L5 – Unsupervised Learning: Clustering

- Different clustering techniques are to be learned here
  - k-Means Clustering
  - Agglomerative Clustering
  - DBSCAN
- Comparing and evaluating different clustering algorithms
- Summary of clustering methods

k-Means Clustering

- One of the simplest and most commonly used clustering algorithms
  - Function: find cluster centers that are representative of certain regions of the data
- Alternating between two steps:
  - Assigning each data point to the closest cluster center
  - Setting each cluster center as the mean of the data points that are assigned to
- The following example illustrates the algorithm on a synthetic dataset:

mglearn.plots.plot_kmeans_algorithm()

- Given new points, k-means will assign each to the closest cluster center
  - Can show the boundaries of the cluster centers already learned

```
mglearn.plots.plot_kmeans_boundaries()
```

  - Learning by k-means can be conducted simply

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
# generate synthetic two-dimensional data
X, y = make_blobs(random_state=1)
# build the clustering model
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

  - Find each training sample's cluster label

```
print("Cluster memberships:\n{}".format(kmeans.labels_))
```

  - Assign cluster labels to new points

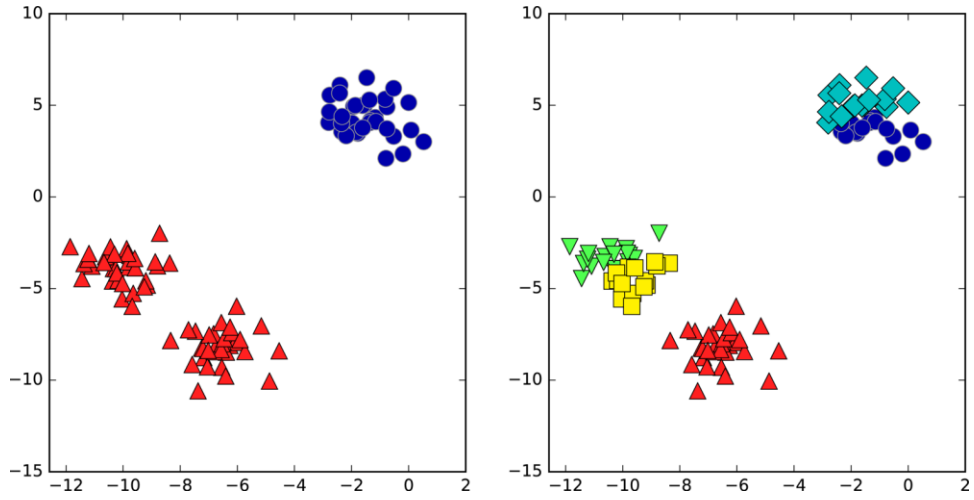
```
# running predict on the training set returns the same result as labels_
print(kmeans.predict(X))
```

- Running again may result in a different numbering of clusters because of the random nature of initialization
  - The cluster centers are stored in the cluster_centers_ attribute

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(kmeans.cluster_centers_[:, 0],
                         kmeans.cluster_centers_[:, 1], [0, 1, 2], markers='^', markeredgewidth=2)
```

- We can also use more or fewer cluster centers

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
# using two cluster centers:
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_
mglearn.discrete_scatter(X[:, 0], X[:, 1],
                         assignments, ax=axes[0])
# using five cluster centers:
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_
mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```



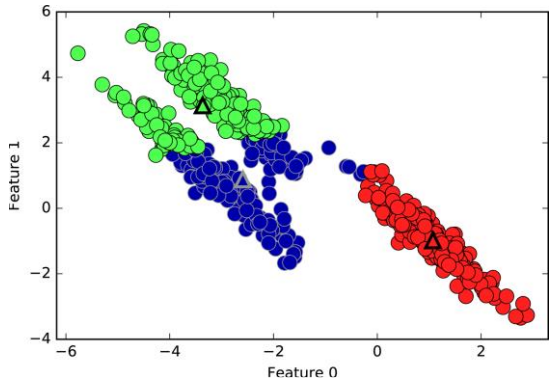
- Failure cases of k-means
  - Even if you know the “right” number of clusters, it still may not be able to recover them
  - As each cluster is defined solely by its center, each cluster can only be a convex shape (i.e., only simple shape can be captured)
  - Assume all clusters have the same “diameter” in some sense

