCA aligns principal components with the axes
- Available as **components_** attribute of PCA object
- Each row defines displacement from mean




```
In [7]: print(model.components_)
[[ 0.64116665  0.76740167]
 [-0.76740167  0.64116665]]
```

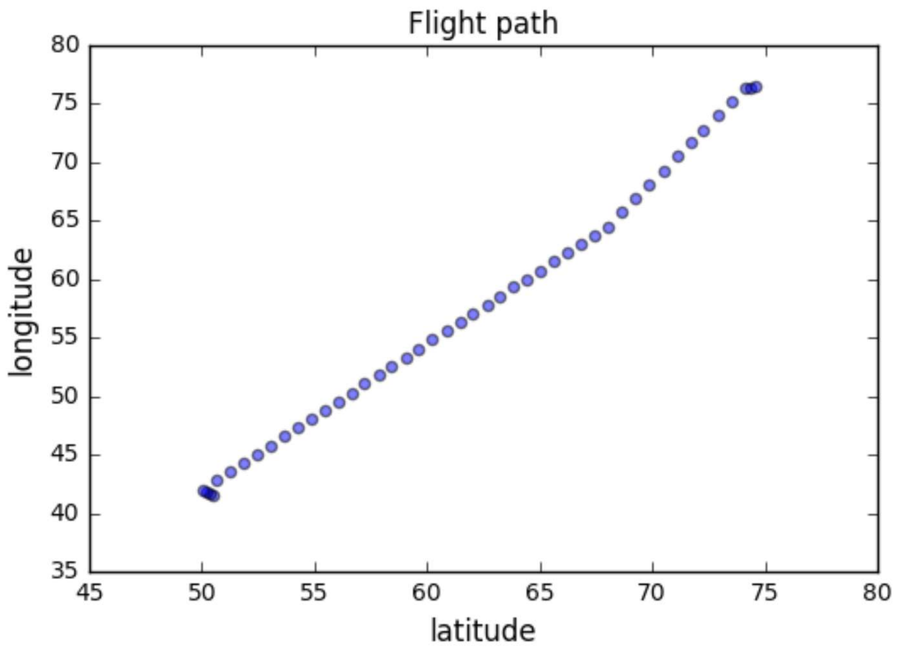
UNSUPERVISED LEARNING IN PYTHON

Intrinsic dimension

Intrinsic dimension of a flight path

- 2 features: longitude and latitude at points along a flight path
- Dataset *appears* to be 2-dimensional
- But can approximate using one feature: displacement along flight path
- Is intrinsically 1-dimensional

```
latitude  longitude
  50.529     41.513
  50.360     41.672
  50.196     41.835
  50.035     42.015
  50.678     42.796
  51.281     43.526
...
```



Intrinsic dimension

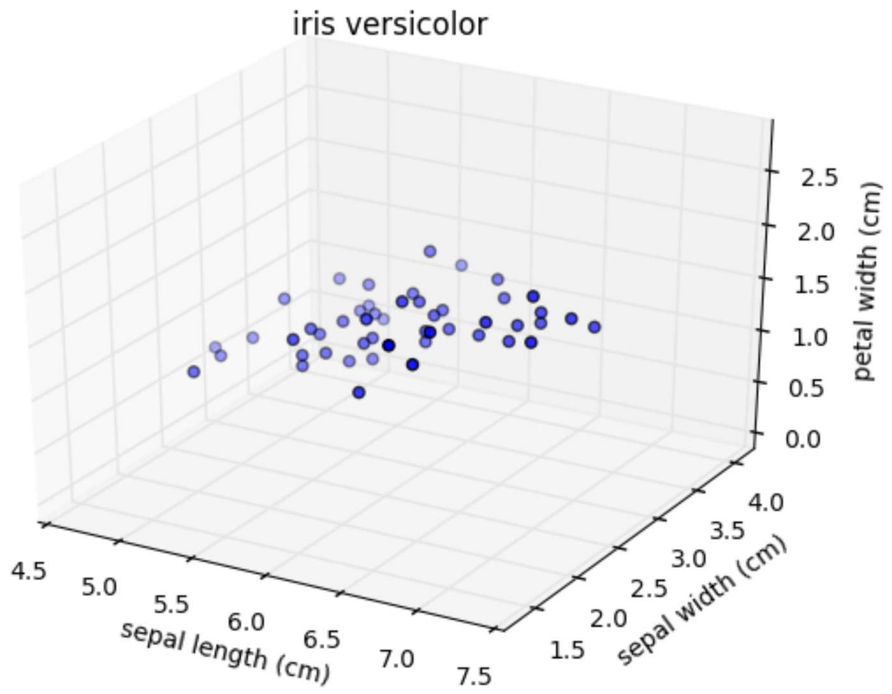
- Intrinsic dimension = number of features needed to approximate the dataset
- Essential idea behind dimension reduction
- What is the most compact representation of the samples?
- Can be detected with PCA

Versicolor dataset

- "versicolor", one of the iris species
- Only 3 features: sepal length, sepal width, and petal width
- Samples are points in 3D space

