CLUSTERING

Definition:

Clustering is a Machine Learning technique that involves the grouping of data points. Given a set of data points, we can use a clustering algorithm to classify each data point into a specific group. In theory, data points that are in the same group should have similar properties and/or features, while data points in different groups should have highly dissimilar properties and/or features. Clustering is a method of unsupervised learning and is a common technique for statistical data analysis used in many fields.

In Data Science, we can use clustering analysis to gain some valuable insights from our data by seeing what groups the data points fall into when we apply a clustering algorithm.

1) K-Means Clustering:

Clustering (or cluster analysis) is a technique that allows us to find groups of similar objects, objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines. The k-means algorithm belongs to the category of prototype-based clustering. Prototype-based clustering means that each cluster is represented by a prototype, which can either be the **centroid** (average) of similar points with continuous features, or the **medoid** (the most representative or most frequently occurring point) in the case of categorical features.

Advantages:

K-Means is pretty fast, as all we're really doing is computing the distances between points and group centers; very few computations! It thus has a linear complexity O(n).

Disadvantages:

- You have to select how many groups/classes there are. This isn't always trivial and ideally with a clustering algorithm we'd want it to figure those out for us because the point of it is to gain some insight from the data.

- K-means also starts with a random choice of cluster centers and therefore it may yield different clustering results on different runs of the algorithm. Thus, the results may not be repeatable and lack consistency.

Syntax:

```
from sklearn.cluster import KMeans

X = inputs

model = KMeans(n_clusters=8, init='k-means++', n_init=10, max_iter=300,
tol=0.0001, precompute_distances='auto', random_state=None, copy_x=True,
n_jobs=None, algorithm='auto')

model.fit(X)
```

Parameters:

- n_clusters : int, optional, default: 8 -The number of clusters to form as well as the number of centroids to generate.
- init : {'k-means++', 'random' or an ndarray} - Method for initialization, defaults to 'k-means++':

  'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in k_init for more details.

  'random': choose k observations (rows) at random from data for the initial centroids.

- **n_init** : int, default: 10 -Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
- **max_iter** : int, default: 300 - Maximum number of iterations of the k-means algorithm for a single run.
- **tol** : float, default: 1e-4 - Relative tolerance with regards to inertia to declare convergence
- **precompute_distances** : {'auto', True, False} - Precompute distances (faster but takes more memory).

  auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision.

  True : always precompute distances

  False : never precompute distances

- **algorithm** : "auto", "full" or "elkan", default="auto"

  K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient by using the triangle inequality, but currently doesn't support sparse data. "auto" chooses "elkan" for dense data and "full" for sparse data.

Attributes:

- cluster_centers_ : array, [n_clusters, n_features] - Coordinates of cluster centers. If the algorithm stops before fully converging (see tol and max_iter), these will not be consistent with labels_.
- labels_ : - Labels of each point
- inertia_ : float - Sum of squared distances of samples to their closest cluster center.
- n_ier_ : int - Number of iterations run.

Methods:

fit(X[, y, sample_weight])	Compute k-means clustering.
fit_predict(X[, y, sample_weight])	Compute cluster centers and predict cluster index for each sample.
fit_transform(X[, y, sample_weight])	Compute clustering and transform X to cluster-distance space.
get_params([deep])	Get parameters for this estimator.
predict(X[, sample_weight])	Predict the closest cluster each sample in X belongs to.
score(X[, y, sample_weight])	Opposite of the value of X on the K-means objective.
transform(X)	Transform X to a cluster-distance space.

Example:

```
from sklearn.cluster import KMeans
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0],[10, 2], [10, 4], [10, 0]])
model = KMeans(n_clusters=2, random_state=0).fit(X)
model.labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
model.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32)
model.cluster_centers_
array([[10.,  2.],[ 1.,  2.]])
```

2) Mean-Shift Clustering:

Mean shift clustering is a sliding-window-based algorithm that attempts to find dense areas of data points. It is a centroid-based algorithm meaning that the goal is to locate the center points of each group/class, which works by updating candidates for center points to be the mean of the points within the sliding-window. These candidate windows are then filtered in a post-processing stage to eliminate near-duplicates, forming the final set of center points and their corresponding groups.