```
X_varied, y_varied = make_blobs(n_samples=200, cluster_std=[1.0, 2.5, 0.5], random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)
mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["cluster 0", "cluster 1", "cluster 2"], loc='best')
plt.xlabel("Feature 0")     plt.ylabel("Feature 1")
```

  - Also assume that all directions are equally important (see below)

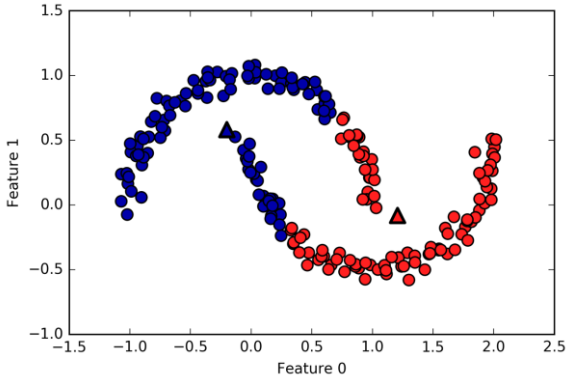
```
# generate some random cluster data
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)
# transform the data to be stretched
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)
```

```
# cluster the data into three clusters
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)
# plot the cluster assignments and cluster centers
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],
                         markers='^', markeredgewidth=2)
plt.xlabel("Feature 0")     plt.ylabel("Feature 1")
```



– Also performs poorly if the clusters have more complex shapes

```
# generate synthetic two_moons data (with less noise this time)
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# cluster the data into two clusters
kmeans = KMeans(n_clusters=2)      kmeans.fit(X)
y_pred = kmeans.predict(X)
# plot the cluster assignments and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60, edgecolor='k')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
          marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2, edgecolor='k')
plt.xlabel("Feature 0")     plt.ylabel("Feature 1")
```



- Vector Quantization (Seeing k-means as Decomposition)
  - PCA tries to find directions of maximum variance in the data
  - NMF tries to find additive components, which of correspond to "extremes" of "parts" of the data
  - *k*-means tries to represent each data point using a cluster center
- See a side-by-side comparison on the face dataset

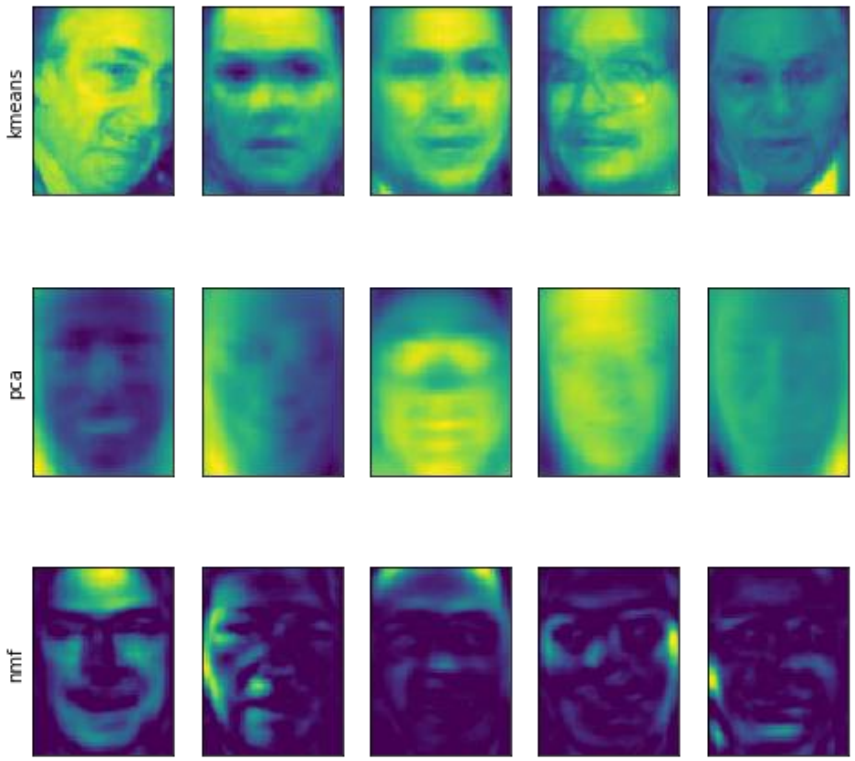
```
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from sklearn.decomposition import NMF
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1
X_people = people.data[mask]
y_people = people.target[mask]
```

```
# scale the grayscale values to be between 0 and 1
# instead of 0 and 255 for better numeric stability
X_people = X_people / 255
print(X_people.shape)
print(y_people.shape)

X_train, X_test, y_train, y_test = train_test_split(X_people, y_people, stratify=y_people, random_state=0)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
pca = PCA(n_components=100, random_state=0)
pca.fit(X_train)
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)
X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)
print("Learning completed!")
```

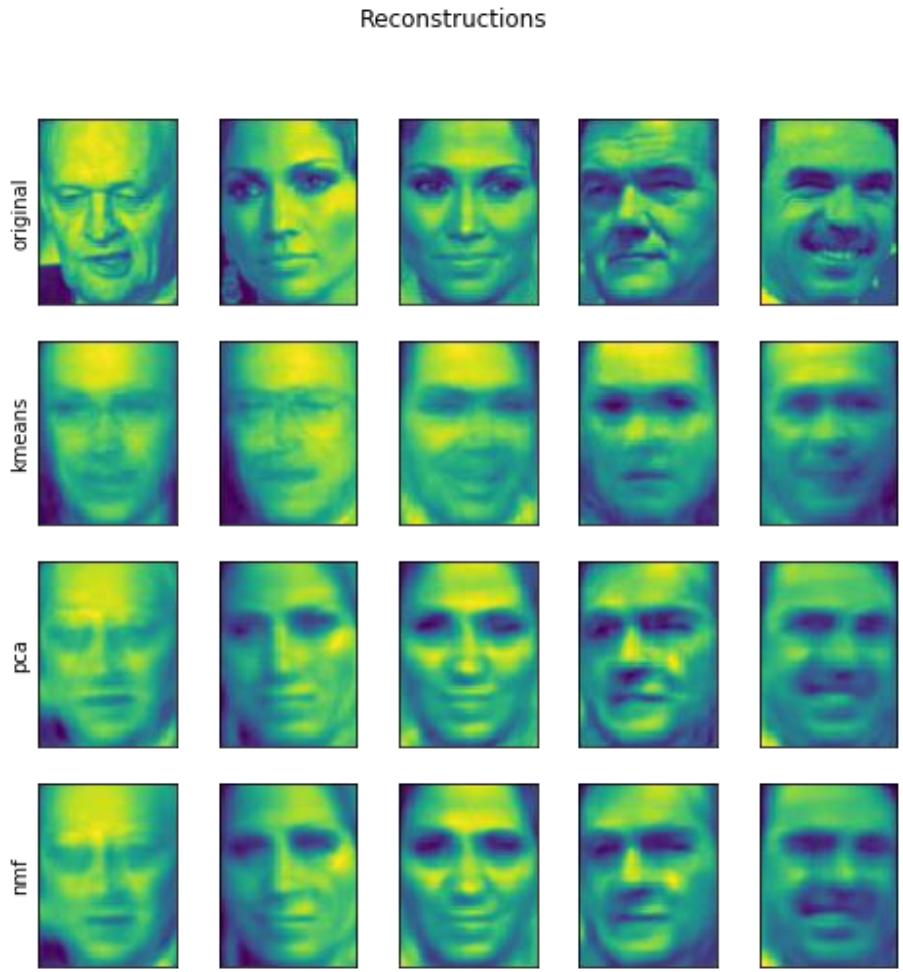
```
import matplotlib.pyplot as plt
image_shape = people.images[0].shape
fig, axes = plt.subplots(3, 5, figsize=(8, 8), subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("Extracted Components")
for ax, comp_kmeans, comp_pca, comp_nmf in zip(
                                axes.T, kmeans.cluster_centers_, pca.components_, nmf.components_):
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')
    ax[2].imshow(comp_nmf.reshape(image_shape))
axes[0, 0].set_ylabel("kmeans")
axes[1, 0].set_ylabel("pca")
axes[2, 0].set_ylabel("nmf")
```



Extracted Components

- See also reconstructions of faces using 100 components

```
fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()}, figsize=(8, 8))
fig.suptitle("Reconstructions")
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(axes.T, X_test, X_reconstructed_kmeans,
                                                  X_reconstructed_pca, X_reconstructed_nmf):
    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(rec_kmeans.reshape(image_shape))
    ax[2].imshow(rec_pca.reshape(image_shape))
    ax[3].imshow(rec_nmf.reshape(image_shape))
axes[0, 0].set_ylabel("original")
axes[1, 0].set_ylabel("kmeans")
axes[2, 0].set_ylabel("pca")
axes[3, 0].set_ylabel("nmf")
```



- An interesting aspect of vector quantization using *k*-means
  - We can use many more clusters than input dimensions
  - PCA or NMF cannot do this
  - As a result, we can find a more expressive rep. with *k*-means

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
kmeans = KMeans(n_clusters=10, random_state=0)
kmeans.fit(X)
y_pred = kmeans.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=60,
            marker='^', c=range(kmeans.n_clusters), linewidth=2, cmap='Paired')
plt.xlabel("Feature 0")     plt.ylabel("Feature 1")
print("Cluster memberships:\n{}".format(y_pred))
```

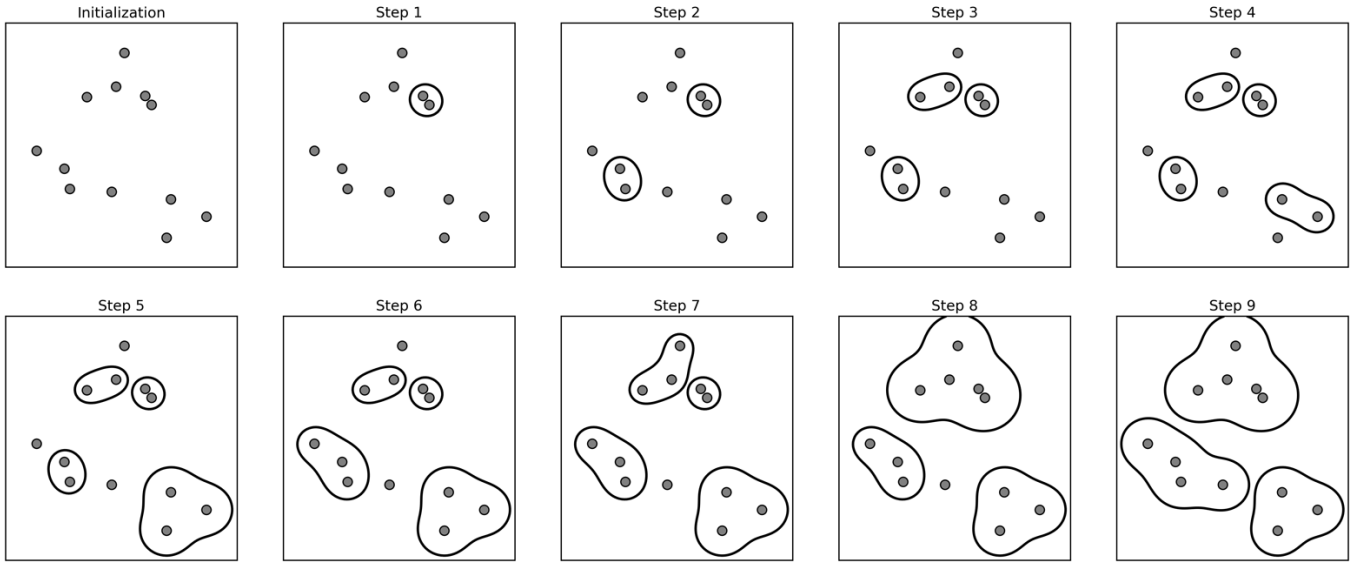
  - Using these 10-dimensional representation to represent original dataset with 2-dimension (i.e., transform)

```
distance_features = kmeans.transform(X)
print("Distance feature shape: {}".format(distance_features.shape))
print("Distance features:\n{}".format(distance_features))
```

- Advantages of k-means
  - Relatively easy to understand and implement
  - Runs relatively fast
- Downside of k-means
  - Relatively restrictive assumptions made on the shape of clusters
  - Requirement to specify the number of clusters you are looking for
- The following clustering algorithm can somewhat improve these properties
  - Agglomerative Clustering
  - DBSCAN (**D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise)

Agglomerative Clustering

- Refers to a collection of clustering algorithms all build upon the same principles
  - The algorithm starts by declaring each point its own cluster
  - Then merges the two most similar clusters iteratively
  - Until some stopping criterion is satisfied (e.g., # of clusters)



Agglomerative clustering iteratively joins the two closest clusters

- There are several linkage criteria that specify how exact the "most similar cluster" is measured
  - ward (default): picks the two clusters to merge such that the variance within all clusters increases the least – this often leads to clusters that are relatively equally sized
  - average: merges the two clusters that have the smallest average distance between all their points
  - Complete (also known as maximum linkage): merges the two clusters that have the smallest maximum distance between their points
- How to choose linkage
  - ward works on most datasets
  - If the clusters have very dissimilar cluster of members (e.g., one cluster is much bigger than all the others), average or complete might work better

- Because of the way the algorithm works, agglomerative clustering cannot make predictions for new data points.
  - As a result, use the fit_predict method instead

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

X, y = make_blobs(random_state=1)
agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.legend(["Cluster 0", "Cluster 1", "Cluster 2"], loc="best")
plt.xlabel("Feature 0")
plt.ylabel("Feature 1”)
```

  - The methods recovers the clustering perfectly
  - You need to specify the number of clusters, but how?

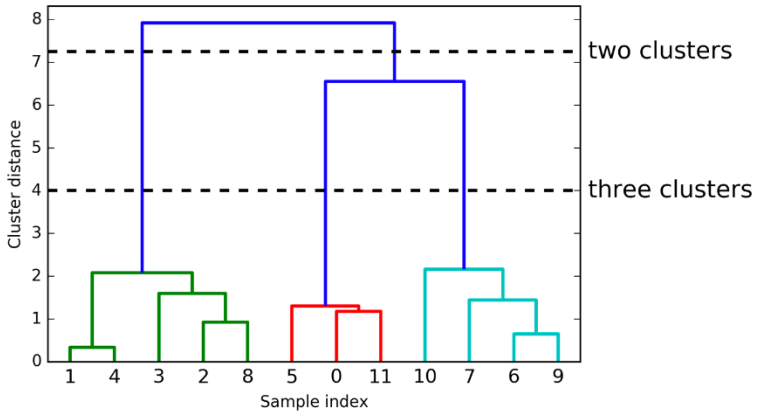
- Hierarchical clustering and dendrograms
  - Agglomerative clustering provides a hierarchical clustering
  - Each intermediate step provides a clustering of the data

mglearn.plots.plot_agglomerative()

  - Dendrogram as a good tool to visualize hierarchical clustering
    - Show data points as points on the bottom
    - Tree structure very clear for us to analyze the clustering

```
# Import the dendrogram function and the ward clustering function from SciPy
from scipy.cluster.hierarchy import dendrogram, ward
X, y = make_blobs(random_state=0, n_samples=12)
# Apply the ward clustering to the data array X
# The SciPy ward function returns an array that specifies the distances bridged
# when performing Agglomerative Clustering
linkage_array = ward(X)
# Now we plot the dendrogram for the linkage_array containing the distances between clusters
dendrogram(linkage_array)
```

```
# Mark the cuts in the tree that signify two or three clusters
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')
ax.text(bounds[1], 7.25, ' two clusters', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' three clusters', va='center', fontdict={'size': 15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```



- Limitation of agglomerative clustering methods
  - Still fails at separating complex shapes like the two_moons dataset
  - Which however can be successfully covered by DBSCAN introduced below

DBSCAN

- Density Based Spatial Clustering of Applications with Noise
  - Main Benefit:
    - not need to specify the number of clusters as prior
    - can capture clusters of complex shapes
    - can identify points nor part of any cluster
  - Downside:
    - slower than k-means and agglomerative clustering
    - but still scales to relatively large datasets
- Idea behind: clusters form dense regions of data, separated by regions that are relatively empty
  - Points that are within a dense region are called core samples