Versicolor dataset has intrinsic dimension 2

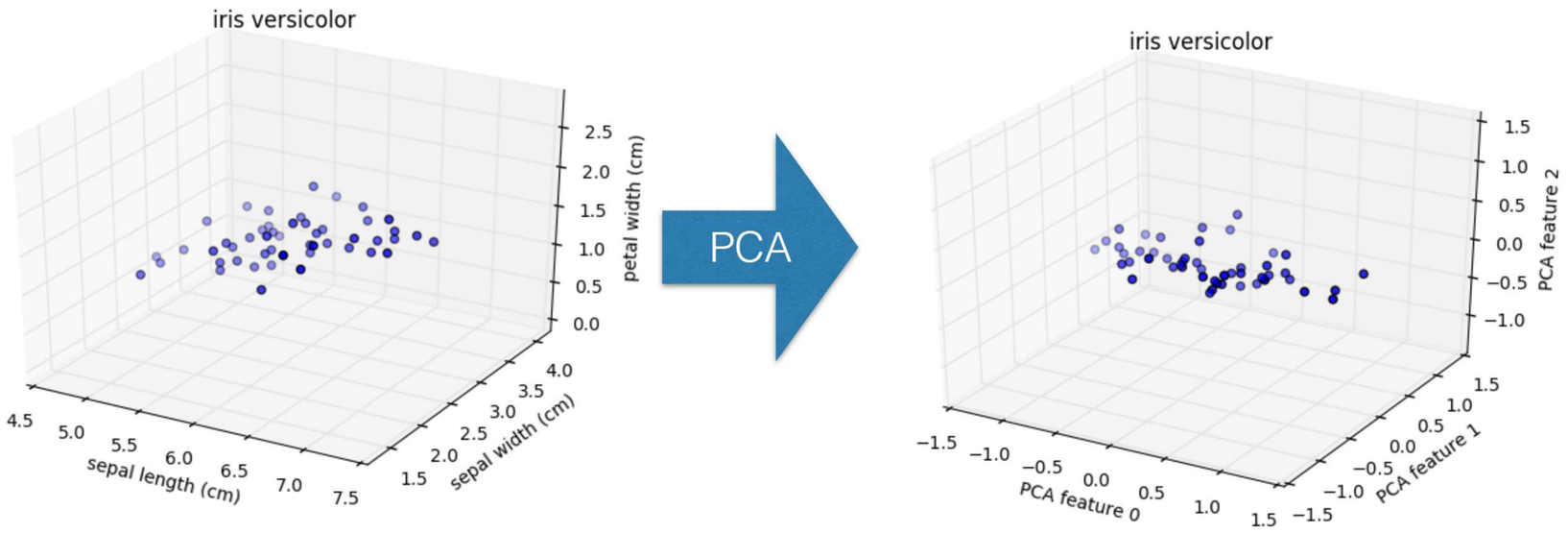
- Samples lie close to a flat 2-dimensional sheet
- So can be approximated using 2 features



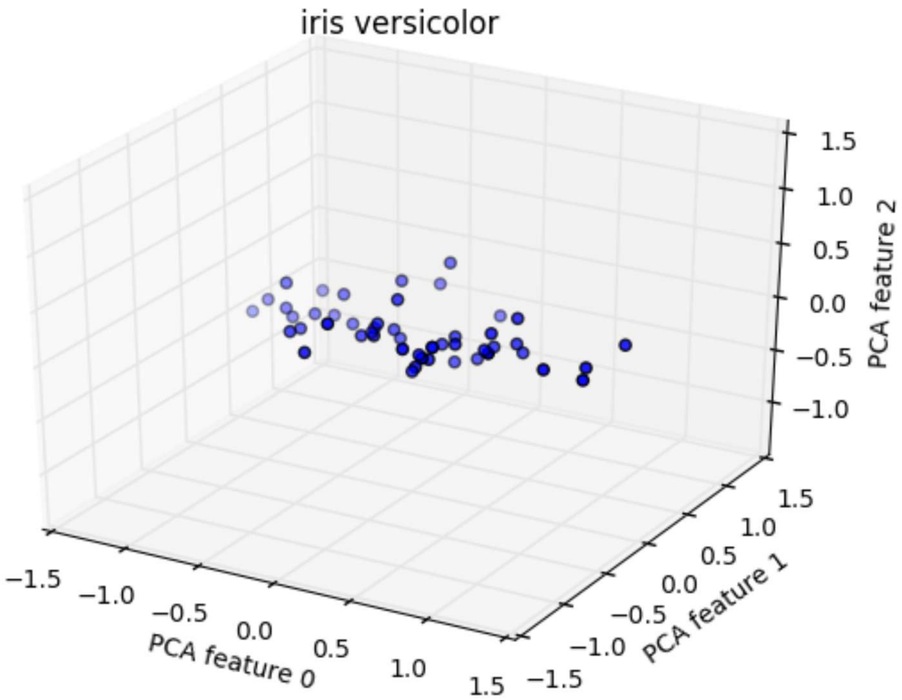
PCA identifies intrinsic dimension

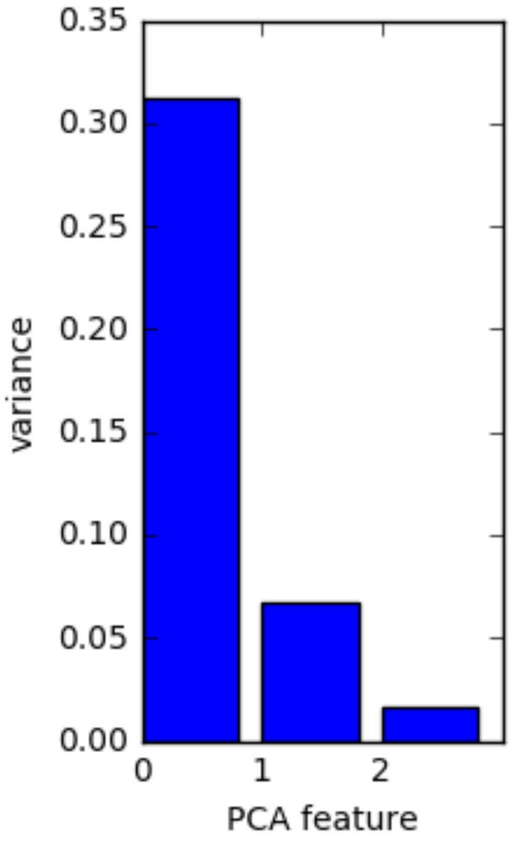
- Scatter plots work only if samples have 2 or 3 features
- PCA identifies intrinsic dimension when samples have *any number* of features
- Intrinsic dimension = number of PCA features with significant variance

PCA of the versicolor samples



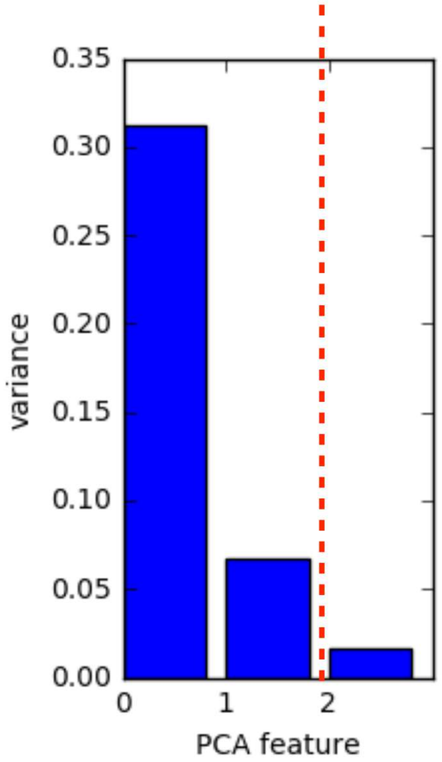
PCA features are ordered by variance descending





Variance and intrinsic dimension

- Intrinsic dimension is number of PCA features with significant variance
- In our example: the first two PCA features
- So intrinsic dimension is 2



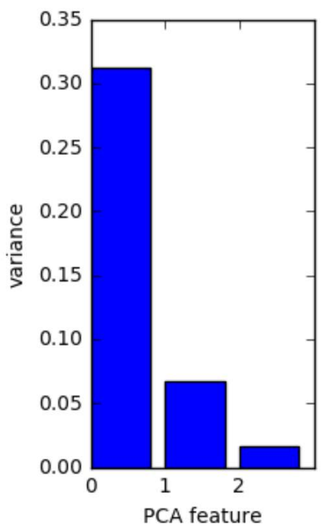
Plotting the variances of PCA features

- **samples** = array of versicolor samples

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

In [2]: from sklearn.decomposition import PCA

In [3]: pca = PCA()

In [4]: pca.fit(samples)
Out[4]: PCA(copy=True, ... )

In [5]: features = range(pca.n_components_)
```

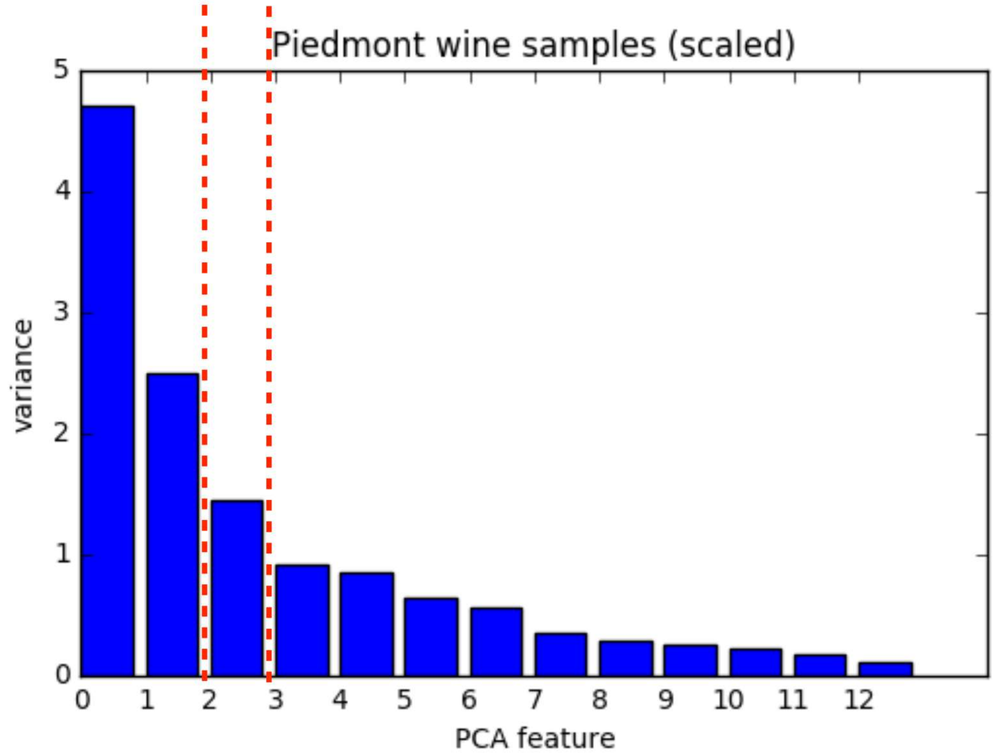
Plotting the variances of PCA features