Advantages:

- In contrast to K-means clustering there is no need to select the number of clusters as mean-shift automatically discovers this. That's a massive advantage.
- The fact that the cluster centers converge towards the points of maximum density is also quite desirable as it is quite intuitive to understand and fits well in a naturally data-driven sense.

Disadvantages:

- The drawback is that the selection of the window size/radius "r" can be non-trivial.

Syntax:

```
from sklearn.cluster import MeanShift
X=inputs
model = MeanShift(bandwidth=None, seeds=None, bin_seeding=False, min_bin_freq=1, cluster_all=True, n_jobs=None)
model.fit(X)
```

Parameters:

- bandwidth : float, optional - Bandwidth used in the RBF kernel.
- seeds : array, shape=[n_samples, n_features], optional - Seeds used to initialize kernels. If not set, the seeds are calculated by clustering.get_bin_seeds with bandwidth as the grid size and default values for other parameters.
- bin_seeding : boolean, optional- If true, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to True will speed up the algorithm because fewer seeds will be initialized. default value: False Ignored if seeds argument is not None
- min_bin_freq : int, optional -To speed up the algorithm, accept only those bins with at least min_bin_freq points as seeds. If not defined, set to 1.
- cluster_all : boolean, default True -If true, then all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If false, then orphans are given cluster label -1.
- n_jobs : int or None, optional (default=None) -The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

Attributes:

- cluster_centers_ : array, [n_clusters, n_features] - Coordinates of cluster centers.
- labels_ : - Labels of each point.

Methods:

fit(X[, y])	Perform clustering.
fit_predict(X[, y])	Performs clustering on X and returns cluster labels.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict the closest cluster each sample in X belongs to.

Example:

```
from sklearn.cluster import MeanShift
import numpy as np
X = np.array([[1, 1], [2, 1], [1, 0],[4, 7], [3, 5], [3, 6]])
clustering = MeanShift(bandwidth=2).fit(X)
clustering.labels_
array([1, 1, 1, 0, 0, 0])
clustering.predict([[0, 0], [5, 5]])
array([1, 0])
clustering
MeanShift(bandwidth=2, bin_seeding=False, cluster_all=True,
min_bin_freq=1,n_jobs=None, seeds=None)
```

3) DBSCAN:

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a popular clustering algorithm used as an replacement to K-means in predictive analytics. It doesn't require that you input the number of clusters in order to run. But in exchange, you have to tune two other parameters.

The scikit-learn implementation provides a default for the eps and min_samples parameters, but you're generally expected to tune those. The eps parameter is the maximum distance between two data points to be considered in the same neighborhood. The min_samples parameter is the minimum amount of data points in a neighborhood to be considered a cluster.

Advantage:

- DBSCAN does not require a pe-set number of clusters at all. It also identifies outliers as noises unlike mean-shift which simply throws them into a cluster even if the data point is very different.
- DBSCAN is able to find arbitrarily sized and arbitrarily shaped clusters quite well.

Disadvantages:

- DBSCAN doesn't perform as well as others when the clusters are of varying density. This is because the setting of the distance threshold ε and minPoints for identifying the neighborhood points will vary from cluster to cluster when the density varies.
- This drawback also occurs with very high-dimensional data since again the distance threshold ε becomes challenging to estimate.

Syntax:

```
from sklearn.cluster import DBSCAN
X=inputs
model = DBSCAN(eps=0.5, min_samples=5, metric='euclidean',
metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=None)\
model.fit()
```

Parameter:

- **eps** : float, optional - The maximum distance between two samples for them to be considered as in the same neighborhood.

- **min_samples :** int, optional - The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself.

- **metric :** string, or callable - The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must

be one of the options allowed by sklearn.metrics.pairwise_distances for its metric parameter. If metric is “precomputed”, X is assumed to be a distance matrix and must be square. X may be a sparse matrix, in which case only “nonzero” elements may be considered neighbors for DBSCAN.