- Two parameters in DBSCAN: min_samples & eps
  - If there are at least min_samples many data points within a distance eps to a given data point, that data point is classified as a core sample.
  - Core samples that are closer to each other than the distance eps are put into the same cluster by DBSCAN
- Algorithm: neighborhood flooding based
  - Starting from a randomly picked point **q**
  - Find all point with distance to **q** less than eps
    - If the pnt # less than min_samples, **q** is classified as noise (means that **q** does not belong to any cluster)
    - If the pnt # more than min_samples, **q** is labeled a core sample and assigned a new cluster label L.
    - All neighbors (within eps) are visited by flooding – if they have not assigned a label, assign L as their label.
  - Picking another unvisited point $\mathbf{q}^{\text{next}}$ and repeating above steps

- In the end of above algorithm, three kinds of points
  - Core points: only neighboring to points have the same label
  - Boundary points: neighboring to core pnts have different labels
  - Noise: not neighboring to enough number of core points
- Try DBSCAN on the synthetic dataset

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs
X, y = make_blobs(random_state=0, n_samples=12)
dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
print("Cluster memberships:\n{}".format(clusters))
```

  - All data points are considered as noise
  - Caused by using the default parameter for eps & min_samples
  - Study the influence of parameters

```
import mglearn
mglearn.plots.plot_dbscan()
```

- Observation:
  - eps ↗, more pnts will be included in a cluster
  - min_samples ↗, fewer points will be core pnts (more restrictive)
- Setting eps implicitly controls # of clusters to be formed

```
from sklearn.preprocessing import StandardScaler
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
dbscan = DBSCAN()      #default eps=0.5; change to eps=0.2 (8 clusters); eps=0.7 (1 cluster)
clusters = dbscan.fit_predict(X_scaled)

# plot the cluster assignments
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap='Paired', s=60)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1”)
```

  - The return of -1 needs to be carefully handled (noise)

Evaluating Different Clustering Algorithms

- Evaluate clustering with ground truth – metrics to be used:
  - Adjusted Rand Index (ARI)
  - Normalized Mutual Information (NMI)
    - Both provide a quantitative measure with an optimum of 1 and a value of 0 for unrelated clustering (though the ARI can become negative)
    - How to calculate?

```
from sklearn.metrics.cluster import adjusted_rand_score

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
fig, axes = plt.subplots(1, 4, figsize=(15, 3),
subplot_kw={'xticks': (), 'yticks': ()})
```

```
# make a list of algorithms to use
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2), DBSCAN()]

# create a random cluster assignment for reference
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# plot random assignment
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters, cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment - ARI: {:.2f}".format(adjusted_rand_score(y, random_clusters)))
for ax, algorithm in zip(axes[1:], algorithms):
    # plot the cluster assignments and cluster centers
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap="Paired", s=60)
    ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__, adjusted_rand_score(y, clusters)))
```

- A common mistake when evaluating clustering
  - to use accuracy_score instead of adjusted_rand_score, normalized_mutual_info_score
  - Reason: the value of cluster label is useless

```
from sklearn.metrics import accuracy_score
# these two labelings of points correspond to the same clustering
clusters1 = [0, 0, 1, 1, 0]
clusters2 = [1, 1, 0, 0, 1]
# accuracy is zero, as none of the labels are the same
print("Accuracy: {:.2f}".format(accuracy_score(clusters1, clusters2)))
# adjusted rand score is 1, as the clustering is exactly the same
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))
```

- Evaluating clustering without ground truth
  - Using metrics like ARI and NMI usually only helps in developing algorithms
  - Not in assessing success in an application
  - Specially according to these metrics and scores, we still don't know if there is any semantic meaning in the clustering
  - The only way to know whether the clustering corresponds to anything we are interested in is to analyze the clusters manually

Comparing Algorithms on Faces Dataset

- Applying different clustering algorithms to the labeled faces in the wide dataset
  - See if interesting structure can be found by any of them
  - Use eigenface rep. as produced by PCA with 100 components

```
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

- Analyzing the faces with DBSCAN

```
# apply DBSCAN with default parameters
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

  - Parameters: making `eps` higher or `min_samples` lower

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

- Even allowing cluster with only 3 samples, still everything labeled as noise
- Increasing eps to 15 – still only get one cluster
- Let's look at how many points are noises and core-samples

```
# Count number of points in all clusters and noise.
# bincount doesn't allow negative numbers, so we need to add 1.
# The first number in the result corresponds to noise points.
print("Number of points per cluster: {}".format(np.bincount(labels + 1)))
```

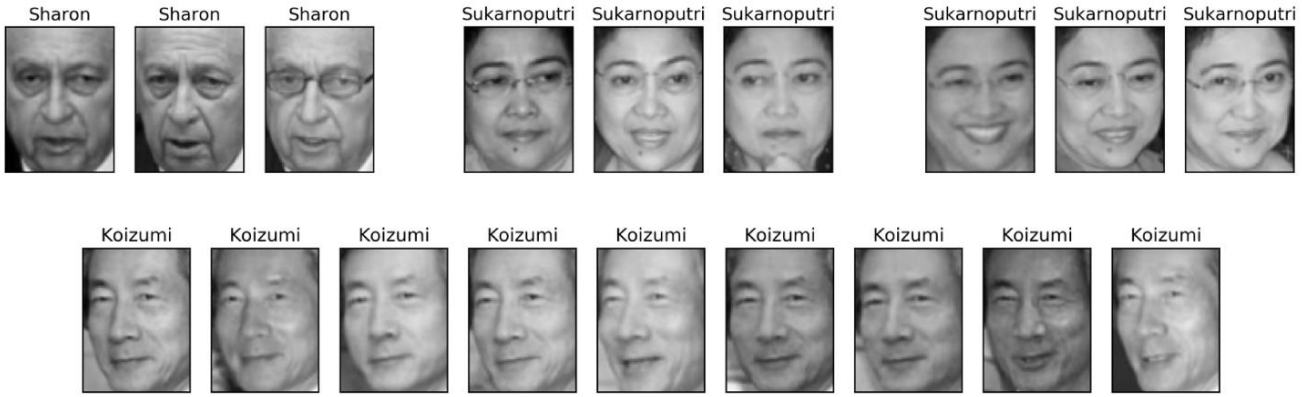
- We display all noises to have a check (called outlier detection)

```
image_shape = people.images[0].shape
noise = X_people[labels==-1]
fig, axes = plt.subplots(3, 9, subplot_kw={'xticks': (), 'yticks': ()},
figsize=(12, 4))
for image, ax in zip(noise, axes.ravel()):
    ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
```

- There is little we can do with the outliers, but it's good to know the reasons cause them
  - E.g., wearing hats, drinking or holding sth in front of their face
  - Let's have a look at what different values of eps result in

```
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Number of clusters: {}".format(len(np.unique(labels))))
    print("Cluster sizes: {}".format(np.bincount(labels + 1)))
```

  - The result for eps=7 look most interesting, with many small clusters – we then visualize all of the points in these 13 clusters



```
dbscan = DBSCAN(min_samples=3, eps=7)
labels = dbscan.fit_predict(X_pca)
for cluster in range(max(labels) + 1):
    mask = labels == cluster
    n_images = np.sum(mask)
    fig, axes = plt.subplots(1, n_images, figsize=(n_images * 1.5, 4), subplot_kw={'xticks': (), 'yticks': ()})
    for image, label, ax in zip(X_people[mask], y_people[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1])
```

  - Some of the clusters correspond to people with very distinct faces
  - Within each cluster, the orientation of the face is also quite fixed, as well as the facial expression
  - Some cluster contains multiple people, but they share a similar orientation and expression

- As you can see, we are doing a manual analysis here, which is different from the supervised learning based on R^2 score or accuracy

- Analyzing the faces dataset with *k*-means
  - Much more likely to create clusters of even size
  - We can start with a low number of clusters (e.g., 10)

```
# extract clusters with k-means
km = KMeans(n_clusters=10, random_state=0)
labels_km = km.fit_predict(X_pca)
print("Cluster sizes k-means: {}".format(np.bincount(labels_km)))
```

  - As you can see, the cluster sized from 64 to 386, which is quite different from the result of DBSCAN
  - We can further analyze the outcome of *k*-means by visualizing the cluster centers (very smooth versions of faces)

```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()}, figsize=(12, 4))
for center, ax in zip(km.cluster_centers_, axes.ravel()):
    ax.imshow(pca.inverse_transform(center).reshape(image_shape), vmin=0, vmax=1)
```

  - The clustering seems to pick up on different orientations of the face, different expressions (the third one); see closest samples

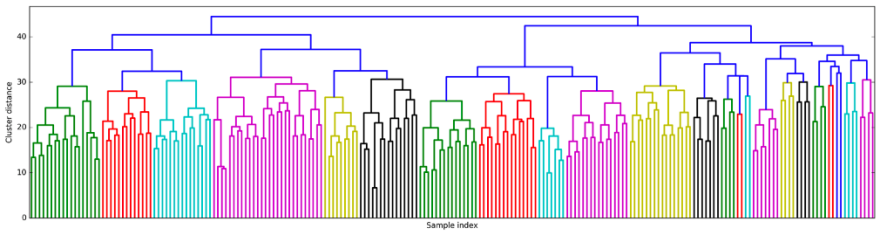
```
mglearn.plots.plot_kmeans_faces(km, pca, X_pca, X_people, y_people, people.target_names)
```

- Analyzing the faces dataset with agglomerative clustering
  - The same, starting from 10 clusters

```
# extract clusters with ward agglomerative clustering
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("Cluster sizes agglomerative clustering: {}".format(np.bincount(labels_agg)))
print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

  - The result is more uneven than *k*-means but more even than DBSCAN
  - ARI with a very low value means that the two clustering results of k-means and agglomerative clustering have little in common
  - We can plot the dendrogram but with limited depth

```
linkage_array = ward(X_pca)
# now we plot the dendrogram for the linkage_array; containing the distances between clusters
plt.figure(figsize=(20, 5))
dendrogram(linkage_array, p=7,
        truncate_mode='level', no_labels=True)
plt.xlabel("Sample index")    plt.ylabel("Cluster distance")
```



- Creating 10 clusters, we cut across the tree at the very top, where there are 10 vertical lines
- Let's visualize the 10 clusters
- Note that, there is no notation of cluster center in agglomerative clustering, we simply show the first few points in each cluster

```
n_clusters = 10
for cluster in range(n_clusters):
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 10, subplot_kw={'xticks': (), 'yticks': ()}, figsize=(15, 8))
    axes[0].set_ylabel(np.sum(mask))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask], labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1], fontdict={'fontsize': 9})
```

- While some of the clusters seem to have a semantic theme, many of them are too large to be actually homogeneous
- Generate more clusters (e.g. 40) to obtain more homogeneous clusters (result in sth like "Hussein" and "Smiling woman")

```
# extract clusters with ward agglomerative clustering
agglomerative = AgglomerativeClustering(n_clusters=40)
labels_agg = agglomerative.fit_predict(X_pca)
print("cluster sizes agglomerative clustering: {}".format(np.bincount(labels_agg)))
n_clusters = 40
for cluster in [10, 13, 19, 38, 39]: # hand-picked "interesting" clusters
        mask = labels_agg == cluster
        fig, axes = plt.subplots(1, 15, subplot_kw={'xticks': (), 'yticks': ()}, figsize=(15, 8))
        cluster_size = np.sum(mask)
        axes[0].set_ylabel("#{}: {}".format(cluster, cluster_size))
        for image, label, asdf, ax in zip(X_people[mask], y_people[mask], labels_agg[mask], axes):
                ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
                ax.set_title(people.target_names[label].split()[-1], fontdict={'fontsize': 9})
        for i in range(cluster_size, 15):
                axes[i].set_visible(False)
```

Summary of Clustering Methods

- Each algorithm has somewhat different strengths
  - *k*-means allows for a characterization of clusters using the cluster means; it can be considered a decomposition method
  - DBSCAN allows for the detection of "noise points" and allows for complex cluster shapes
  - Agglomerative clustering can provide a whole hierarchy of possible partitions
  - It is hard to quantify the usefulness of an unsupervised algorithm, though this shouldn't deter you from using them to gather insight from your data.