```
In [6]: plt.bar(features, pca.explained_variance_)

In [7]: plt.xticks(features)

In [8]: plt.ylabel('variance')

In [9]: plt.xlabel('PCA feature')

In [10]: plt.show()
```



Intrinsic dimension can be ambiguous

- Intrinsic dimension is an idealization
- ... there is not always one correct answer!
- Piedmont wines: could argue for 2, or for 3, or more



UNSUPERVISED LEARNING IN PYTHON

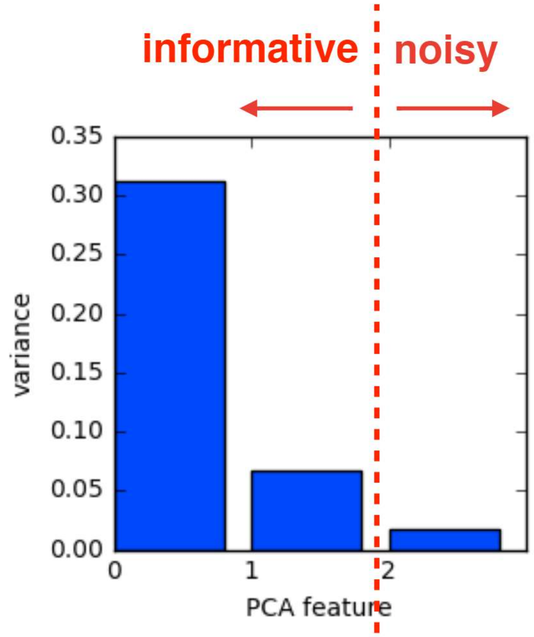
Dimension reduction with PCA

Dimension reduction

- Represents same data, using less features
- Important part of machine-learning pipelines
- Can be performed using PCA

Dimension reduction with PCA

- PCA features are in decreasing order of variance
- Assumes the low variance features are "noise"
- ... and high variance features are informative



Dimension reduction with PCA

- Specify how many features to keep
- E.g. **PCA(n_components=2)**
- Keeps the first 2 PCA features
- Intrinsic dimension is a good choice

Dimension reduction of iris dataset

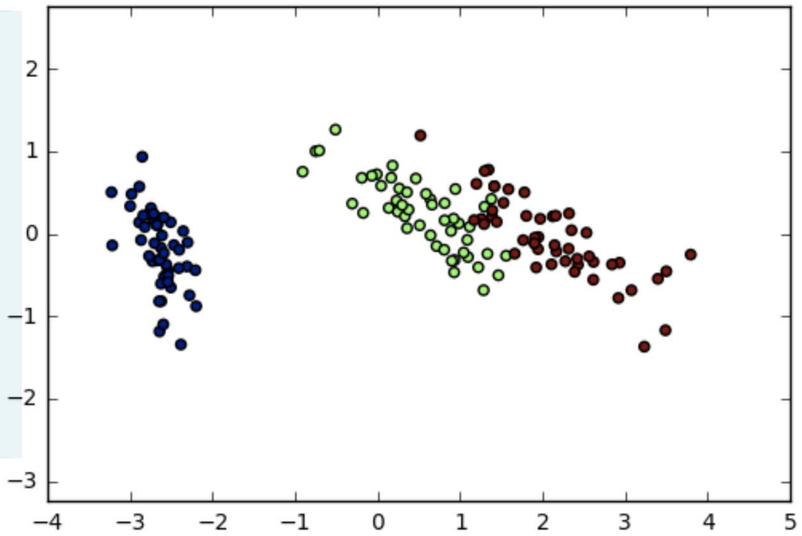
- **samples** = array of iris measurements (4 features)
- **species** = list of iris species numbers

```
In [1]: from sklearn.decomposition import PCA

In [2]: pca = PCA(n_components=2)

In [3]: pca.fit(samples)
Out[3]: PCA(copy=True, ... )

In [4]: transformed = pca.transform(samples)

In [5]: print(transformed.shape)
(150, 2)
```

Iris dataset in 2 dimensions

- PCA has reduced the dimension to 2
- Retained the 2 PCA features with highest variance
- Important information preserved: species remain distinct

```
In [6]: import matplotlib.pyplot as plt

In [7]: xs = transformed[:,0]
   ...: ys = transformed[:,1]

In [8]: plt.scatter(xs, ys, c=species)
   ...: plt.show()
```



Dimension reduction with PCA

- Discards low variance PCA features
- Assumes the high variance features are informative
- Assumption typically holds in practice (e.g. for iris)

Word frequency arrays

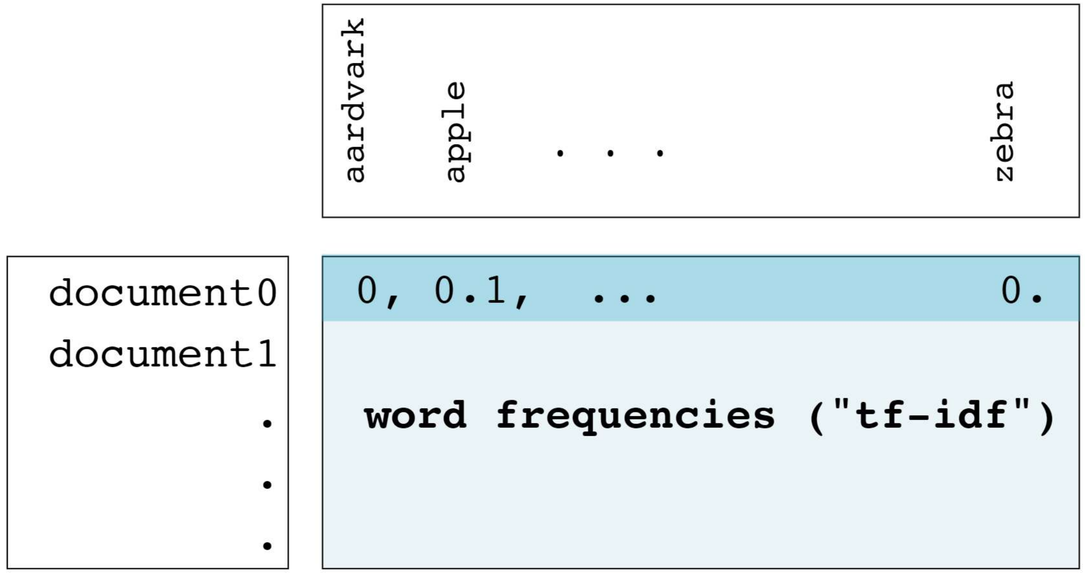
- Rows represent documents, columns represent words
- Entries measure presence of each word in each document
- ... measure using "tf-idf" (more later)

	aardvark	apple	. . .	zebra
document0	0,	0.1,	...	0.
document1				
.				
.				
.				

word frequencies ("tf-idf")

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!)



TruncatedSVD and csr_matrix

- scikit-learn PCA doesn't support **csr_matrix**
- Use scikit-learn **TruncatedSVD** instead
- Performs same transformation

```
In [1]: from sklearn.decomposition import TruncatedSVD

In [2]: model = TruncatedSVD(n_components=3)

In [3]: model.fit(documents)  # documents is csr_matrix
Out[3]: TruncatedSVD(algorithm='randomized', ... )

In [4]: transformed = model.transform(documents)
```

UNSUPERVISED LEARNING IN PYTHON

Non-negative matrix factorization (NMF)

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)

Interpretable parts

- NMF expresses documents as combinations of topics (or "themes")

DataCamp
DataCamp is the first and foremost leader in Data Science Education offering skill-based training, pioneering technical innovation
…

≈ 0.6 *

```
python
r
program
function
method
…
```

+ 0.5 *

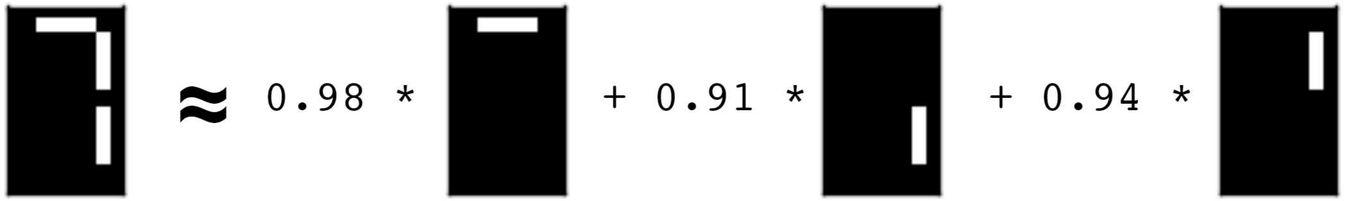
```
data
analysis
cluster
statistics
mean
…
```

+ 0.7 *

```
learn
teaching
lesson
lessons
course
…
```

Interpretable parts

- NMF expresses images as combinations of patterns



Using scikit-learn NMF

- Follows **fit() / transform()** pattern
- Must specify number of components e.g. **NMF(n_components=2)**
- Works with NumPy arrays and with **csr_matrix**

Example word-frequency array

- Word frequency array, 4 words, many documents
- Measure presence of words in each document using "tf-idf"
- "tf" = frequency of word in document
- "idf" reduces influence of frequent words

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

Example usage of NMF

- **samples** is the word-frequency array

```
In [1]: from sklearn.decomposition import NMF

In [2]: model = NMF(n_components=2)

In [3]: model.fit(samples)
Out[3]: NMF(alpha=0.0, ... )

In [4]: nmf_features = model.transform(samples)
```

NMF components

- NMF has components
- ... just like PCA has principal components
- Dimension of components = dimension of samples
- Entries are non-negative

```
In [5]: print(model.components_)
[[ 0.01  0.    2.13  0.54]
 [ 0.99  1.47  0.    0.5 ]]
```

NMF features

- NMF feature values are non-negative
- Can be used to reconstruct the samples
- ... combine feature values with components

```
In [6]: print(nmf_features)
[[ 0.    0.2 ]
 [ 0.19  0.  ]
  ...
 [ 0.15  0.12]]
```

Reconstruction of a sample

```
In [7]: print(samples[i,:])
[ 0.12  0.18  0.32  0.14]

In [8]: print(nmf_features[i,:])
[ 0.15  0.12]
```

model.components_

```
  0.15 *
+ 0.12 *
```

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

reconstruction of sample

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

Sample reconstruction

- Multiply components by feature values, and add up
- Can also be expressed as a product of matrices
- This is the "**M**atrix **F**actorization" in "NMF"

NMF fits to non-negative data, only

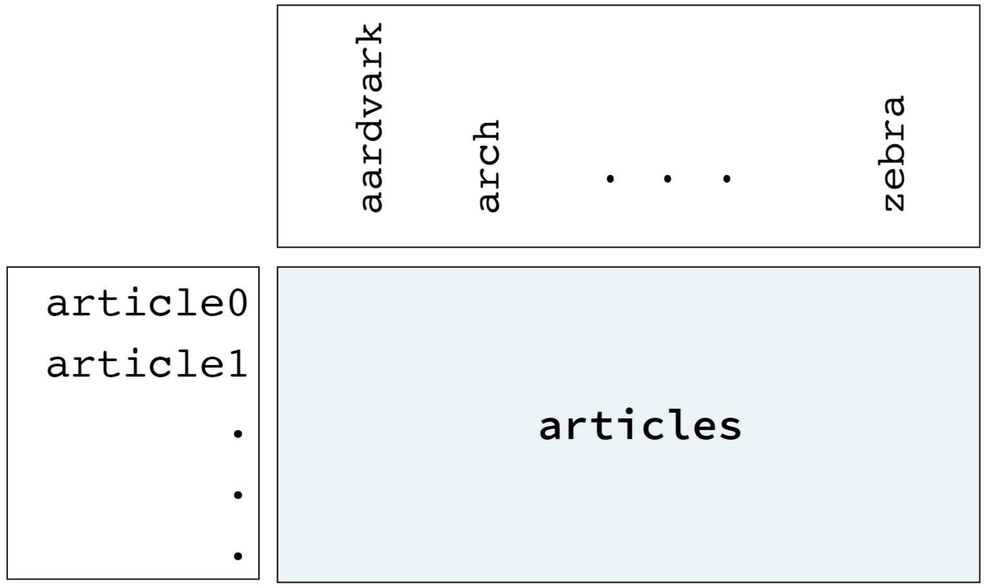
- Word frequencies in each document
- Images encoded as arrays
- Audio spectrograms
- Purchase histories on e-commerce sites
- ... and many more!

UNSUPERVISED LEARNING IN PYTHON

NMF learns interpretable parts

Example: NMF learns interpretable parts

- Word-frequency array **articles** (tf-idf)
- 20,000 scientific articles (rows)
- 800 words (columns)



Applying NMF to the articles

```
In [1]: print(articles.shape)
(20000, 800)

In [2]: from sklearn.decomposition import NMF

In [3]: nmf = NMF(n_components=10)

In [4]: nmf.fit(articles)
Out[4]: NMF(alpha=0.0, ... )

In [5]: print(nmf.components_.shape)
(10, 800)
```

NMF components are topics

aardvark	arch	. . .	zebra
0.1	0.0	. . .	0.1

nmf.components_

```
top words:
species        2.95
plant          1.05
plants         0.78
genetic        0.58
evolution      0.53
life           0.53
...
```

NMF components are topics

aardvark	arch	. . .	zebra

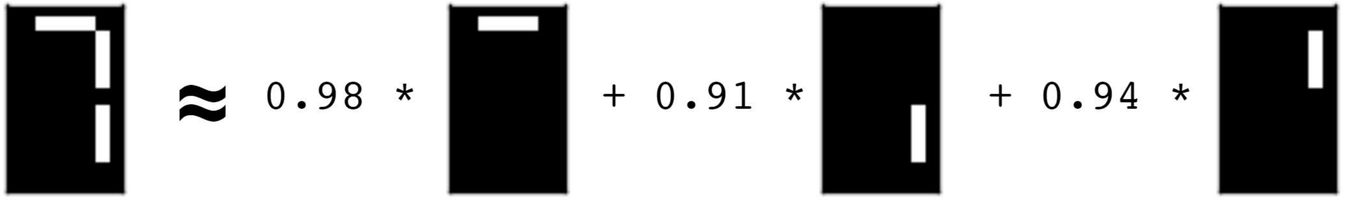
nmf.components_

0.0	0.01	. . .	0.0

```
top words:
university    3.57
prof          1.19
college       0.88
department    0.42
education     0.33
school        0.31
...
```

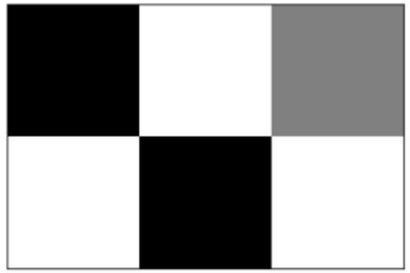
NMF components

- For documents:
  - NMF components represent topics
  - NMF features combine topics into documents
- For images, NMF components are parts of images



Grayscale images

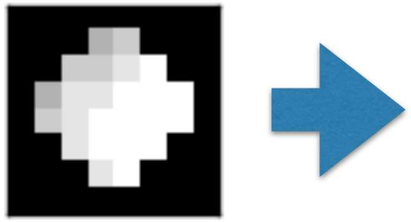
- "Grayscale" image = no colors, only shades of gray
- Measure pixel brightness
- Represent with value between 0 and 1 (0 is black)
- Convert to 2D array



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

Grayscale image example

- An 8x8 grayscale image of the moon, written as an array



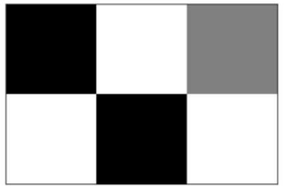
```
[[ 0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.7  0.8  0.   0.   0. ]
 [ 0.   0.   0.8  0.8  0.9  1.   0.   0. ]
 [ 0.   0.7  0.9  0.9  1.   1.   1.   0. ]
 [ 0.   0.8  0.9  1.   1.   1.   1.   0. ]
 [ 0.   0.   0.9  1.   1.   1.   0.   0. ]
 [ 0.   0.   0.   0.9  1.   0.   0.   0.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

Grayscale images as flat arrays

- Enumerate the entries
- Row-by-row
- From left to right

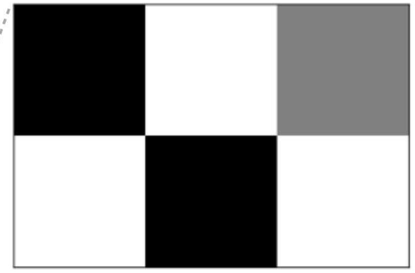
```
[[ 0.   1.   0.5]
 [ 1.   0.   1. ]]
```

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



Encoding a collection of images

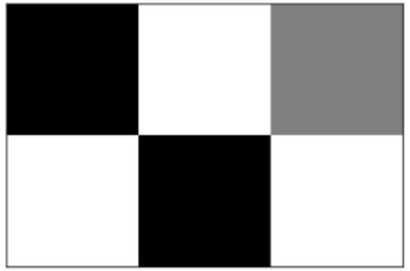
- Collection of images of the same size
- Encode as 2D array
- Each row corresponds to an image
- Each column corresponds to a pixel
- ... can apply NMF!



```
0.   1.   0.5   1.   0.   1.
```

Visualizing samples

```
In [1]: print(sample)
[ 0.   1.   0.5  1.   0.   1. ]

In [2]: bitmap = sample.reshape((2, 3))

In [3]: print(bitmap)
[[ 0.   1.   0.5]
 [ 1.   0.   1. ]]

In [4]: from matplotlib import pyplot as plt

In [5]: plt.imshow(bitmap, cmap='gray', interpolation='nearest')

In [6]: plt.show()
```



UNSUPERVISED LEARNING IN PYTHON

Building recommender systems using NMF

Finding similar articles

- Engineer at a large online newspaper
- Task: recommend articles similar to article being read by customer
- Similar articles should have similar topics

Strategy

- Apply NMF to the word-frequency array
- NMF feature values describe the topics
- … so similar documents have similar NMF feature values
- Compare NMF feature values?

Apply NMF to the word-frequency array

- **articles** is a word frequency array

```
In [1]: from sklearn.decomposition import NMF

In [2]: nmf = NMF(n_components=6)

In [3]: nmf_features = nmf.fit_transform(articles)
```

Strategy

- Apply NMF to the word-frequency array
- NMF feature values describe the topics
- … so similar documents have similar NMF feature values
- Compare NMF feature values?

Versions of articles

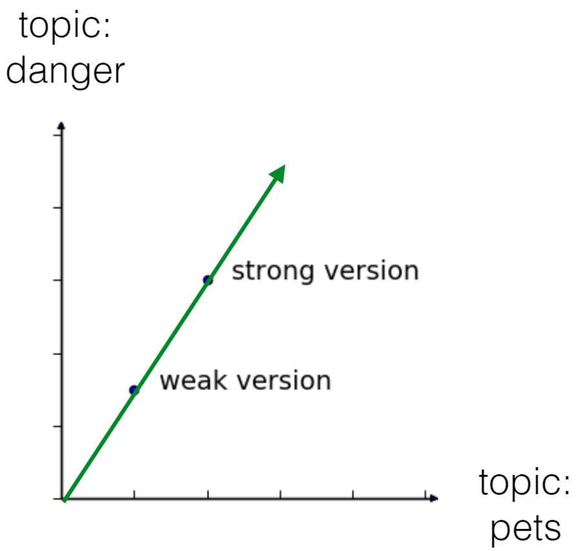
- Different versions of the same document have same topic *proportions*
- ... exact feature values may be different!
- E.g. because one version uses many meaningless words
- But all versions lie on the same *line* through the origin

strong version

Dog bites man! Attack by terrible canine leaves man paralyzed...

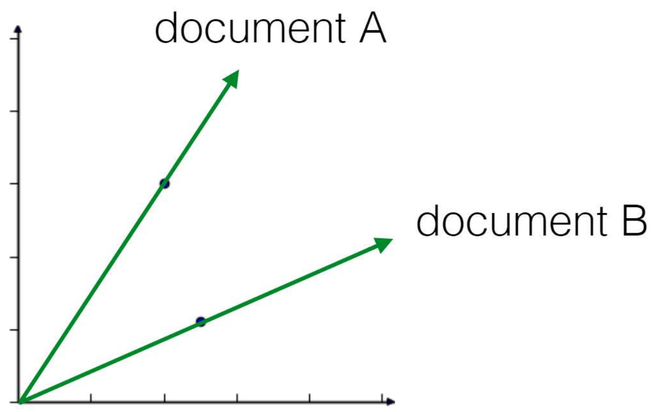
weak version

You may have heard, unfortunately it seems that a dog has perhaps bitten a man ...