- **metric_params** : dict, optional - Additional keyword arguments for the metric function.

- **leaf_size :** int, optional (default = 30) - Leaf size passed to BallTree or cKDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

- **p :** float, optional - The power of the Minkowski metric to be used to calculate distance between points.

Attribute:

- **core_sample_indices_ :** array, shape = [n_core_samples] - Indices of core samples.
- **components_ :** array, shape = [n_core_samples, n_features] - Copy of each core sample found by training.
- **labels_ :** array, shape = [n_samples] - Cluster labels for each point in the dataset given to fit(). Noisy samples are given the label -1.

Methods:

fit(X[, y, sample_weight])	Perform DBSCAN clustering from features or distance matrix.
fit_predict(X[, y, sample_weight])	Performs clustering on X and returns cluster labels.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

Examples:

```
from sklearn.cluster import DBSCAN
import numpy as np
X = np.array([[1, 2], [2, 2], [2, 3],[8, 7], [8, 8], [25, 80]])
clustering = DBSCAN(eps=3, min_samples=2).fit(X)
clustering.labels_
array([ 0,  0,  0,  1,  1, -1])
clustering
DBSCAN(algorithm='auto', eps=3, leaf_size=30, metric='euclidean',
    metric_params=None, min_samples=2, n_jobs=None, p=None)
```

4) Hierarchical clustering:

Hierarchical clustering algorithms actually fall into 2 categories: **top-down** or bottom-up. **Bottom-up** algorithms treat each data point as a single cluster at the outset and then successively merge (or **agglomerate**) pairs of clusters until all clusters have been merged into a single cluster that contains all data points. Bottom-up hierarchical clustering is therefore called hierarchical agglomerative clustering or HAC. This hierarchy of clusters is represented as a tree (or **dendrogram**). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.

Advantages:

- Hierarchical clustering does not require us to specify the number of clusters and we can even select which number of clusters looks best since we are building a tree.

- Additionally, the algorithm is not sensitive to the choice of distance metric; all of them tend to work equally well whereas with other clustering algorithms, the choice of distance metric is critical.
- A particularly good use case of hierarchical clustering methods is when the underlying data has a hierarchical structure and you want to recover the hierarchy; other clustering algorithms can't do this.
- These advantages of hierarchical clustering come at the cost of lower efficiency, as it has a time complexity of $O(n^3)$, unlike the linear complexity of K-Means and GMM.

Syntax:

```
from sklearn.cluster import AgglomerativeClustering
X=inputs
model = AgglomerativeClustering(n_clusters=2, affinity='euclidean', memory=None, connectivity=None, compute_full_tree='auto', linkage='ward', pooling_func='deprecated')
model.fit()
```

Parameters:

- **n_clusters** : int, default=2 - The number of clusters to find.

- **affinity :** string or callable, default: "euclidean" - Metric used to compute the linkage. Can be "euclidean", "l1", "l2", "manhattan", "cosine", or 'precomputed'. If linkage is "ward", only "euclidean" is accepted.

- **memory :** None, str or object with the joblib.Memory interface, optional - Used to cache the output of the computation of the tree. By default, no caching is done. If a string is given, it is the path to the caching directory.

- **connectivity** : array-like or callable, optional - Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors_graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.

- **compute_full_tree** : bool or 'auto' (optional) - Stop early the construction of the tree at n_clusters. This is useful to decrease computation time if the number of clusters is not small compared to the number of samples. This option is useful only when specifying a connectivity matrix. Note also that when varying the number of clusters and using caching, it may be advantageous to compute the full tree.

- **linkage :** {"ward", "complete", "average", "single"}, optional (default="ward") - Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.

    - ward minimizes the variance of the clusters being merged.
    - average uses the average of the distances of each observation of the two sets.
    - complete or maximum linkage uses the maximum distances between all observations of the two sets.
    - single uses the minimum of the distances between all observations of the two sets.

