Machine learning workshops for material scientists

Lecture 4: Unsupervised learning with Python

October 19, 2022

Sira Sriswasdi, PhD

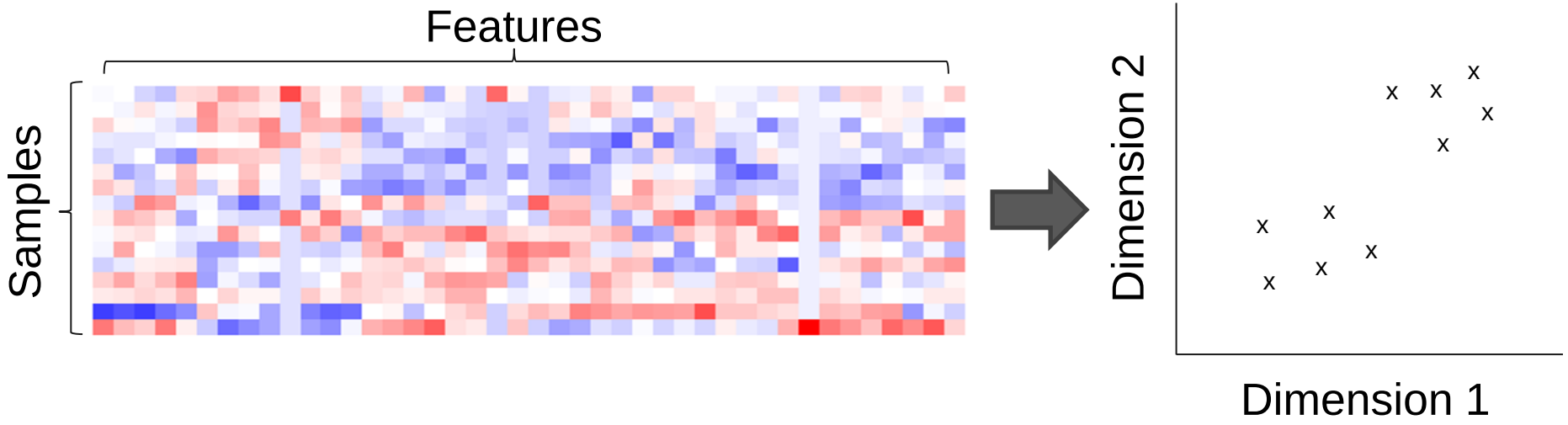
- Research Affairs
- Center of Excellence in Computational Molecular Biology (CMB)
- Center for Artificial Intelligence in Medicine (CU-AIM)

Today’s goals

- Quick recap of key unsupervised learning techniques
- Get to know Python library
- Practice on Colab

Dimensionality reduction

Dimensionality reduction

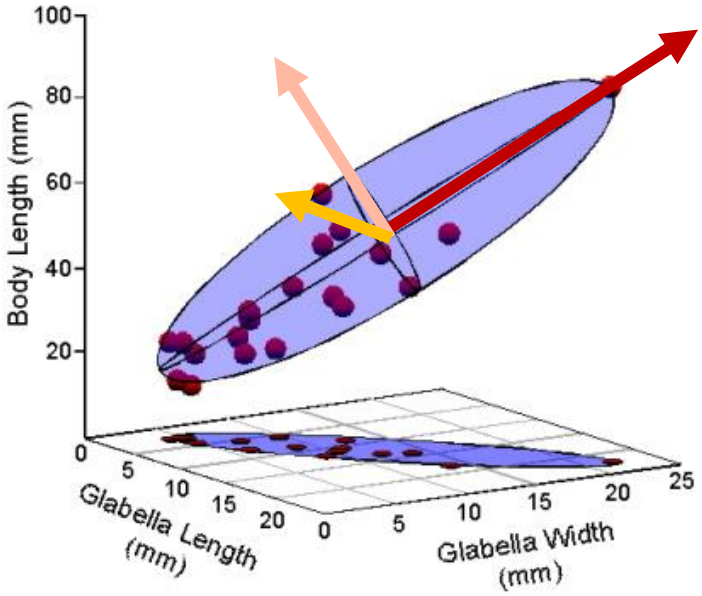


- Summarizing high-dimensional data for visualization
- Based on variance or sample-sample similarity

Dimensionality reduction techniques

- Principal Component Analysis
- Multidimensional Scaling / Principal Coordinate Analysis
- *t*-distributed Stochastic Neighbor Embedding
- Uniform Manifold Approximation and Projection
- Density-Preserving Map
- Diffusion Map

Principal Component Analysis (PCA)



Source: the paleontological association

- Find new directions/axes from highest to lowest variance
  - **Assume**: high variance = informative
- New directions/axes are linear combinations of original features
  - $x_1^{new} = w_1x_1 + \ldots + w_nx_n$
  - Preserve Euclidean distance
- Group correlated features onto the same axes
- Typically used as a prelim step

PCA in Python

- n_components = number of directions/axes to produce
  - default = min(sample, feature)
- Input data will be automatically centered (set mean = 0 for each feature) but not scaled (set variance = 1 for each feature)
- random_state controls the behavior of algorithm that have some randomness
  - Always set to a fixed value for reproducibility

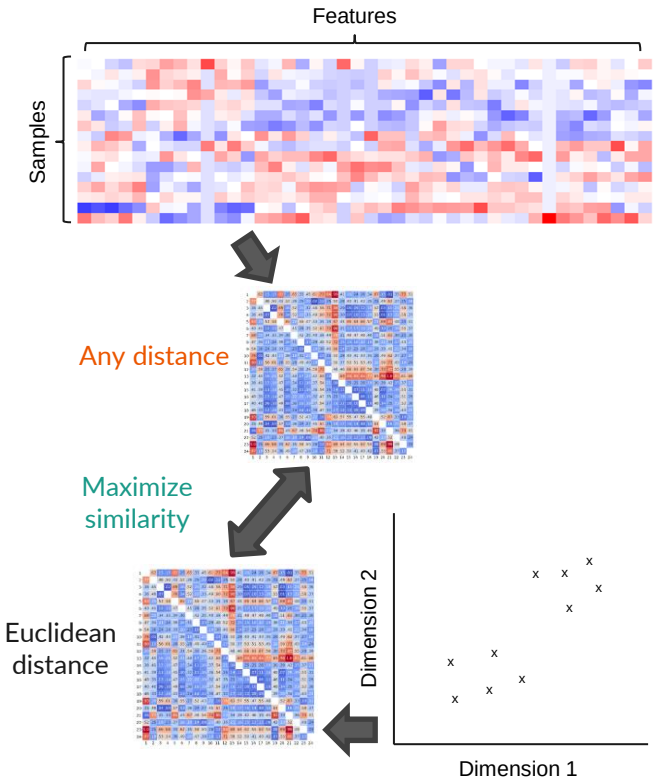
PCA usage

```
from sklearn.decomposition import PCA

pca = PCA(random_state = 25)
pca.fit(input_data)

## a fitted PCA can be used to transform any new data
input_embed = pca.transform(input_data)
output_embed = pca.transform(output_data)

## you can fit & transform directly
new_embed = pca.fit_transform(new_data)
```

PCA output

```
## number of output directions/axes
pca.n_components_

## array of variance explained by each axis (from high to low, sum to one)
pca.explained_variance_ratio_

## 2D array of the loading (w_i's) of each feature on each direction/axis
##    dimension = # direction x # feature
pca.components_

## the loading (w_i's) of each feature on the 3rd direction/axis
pca.components_[2]
```

Multidimensional scaling



- Support any definition of distance/similarity on the original feature space
  - Correlation, edit distance, etc.
  - User-defined matrix
- Find low-dimension embedding that induces a similar sample-sample distance pattern on Euclidean space
  - Visualization is based on Euclidean
- Changing the number of output dimension will change the result

MDS in Python

sklearn.manifold.MDS

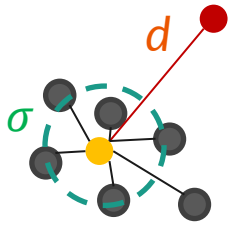
class `sklearn.manifold.MDS`(*n_components=2, *, metric=True, n_init=4, max_iter=300, verbose=0, eps=0.001, n_jobs=None, random_state=None, dissimilarity='euclidean'*) [source]

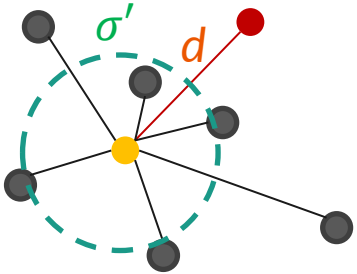
- n_components = number of dimensions to embed onto
- Set metric = True if using a known distance metric
- Set metric = False if inputting a custom dissimilarity that might be a metric
- dissimilarity indicates how the sample-sample distance/dissimilarity should be calculated
  - Use precomputed to input your own custom dissimilarity

MDS usage

```
from sklearn.manifold import MDS
from scipy.spatial.distance import pdist, squareform

## calculate pairwise distance matrix between all samples
corr_mat = squareform(pdist(data, metric = 'correlation'))

## embed with MDS's 'precomputed' option
mds = MDS(n_components = 2, dissimilarity = 'precomputed', random_state = 25)
embed = mds.fit_transform(corr_mat)
```

t-distributed Stochastic Neighbor Embedding





- Convert distance into relative probability of being neighbor
  - Normal distribution on the origin space
  - *t* distribution on the target space

- Scale probability based on the local density of data points

$$\frac{e^{-\frac{d^2}{2\sigma^2}}/\sigma}{\sum e^{-\frac{(\mathrm{dist}(o,\,o))^2}{2\sigma^2}}/\sigma}$$

- Place similar samples near each other and dissimilar samples further away

t-SNE in Python

sklearn.manifold.TSNE

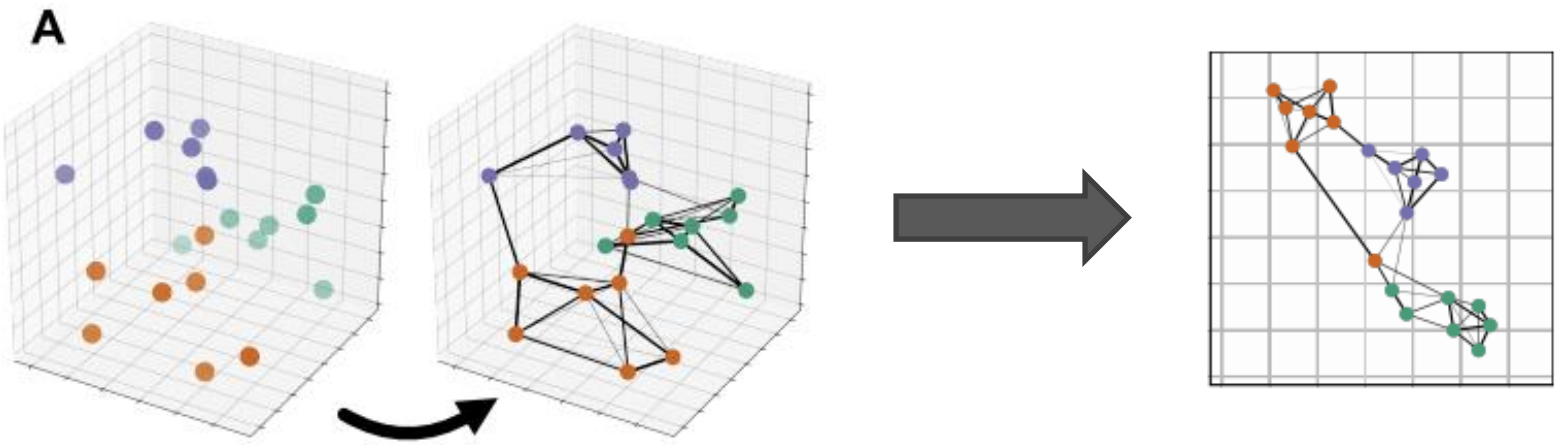
class `sklearn.manifold.`**TSNE**(*n_components=2, *, perplexity=30.0, early_exaggeration=12.0, learning_rate='warn', n_iter=1000, n_iter_without_progress=300, min_grad_norm=1e-07, metric='euclidean', metric_params=None, init='warn', verbose=0, random_state=None, method='barnes_hut', angle=0.5, n_jobs=None, square_distances='deprecated'*) [source]

- n_components = number of dimensions to embed onto
- perplexity ~ number of nearby data points to use when calculating density
  - Recommend trying several values from 5 to 50
- metric defines the distance/dissimilarity
  - Use precomputed to input your own custom dissimilarity
  - For list of available distance, see scipy.spatial.distance.pdist

t-SNE usage

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components = 2, perplexity = 25, metric = 'euclidean', random_state = 25)
embed = tsne.fit_transform(data)

## custom dissimilarity
from scipy.spatial.distance import pdist, squareform

## calculate pairwise distance matrix between all samples
corr_mat = squareform(pdist(data, metric = 'correlation'))

## 'precomputed' option
tsne = TSNE(n_components = 2, perplexity = 25, metric = 'precomputed', random_state = 25)
embed = tsne.fit_transform(corr_mat)
```

Uniform Manifold Approximation and Projection



Sainburg, T. *et al.*, Neural Comput 33(11):2881-2907 (2021)

- **Strong assumption**: data were uniformly sampled from the input space
- Build similarity graph and embed onto low dimension
- Better than *t*-SNE at capturing long-distance relationship
- Can be fitted and applied on new data!

UMAP in Python

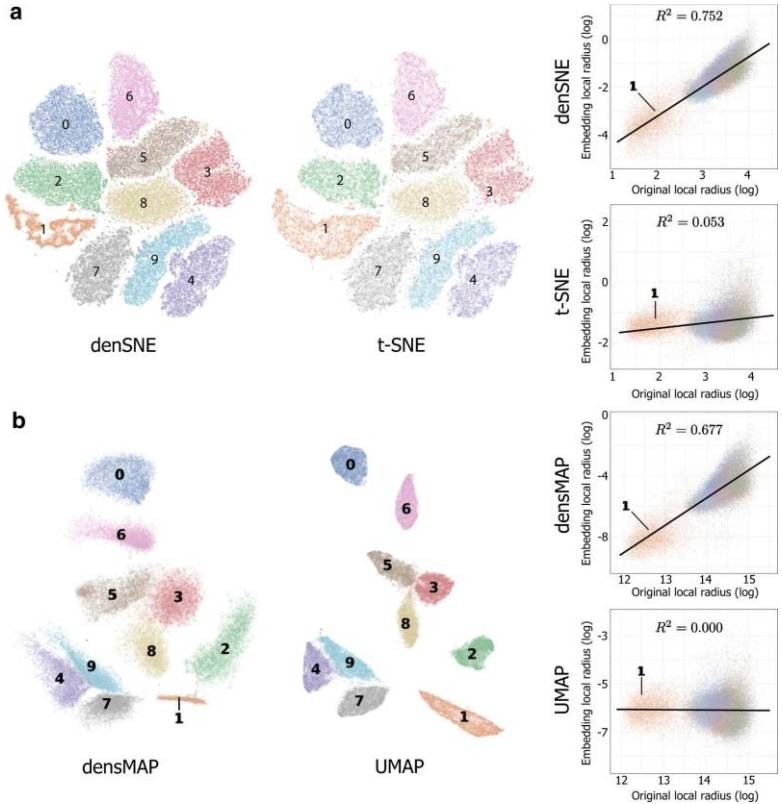
class umap.umap_.UMAP(*n_neighbors=15, n_components=2, metric='euclidean', metric_kwds=None, output_metric='euclidean', output_metric_kwds=None, n_epochs=None, learning_rate=1.0, init='spectral', min_dist=0.1, spread=1.0, low_memory=True, n_jobs=-1, set_op_mix_ratio=1.0, local_connectivity=1.0, repulsion_strength=1.0, negative_sample_rate=5, transform_queue_size=4.0, a=None, b=None, random_state=None, angular_rp_forest=False, target_n_neighbors=-1, target_metric='categorical', target_metric_kwds=None, target_weight=0.5, transform_seed=42, transform_mode='embedding', force_approximation_algorithm=False, verbose=False, tqdm_kwds=None, unique=False, densmap=False, dens_lambda=2.0, dens_frac=0.3, dens_var_shift=0.1, output_dens=False, disconnection_distance=None, precomputed_knn=(None, None, None)*) [source]

- n_components = number of dimensions to embed onto
- n_neighbors = number of nearby data points to use when calculating neighbor graph
  - Recommend trying several values based on dataset size
- min_dist = minimum distance between embedded data points
  - For visualization purpose
- metric (see https://umap-learn.readthedocs.io/en/latest/api.html)

UMAP usage

```
!pip install umap-learn
from umap import UMAP

## fit on a dataset
umap = UMAP(n_components = 2, n_neighbors = 25, metric = 'euclidean', min_dist = 0.5,
random_state = 25).fit(input_data)

## apply to any dataset
input_embed = umap.transform(input_data)
output_embed = umap.transform(output_data)
```

Density-Preserving Map



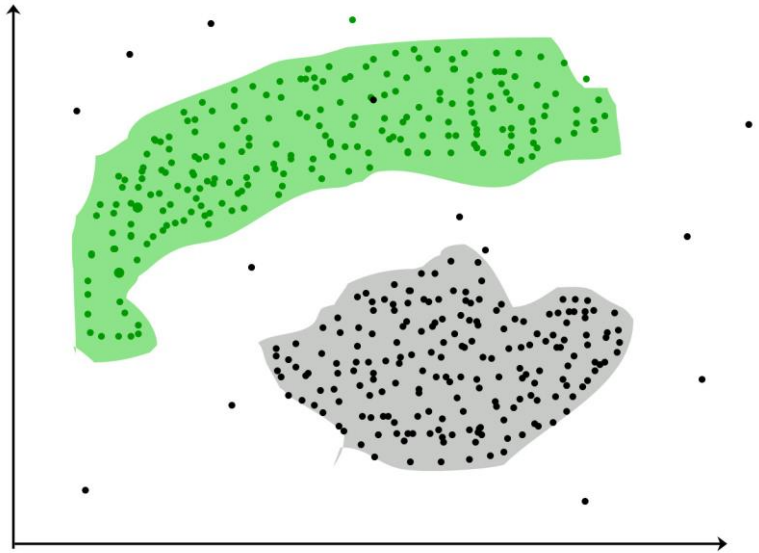
- Original *t*-SNE and UMAP scale the distance so that the density is 1 on the output space
- In some case, density of the data points can be informative
  - denSNE and densMAP preserve the density across embedding process
- Add density loss to the optimization objective

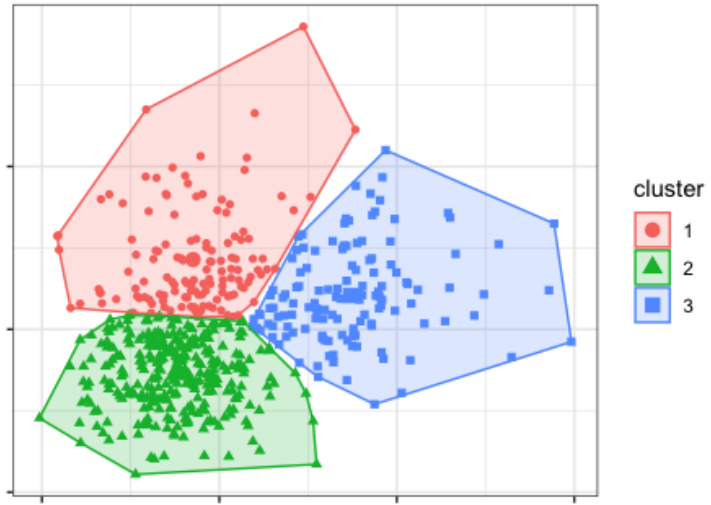
densMAP in Python

```
from umap import UMAP

## fit on a dataset
umap = UMAP(n_components = 2, n_neighbors = 25, metric = 'euclidean', min_dist = 0.5,
random_state = 25, densmap = True).fit(input_data)

## apply on any dataset
input_embed = umap.transform(input_data)
output_embed = umap.transform(output_data)
```

Clustering

Clustering



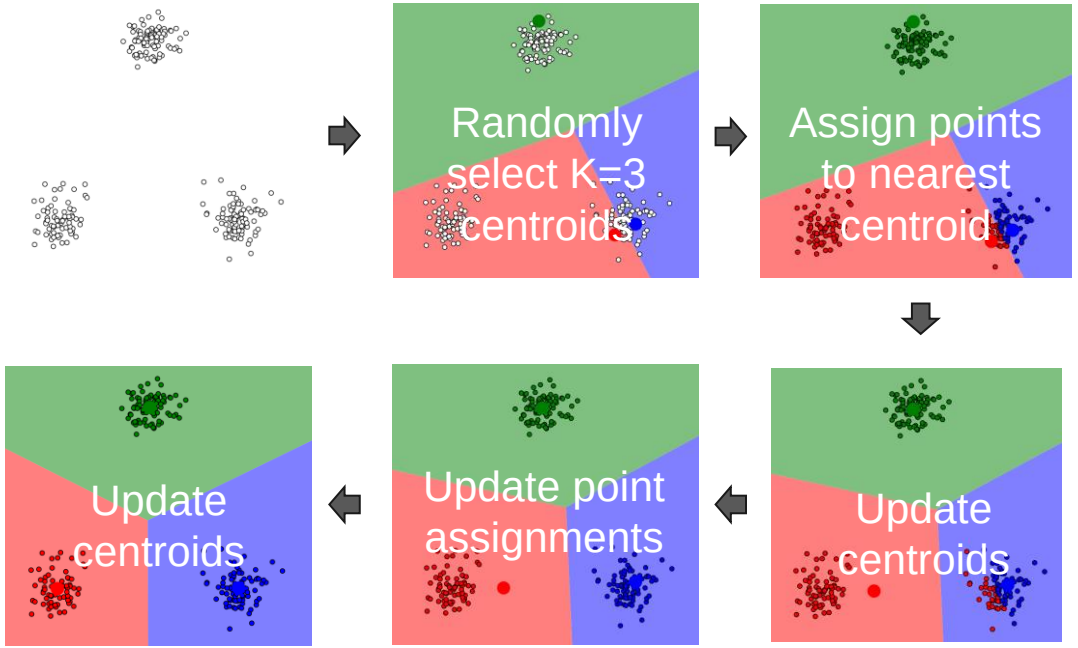


- Identify data subgroup based on density and/or sample-sample similarity

Clustering techniques

- *k*-mean
- Hierarchical/Agglomerative
- DBSCAN (density-based)

k-mean clustering



- Assume that data consist of *k* groups
- Assume that each data group has similar radius & size
- Guess the locations of *k* centroids and update group assignment iteratively
- Assume Euclidean distance

k-mean clustering in Python

sklearn.cluster.KMeans

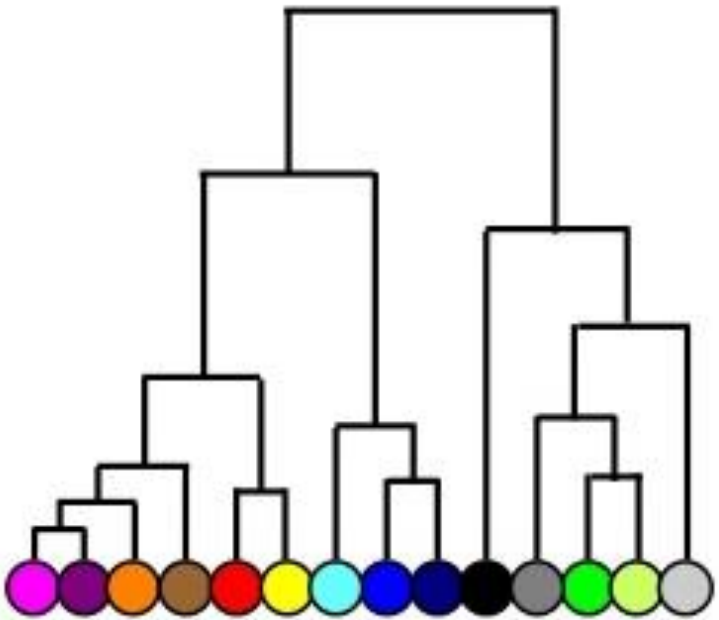
class `sklearn.cluster.`**`KMeans`**(*n_clusters=8, *, init='k-means++', n_init=10, max_iter=300, tol=0.0001, verbose=0, random_state=None, copy_x=True, algorithm='lloyd'*)

- n_clusters = number of clusters
- n_init = number of random initial centroid locations to try
  - The final output is the best among all runs

k-mean clustering usage

```
from sklearn.cluster import KMeans

## fit on a dataset
kmeans = KMeans(n_clusters = 5, n_init = 20).fit(input_data)

## predict cluster ID on any dataset
input_cluster = kmeans.predict(input_data)
output_cluster = kmeans.predict(output_data)

## calculate distance to the centroids
input_dist = kmeans.transform(input_data)
output_dist = kmeans.transform(output_data)
```

Hierarchical/Agglomerative clustering



- Link the most similar samples first
  - Works with any dissimilarity function
- How to calculate similarity between groups of samples?
  - Linkage function
  - Average, max, min, etc.
- Where to cut?
  - Number of target clusters
  - Maximum allowed dissimilarity within group

Agglomerative clustering in Python

sklearn.cluster.AgglomerativeClustering

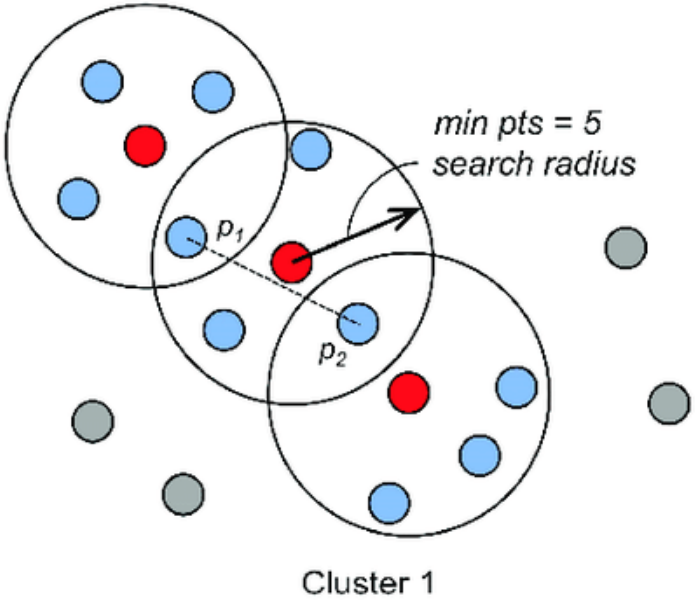
class sklearn.cluster.**AgglomerativeClustering**(*n_clusters=2, *, affinity='euclidean', memory=None, connectivity=None, compute_full_tree='auto', linkage='ward', distance_threshold=None, compute_distances=False*) [source]

- n_clusters = number of clusters
- distance_threshold = maximum allowed distance for joining two data groups
- affinity = metric for calculating sample-sample dissimilarity
  - Use precomputed to input custom sample-sample distance matrix
- linkage = metric for calculating group-group dissimilarity
  - Choices are ward, complete, average, and single.

Agglomerative clustering usage

```
from sklearn.cluster import AgglomerativeClustering

## fit on a dataset
aggclust = AgglomerativeClustering(n_clusters = 5, affinity = 'cosine', linkage =
'single').fit(data)

## get cluster IDs
cluster_id = aggclust.labels_

## fit and get cluster IDs directly
cluster_id = AgglomerativeClustering(n_clusters = 5, affinity = 'cosine', linkage =
'single').fit_predict(data)
```

DBSCAN



- Find core data points with high density as seeds of clusters
- Connect nearby data points to grow the clusters
- Disconnected data points are labeled as outliers

Difrancesco, P.-M. Remote Sensing 12:1885 (2020)

DBSCAN in Python

sklearn.cluster.DBSCAN

class `sklearn.cluster.`**DBSCAN**(*eps=0.5, *, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=None*)

- eps = maximum distance to define data points as neighbors
  - Must be tuned
- min_samples = minimum number of neighbors of a data point to be called “core”
- metrics = metric for calculating sample-sample dissimilarity
  - See sklearn.metrics.pairwise_distances

DBSCAN usage

```
from sklearn.cluster import DBSCAN

## fit on a dataset
dbscan = DBSCAN(eps = 0.5, min_samples =10, metric = 'euclidean').fit(data)

## get cluster IDs
cluster_id = dbscan.labels_

## fit and get cluster IDs directly
cluster_id = DBSCAN(eps = 0.5, min_samples =10, metric = 'euclidean').fit_predict(data)
```

Any question?