Cosine similarity

- Uses the angle between the lines
- Higher values means more similar
- Maximum value is 1, when angle is 0°



Calculating the cosine similarities

```
In [4]: from sklearn.preprocessing import normalize

In [5]: norm_features = normalize(nmf_features)

In [6]: current_article = norm_features[23,:]  # if has index 23

In [7]: similarities = norm_features.dot(current_article)

In [8]: print(similarities)
[ 0.7150569   0.26349967  0.40210445 ...,  0.70462768  0.20323616
  0.05047817]
```

DataFrames and labels

- Label similarities with the article titles, using a DataFrame
- Titles given as a list: **titles**

```
In [9]: import pandas as pd

In [10]: norm_features = normalize(nmf_features)

In [11]: df = pd.DataFrame(norm_features, index=titles)

In [12]: current_article = df.loc['Dog bites man']

In [13]: similarities = df.dot(current_article)
```

DataFrames and labels

```
In [14]: 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
```

UNSUPERVISED LEARNING IN PYTHON

Building recommender systems using NMF

Finding similar articles

- Engineer at a large online newspaper
- Task: recommend articles similar to article being read by customer
- Similar articles should have similar topics

Strategy

- Apply NMF to the word-frequency array
- NMF feature values describe the topics
- … so similar documents have similar NMF feature values
- Compare NMF feature values?

Apply NMF to the word-frequency array

- **articles** is a word frequency array

```
In [1]: from sklearn.decomposition import NMF

In [2]: nmf = NMF(n_components=6)

In [3]: nmf_features = nmf.fit_transform(articles)
```

Strategy

- Apply NMF to the word-frequency array
- NMF feature values describe the topics
- … so similar documents have similar NMF feature values
- Compare NMF feature values?

Versions of articles

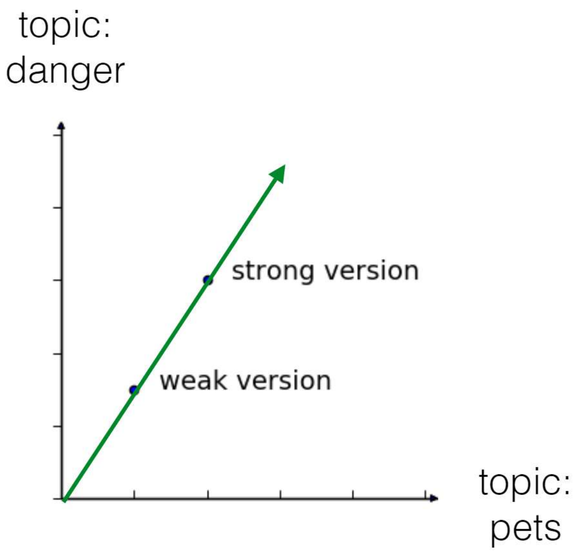
- Different versions of the same document have same topic *proportions*
- ... exact feature values may be different!
- E.g. because one version uses many meaningless words
- But all versions lie on the same *line* through the origin

strong version

Dog bites man! Attack by terrible canine leaves man paralyzed...

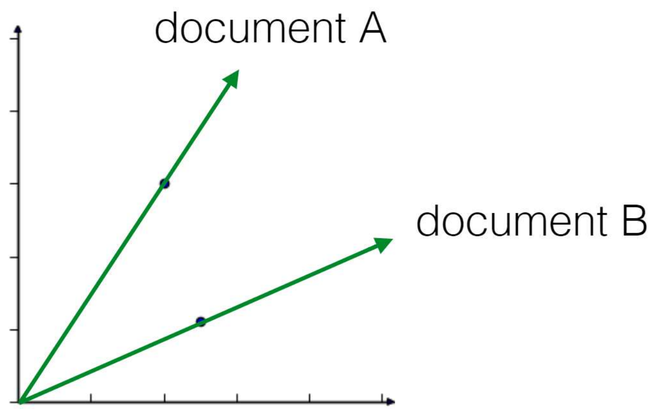
weak version

You may have heard, unfortunately it seems that a dog has perhaps bitten a man ...



Cosine similarity

- Uses the angle between the lines
- Higher values means more similar
- Maximum value is 1, when angle is 0°



Calculating the cosine similarities

```
In [4]: from sklearn.preprocessing import normalize

In [5]: norm_features = normalize(nmf_features)

In [6]: current_article = norm_features[23,:]  # if has index 23

In [7]: similarities = norm_features.dot(current_article)

In [8]: print(similarities)
[ 0.7150569   0.26349967  0.40210445 ...,  0.70462768  0.20323616
  0.05047817]
```

DataFrames and labels

- Label similarities with the article titles, using a DataFrame
- Titles given as a list: **titles**

```
In [9]: import pandas as pd

In [10]: norm_features = normalize(nmf_features)

In [11]: df = pd.DataFrame(norm_features, index=titles)

In [12]: current_article = df.loc['Dog bites man']

In [13]: similarities = df.dot(current_article)
```

DataFrames and labels

```
In [14]: 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
```