Methods:

fit(X[, y])	Fit the hierarchical clustering on the data
fit_predict(X[, y])	Performs clustering on X and returns cluster labels.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.

Attributes:

- labels_ : array [n_samples] - cluster labels for each point

- n_leaves_ : int - Number of leaves in the hierarchical tree.

- n_components_ : int - The estimated number of connected components in the graph.

- children_ : array-like, shape (n_samples-1, 2) - The children of each non-leaf node. Values less than n_samples correspond to leaves of the tree which are the original samples. A node i greater than or equal to n_samples is a non-leaf node and has children children_[i - n_samples]. Alternatively at the i-th iteration, children[i][0] and children[i][1] are merged to form node n_samples + i

Example:

```
from sklearn.cluster import AgglomerativeClustering
import numpy as np
X = np.array([[1, 2], [1, 4], [1, 0],[4, 2], [4, 4], [4, 0]])
clustering = AgglomerativeClustering().fit(X)clustering
AgglomerativeClustering(affinity='euclidean',
compute_full_tree='auto',connectivity=None, linkage='ward', memory=None,
n_clusters=2,pooling_func='deprecated')clustering.labels_
array([1, 1, 1, 0, 0, 0])
```

5) Gaussian Mixture Models(GMM):

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

Gaussian Mixture Models (GMMs) give us more flexibility than K-Means. With GMMs we assume that the data points are Gaussian distributed; this is a less restrictive assumption than saying they are circular by using the mean. That way, we have two parameters to describe the shape of the clusters: the mean and the standard deviation! Taking an example in two dimensions, this means that the clusters can take any kind of elliptical shape (since we have standard deviation in both the x and y directions). Thus, each Gaussian distribution is assigned to a single cluster.

In order to find the parameters of the Gaussian for each cluster (e.g the mean and standard deviation) we will use an optimization algorithm called Expectation–Maximization (EM). Take a look at the graphic below as an illustration of the Gaussians being fitted to the clusters. Then we can proceed on to the process of Expectation–Maximization clustering using GMMs.

Advantages:

- Firstly GMMs are a lot more flexible in terms of cluster covariance than K-Means; due to the standard deviation parameter, the clusters can take on any ellipse shape, rather than being restricted to circles. K-Means is actually a special case of GMM in which each cluster's covariance along all dimensions approaches 0.
- Secondly, since GMMs use probabilities, they can have multiple clusters per data point. So if a data point is in the middle of two overlapping clusters, we can simply define its class by saying it belongs X-percent to class 1 and Y-percent to class 2. I.e GMMs support mixed membership.

Syntax:

```
from sklearn.mixture import GaussianMixture
X=inputs
model = GaussianMixture(n_components=1, covariance_type='full', tol=0.001, reg_covar=1e-06, max_iter=100, n_init=1, init_params='kmeans', weights_init=None, means_init=None, precisions_init=None, random_state=None, warm_start=False, verbose=0, verbose_interval=10)
model.fit(X)
```

Parameters:

- n_components : int, defaults to 1. - The number of mixture components.
- **covariance_type :** {'full' (default), 'tied', 'diag', 'spherical'} - String describing the type of covariance parameters to use. Must be one of:
  - 'full'- each component has its own general covariance matrix
  - 'tied' - all components share the same general covariance matrix
  - 'diag' - each component has its own diagonal covariance matrix
  - 'spherical' - each component has its own single variance

Attributes:

- **weights_** : array-like, shape (n_components,) - The weights of each mixture components.
- **means_** : array-like, shape (n_components, n_features) - The mean of each mixture component.
- **covariances_** : array-like - The covariance of each mixture component. The shape depends on covariance_type:
- **precisions_** : array-like - The precision matrices for each component in the mixture. A precision matrix is the inverse of a covariance matrix. A covariance matrix is symmetric positive definite so the mixture of Gaussian can be equivalently parameterized by the precision matrices. Storing the precision matrices instead of the covariance matrices makes it more efficient to compute the log-likelihood of new samples at test time. The shape depends on covariance_type: