Machine Learning using Python Lab: Implementation of Clustering Algorithms

Neelotpal Chakraborty

Department of Computer Science and Engineering

Jadavpur University

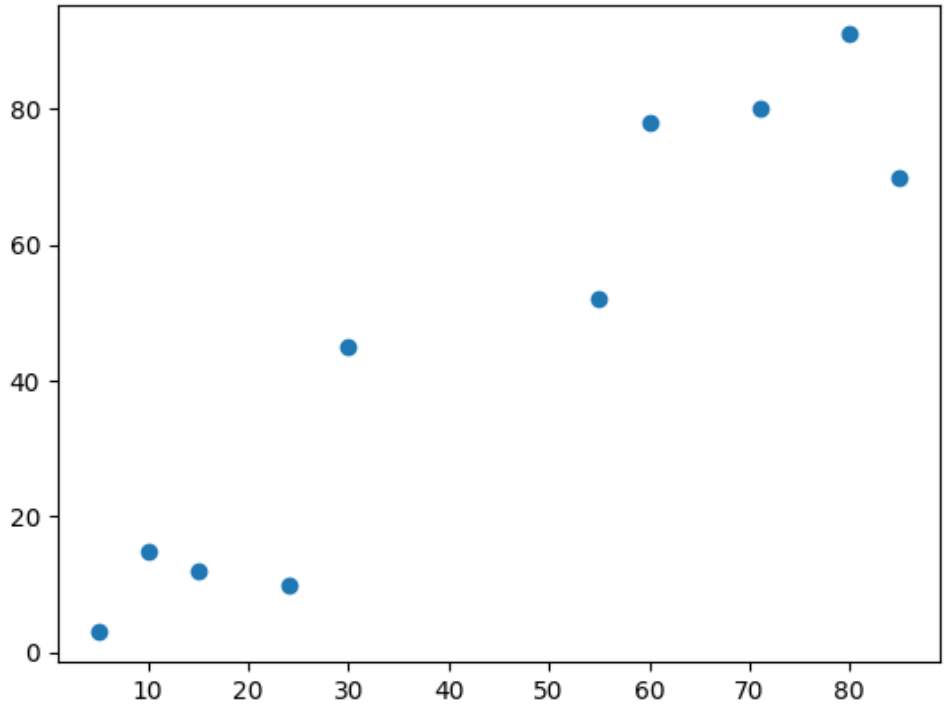
Partitioning based Clustering

- K-Means
- K-Medoids/Partitioning Around Medoids (PAM)

Clustering: K-means

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans

X = np.array([[5,3],
      [10,15],
      [15,12],
      [24,10],
      [30,45],
      [85,70],
      [71,80],
      [60,78],
      [55,52],
      [80,91],])
plt.scatter(X[:,0],X[:,1], label='True Position')
plt.show()
```

Output:

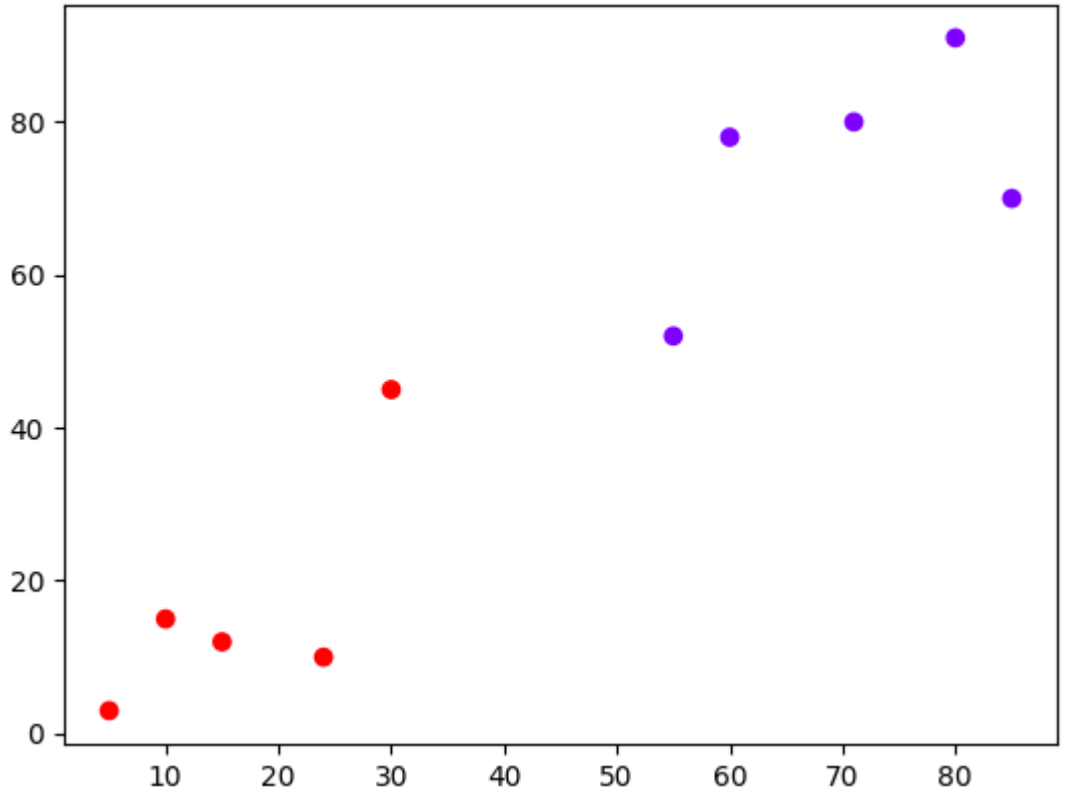


Clustering: K-means

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)

print("K-Means CLuster Centers:")
print(kmeans.cluster_centers_)

print("Cluster Labels:")
print(kmeans.labels_)

plt.scatter(X[:,0],X[:,1], c=kmeans.labels_, cmap='rainbow')
plt.show()
```

Output:

```
K-Means CLuster Centers:
[[70.2 74.2]
 [16.8 17. ]]
Cluster Labels:
[1 1 1 1 1 0 0 0 0 0]
```



Clustering (Parameters): K-means

```
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='auto')
```

Clustering (Parameters): K-means

n_clusters : *int, default=8*

The number of clusters to form as well as the number of centroids to generate.

init : *{'k-means++', 'random'}, callable or array-like of shape (n_clusters, n_features), default='k-means++'*

Method for initialization:

'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 `n_clusters` observations (rows) at random from data for the initial centroids.

If an array is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

If a callable is passed, it should take arguments X, n_clusters and a random state and return an initialization.

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.

Clustering (Parameters): K-means

tol : *float, default=1e-4*

Relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence.

precompute_distances : *{'auto', True, False}, default='auto'*

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.

Clustering (Parameters): K-means

verbose : *int, default=0*

Verbosity mode.

random_state : *int, RandomState instance or None, default=None*

Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See Glossary.

copy_x : *bool, default=True*

When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean. Note that if the original data is not C-contiguous, a copy will be made even if copy_x is False. If the original data is sparse, but not in CSR format, a copy will be made even if copy_x is False.

n_jobs : *int, default=None*

The number of OpenMP threads to use for the computation. Parallelism is sample-wise on the main cython loop which assigns each sample to its closest center.

`None` or `-1` means using all processors.

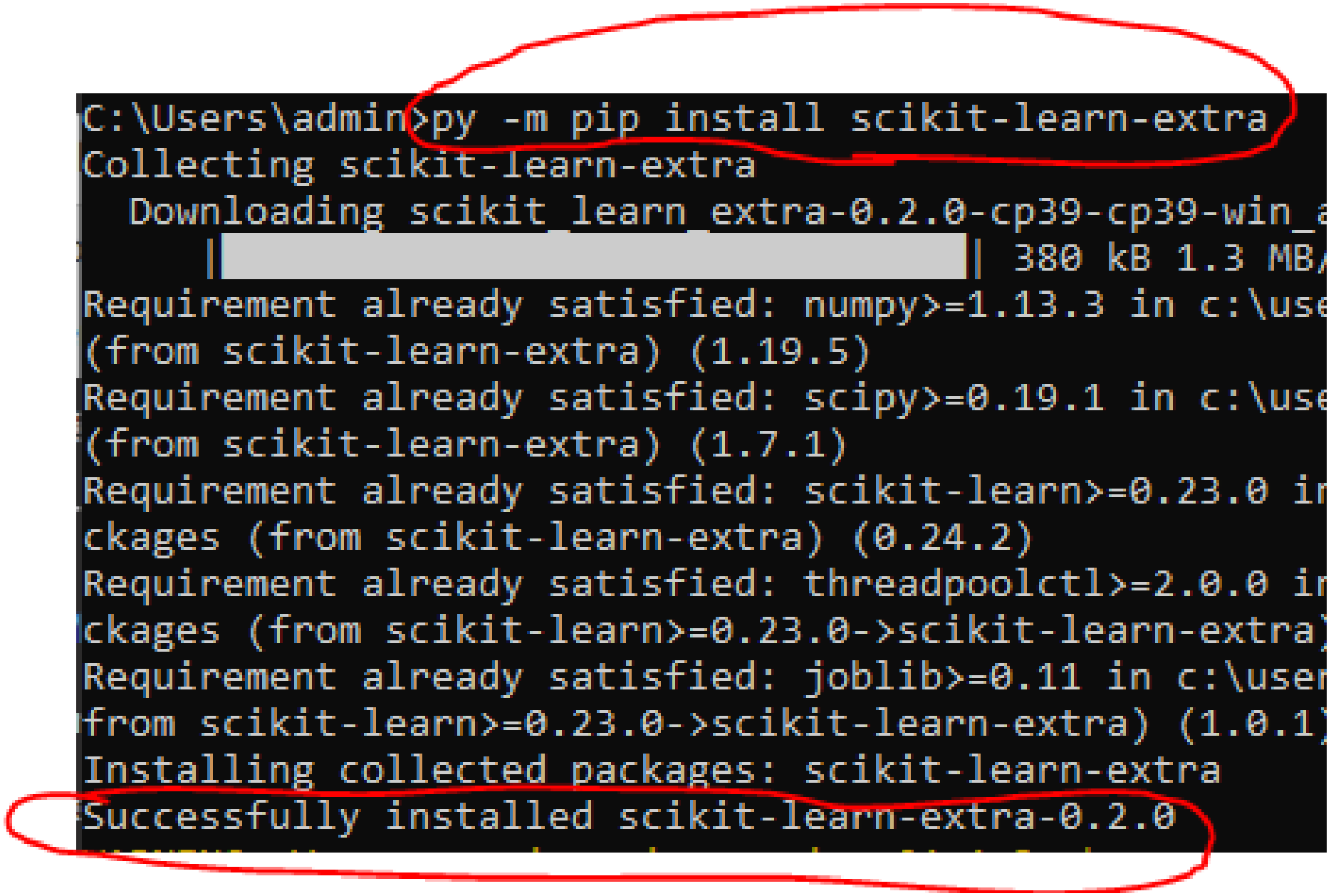
Clustering (Parameters): K-means

algorithm : *{"auto", "full", "elkan"}, default="auto"*

K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient on data with well-defined clusters, by using the triangle inequality. However it's more memory intensive due to the allocation of an extra array of shape (n_samples, n_clusters).

For now "auto" (kept for backward compatibiliy) chooses "elkan" but it might change in the future for a better heuristic.

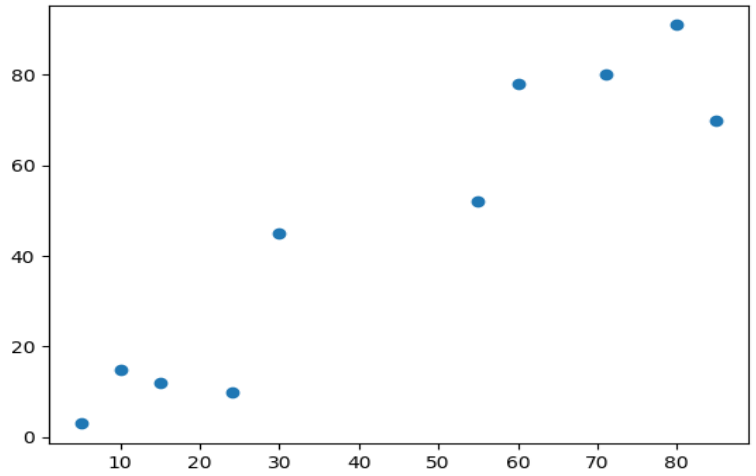
Clustering: K-Medoids



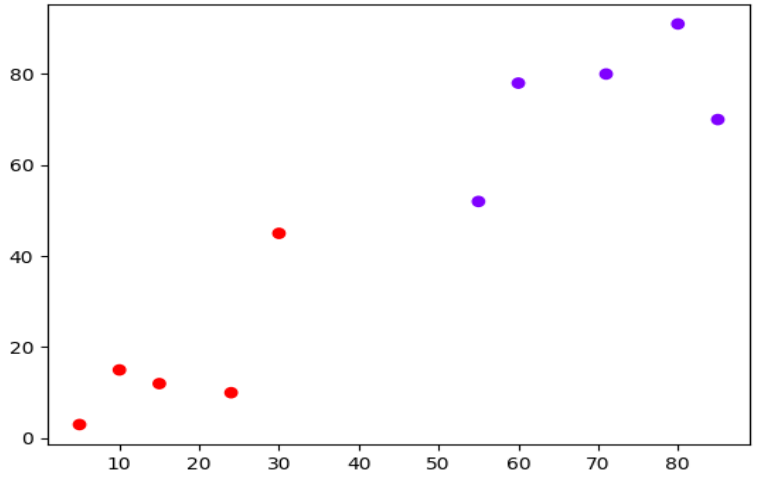
Clustering: K-Medoids

```
import matplotlib.pyplot as plt
import numpy as np

from sklearn_extra.cluster import KMedoids

X = np.array([[5,3],
      [10,15],
      [15,12],
      [24,10],
      [30,45],
      [85,70],
      [71,80],
      [60,78],
      [55,52],
      [80,91],])
plt.scatter(X[:,0],X[:,1], label='True Position')
plt.show()

kmedoid = KMedoids(n_clusters=2, random_state=0).fit(X)


print("K-Medoids CLuster Centers:")
print(kmedoid.cluster_centers_)

print("Cluster Labels:")
print(kmedoid.labels_)

plt.scatter(X[:,0],X[:,1], c=kmedoid.labels_, cmap='rainbow')
plt.show()
```

Output:




```
K-Medoids CLuster Centers:
[[71 80]
 [15 12]]
Cluster Labels:
[1 1 1 1 1 0 0 0 0 0]
```



Clustering (Parameters): K-Medoids

```
class sklearn_extra.cluster.KMedoids(n_clusters=8, metric='euclidean', method='alternate',
init='heuristic', max_iter=300, random_state=None)      [source]
```

n_clusters : int, optional, default: 8

The number of clusters to form as well as the number of medoids to generate.

metric : string, or callable, optional, default: 'euclidean'

What distance metric to use. See :func:metrics.pairwise_distances metric can be 'precomputed', the user must then feed the fit method with a precomputed kernel matrix and not the design matrix X.

method : {'alternate', 'pam'}, default: 'alternate'

Which algorithm to use. 'alternate' is faster while 'pam' is more accurate.

Clustering (Parameters): K-Medoids

init : {'random', 'heuristic', 'k-medoids++', 'build'}, optional, default: 'build'

Specify medoid initialization method. 'random' selects n_clusters elements from the dataset. 'heuristic' picks the n_clusters points with the smallest sum distance to every other point. 'k-medoids++' follows an approach based on k-means++_, and in general, gives initial medoids which are more separated than those generated by the other methods. 'build' is a greedy initialization of the medoids used in the original PAM algorithm. Often 'build' is more efficient but slower than other initializations on big datasets and it is also very non-robust, if there are outliers in the dataset, use another initialization.

max_iter : int, optional, default

Specify the maximum number of iterations when fitting. It can be zero in which case only the initialization is computed which may be suitable for large datasets when the initialization is sufficiently efficient (i.e. for 'build' init).

random_state : int, RandomState instance or None, optional

Specify random state for the random number generator. Used to initialise medoids when init='random'.

Clustering (Parameters): K-Medoids

For more details, explore the link below:

https://scikit-learn-extra.readthedocs.io/en/stable/generated/sklearn_extra.cluster.KMedoids.html

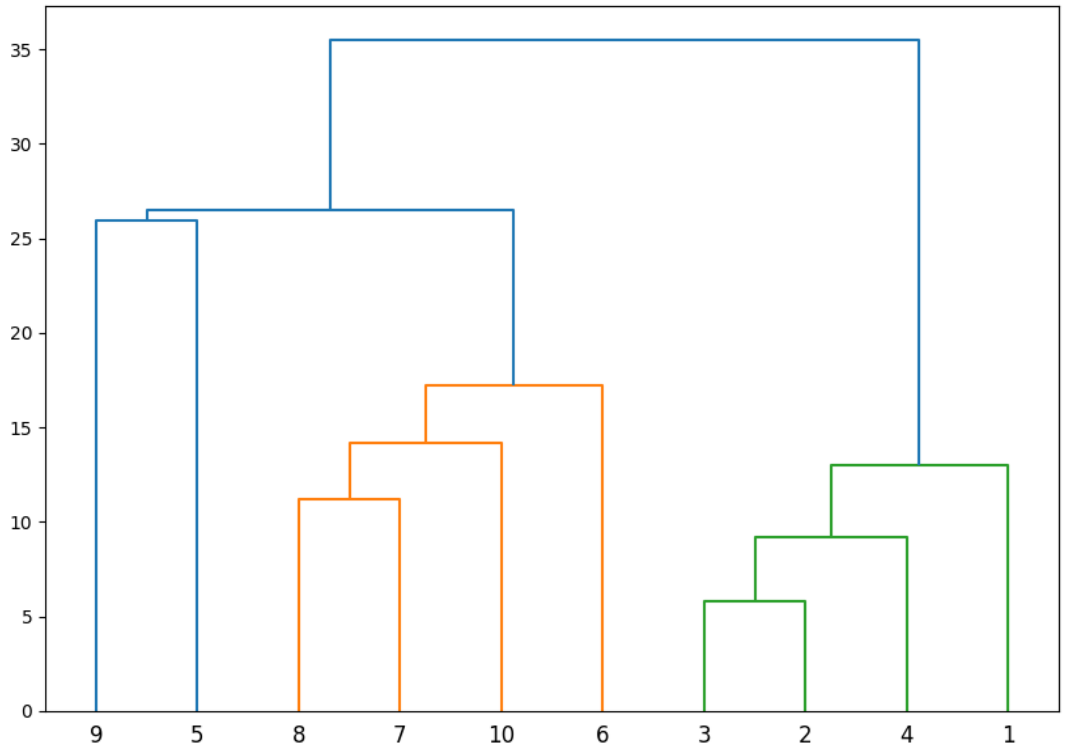
Hierarchical based Clustering

- Dendrogram
- Agglomerative Nesting (AGNES)
- Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH)

Clustering: Dendrogram

```
# Hierarchical Clustering

from scipy.cluster.hierarchy import dendrogram, linkage
from matplotlib import pyplot as plt

linked = linkage(X, 'single')

labelList = range(1, 11)

plt.figure(figsize=(10, 7))
dendrogram(linked,
            orientation='top',
            labels=labelList,
            distance_sort='descending',
            show_leaf_counts=True)
plt.show()
```

Output:



Clustering (Parameters): Dendrogram

```
scipy.cluster.hierarchy.dendrogram(Z, p=30, truncate_mode=None, color_threshold=None,
get_leaves=True, orientation='top', labels=None, count_sort=False, distance_sort=False,
show_leaf_counts=True, no_plot=False, no_labels=False, leaf_font_size=None,
leaf_rotation=None, leaf_label_func=None, show_contracted=False, link_color_func=None,
ax=None, above_threshold_color='C0')                                          [source]
```

Clustering (Parameters): Dendrogram

Z : *ndarray*

The linkage matrix encoding the hierarchical clustering to render as a dendrogram. See the `linkage` function for more information on the format of `Z`.

p : *int, optional*

The `p` parameter for `truncate_mode`.

truncate_mode : *str, optional*

The dendrogram can be hard to read when the original observation matrix from which the linkage is derived is large. Truncation is used to condense the dendrogram. There are several modes:

`None`

No truncation is performed (default). Note: `'none'` is an alias for `None` that's kept for backward compatibility.

`'lastp'`

The last `p` non-singleton clusters formed in the linkage are the only non-leaf nodes in the linkage; they correspond to rows `Z[n-p-2:end]` in `Z`. All other non-singleton clusters are contracted into leaf nodes.

`'level'`

No more than `p` levels of the dendrogram tree are displayed. A "level" includes all nodes with `p` merges from the final merge.

Note: `'mtica'` is an alias for `'level'` that's kept for backward compatibility.

Clustering (Parameters): Dendrogram

color_threshold : *double, optional*

For brevity, let t be the `color_threshold`. Colors all the descendent links below a cluster node k the same color if k is the first node below the cut threshold t. All links connecting nodes with distances greater than or equal to the threshold are colored with de default matplotlib color `'C0'`. If t is less than or equal to zero, all nodes are colored `'C0'`. If `color_threshold` is None or 'default', corresponding with MATLAB(TM) behavior, the threshold is set to `0.7*max(Z[:,2])`.

get_leaves : *bool, optional*

Includes a list `R['leaves']=H` in the result dictionary. For each i, `H[i] == j`, cluster node `j` appears in position `i` in the left-to-right traversal of the leaves, where $j < 2n - 1$ and $i < n$.

orientation : *str, optional*

The direction to plot the dendrogram, which can be any of the following strings:

`'top'`

Plots the root at the top, and plot descendent links going downwards. (default).

`'bottom'`

Plots the root at the bottom, and plot descendent links going upwards.

`'left'`

Plots the root at the left, and plot descendent links going right.

`'right'`

Plots the root at the right, and plot descendent links going left.

Clustering (Parameters): Dendrogram

labels : *ndarray, optional*

By default, `labels` is None so the index of the original observation is used to label the leaf nodes. Otherwise, this is an n-sized sequence, with `n == Z.shape[0] + 1`. The `labels[i]` value is the text to put under the i th leaf node only if it corresponds to an original observation and not a non-singleton cluster.

count_sort : *str or bool, optional*

For each node n, the order (visually, from left-to-right) n's two descendent links are plotted is determined by this parameter, which can be any of the following values:

`False`

Nothing is done.

`'ascending'` or `True`

The child with the minimum number of original objects in its cluster is plotted first.

`'descending'`

The child with the maximum number of original objects in its cluster is plotted first.

Note, `distance_sort` and `count_sort` cannot both be True.

distance_sort : *str or bool, optional*

For each node n, the order (visually, from left-to-right) n's two descendent links are plotted is determined by this parameter, which can be any of the following values:

`False`

Nothing is done.

`'ascending'` or `True`

The child with the minimum distance between its direct descendents is plotted first.

`'descending'`

The child with the maximum distance between its direct descendents is plotted first.

Note `distance_sort` and `count_sort` cannot both be True.

Clustering (Parameters): Dendrogram

show_leaf_counts : *bool, optional*

When True, leaf nodes representing $k > 1$ original observation are labeled with the number of observations they contain in parentheses.

no_plot : *bool, optional*

When True, the final rendering is not performed. This is useful if only the data structures computed for the rendering are needed or if matplotlib is not available.

no_labels : *bool, optional*

When True, no labels appear next to the leaf nodes in the rendering of the dendrogram.

leaf_rotation : *double, optional*

Specifies the angle (in degrees) to rotate the leaf labels. When unspecified, the rotation is based on the number of nodes in the dendrogram (default is 0).

leaf_font_size : *int, optional*

Specifies the font size (in points) of the leaf labels. When unspecified, the size based on the number of nodes in the dendrogram.

leaf_label_func : *lambda or function, optional*

When `leaf_label_func` is a callable function, for each leaf with cluster index $k < 2n - 1$. The function is expected to return a string with the label for the leaf.
Indices $k < n$ correspond to original observations while indices $k \geq n$ correspond to non-singleton clusters.

Clustering (Parameters): Dendrogram

show_contracted : *bool, optional*

When True the heights of non-singleton nodes contracted into a leaf node are plotted as crosses along the link connecting that leaf node. This really is only useful when truncation is used (see `truncate_mode` parameter).

link_color_func : *callable, optional*

If given, *link_color_function* is called with each non-singleton id corresponding to each U-shaped link it will paint. The function is expected to return the color to paint the link, encoded as a matplotlib color string code. For example:

```
dendrogram(Z, link_color_func=lambda k: colors[k])
```

colors the direct links below each untruncated non-singleton node `k` using `colors[k]`.

ax : *matplotlib Axes instance, optional*

If None and *no_plot* is not True, the dendrogram will be plotted on the current axes. Otherwise if *no_plot* is not True the dendrogram will be plotted on the given `Axes` instance. This can be useful if the dendrogram is part of a more complex figure.

above_threshold_color : *str, optional*

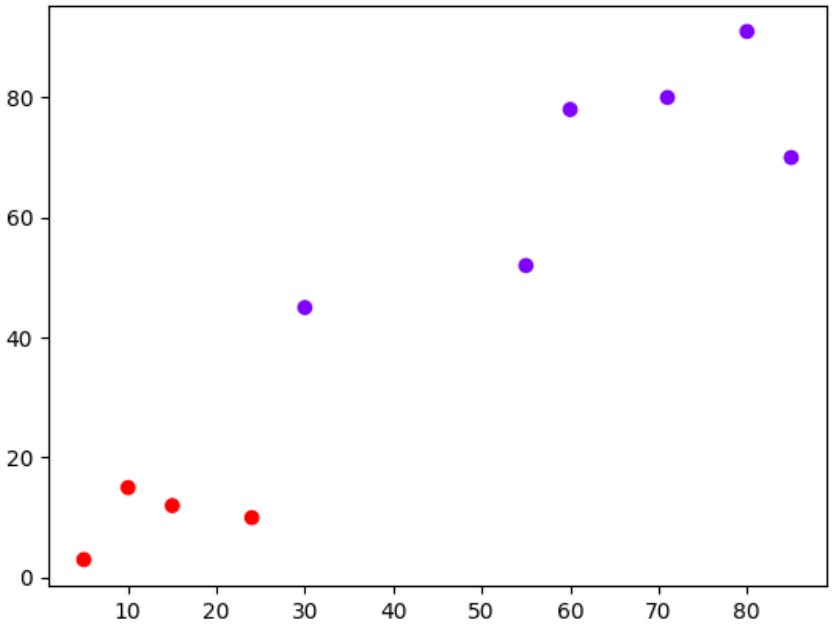
This matplotlib color string sets the color of the links above the color_threshold. The default is `'C0'`.

Clustering: Agglomerative

```
# Agglomerative Clustering

from sklearn.cluster import AgglomerativeClustering

cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')
cluster.fit_predict(X)
print("Cluster Labels:")
print(cluster.labels_)

plt.scatter(X[:,0],X[:,1], c=cluster.labels_, cmap='rainbow')
plt.show()
```

Output:

```
Cluster Labels:
[1 1 1 1 0 0 0 0 0 0]
```



Clustering (Parameters): Agglomerative

Agglomerative Clustering.

Recursively merges pair of clusters of sample data; uses linkage distance.

```
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)
```

Clustering (Parameters): Agglomerative

n_clusters : *int or None, default=2*

The number of clusters to find. It must be `None` if `distance_threshold` is not `None`.

affinity : *str 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. If "precomputed", a distance matrix (instead of a similarity matrix) is needed as input for the fit method.

memory : *str or object with the joblib.Memory interface, default=None*

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, default=None*

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.

Clustering (Parameters): Agglomerative

compute_full_tree : *'auto' or bool, default='auto'*

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. It must be `True` if `distance_threshold` is not `None`. By default `compute_full_tree` is "auto", which is equivalent to `True` when `distance_threshold` is not `None` or that `n_clusters` is inferior to the maximum between 100 or `0.02 * n_samples`. Otherwise, "auto" is equivalent to `False`.

linkage : *{'ward', 'complete', 'average', 'single'}, 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.

Clustering (Parameters): Agglomerative

distance_threshold : *float, default=None*

The linkage distance threshold above which, clusters will not be merged. If not `None`, `n_clusters` must be `None` and `compute_full_tree` must be `True`.

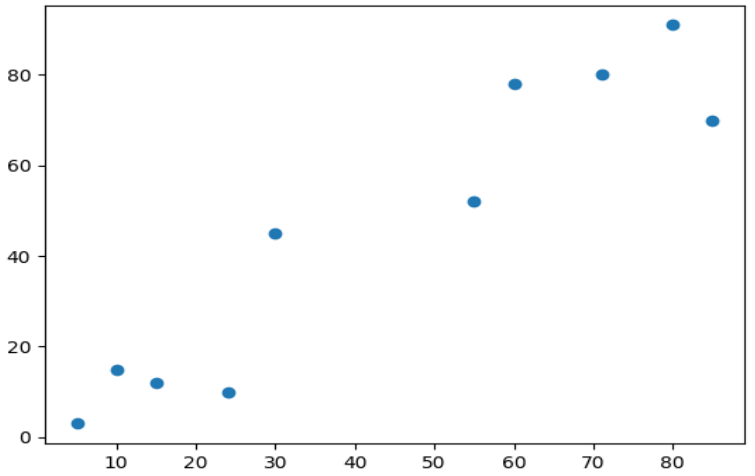
compute_distances : *bool, default=False*

Computes distances between clusters even if `distance_threshold` is not used. This can be used to make dendrogram visualization, but introduces a computational and memory overhead.

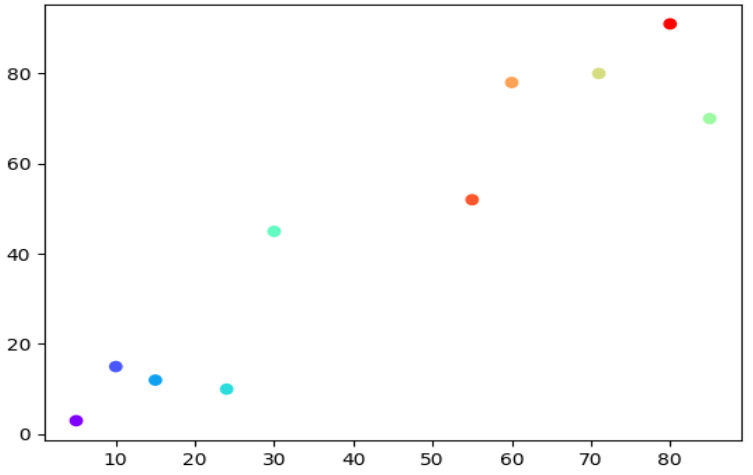
Clustering: BIRCH

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import Birch

X = np.array([[5,3],
      [10,15],
      [15,12],
      [24,10],
      [30,45],
      [85,70],
      [71,80],
      [60,78],
      [55,52],
      [80,91],])
plt.scatter(X[:,0],X[:,1], label='True Position')
plt.show()

birch = Birch(n_clusters=None)
birch.fit(X)

print("Cluster Labels:")
print(birch.labels_)

plt.scatter(X[:,0],X[:,1], c=birch.labels_, cmap='rainbow')
plt.show()
```

Output:



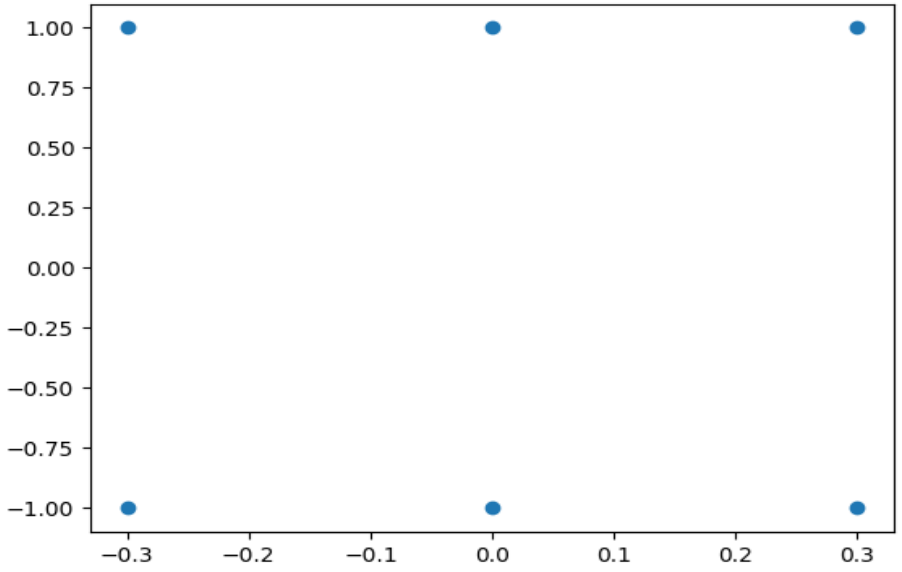

```
Cluster Labels:
[0 1 2 3 4 5 6 7 8 9]
```



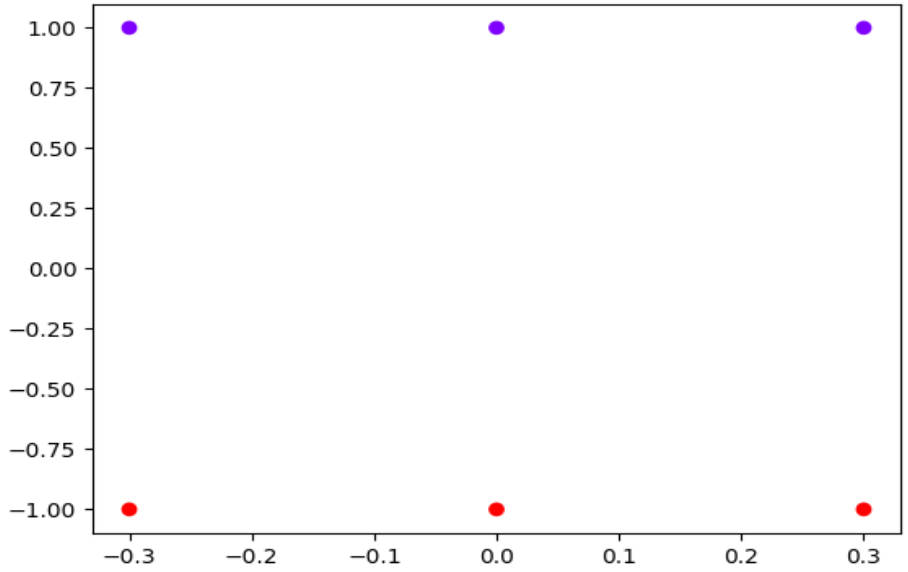
Clustering: BIRCH

```
X = np.array([[0, 1],
              [0.3, 1],
              [-0.3, 1],
              [0, -1],[0.3, -1],
              [-0.3, -1]])
```

Output:




```
Cluster Labels:
[0 0 0 1 1 1]
```



Clustering (Parameters): BIRCH

It is a memory-efficient, online-learning algorithm provided as an alternative to `MiniBatchKMeans`. It constructs a tree data structure with the cluster centroids being read off the leaf. These can be either the final cluster centroids or can be provided as input to another clustering algorithm such as `AgglomerativeClustering`.

class sklearn.cluster.Birch(*, *threshold=0.5*, *branching_factor=50*, *n_clusters=3*, *compute_labels=True*, *copy=True*)

Clustering (Parameters): BIRCH

threshold : *float, default=0.5*

The radius of the subcluster obtained by merging a new sample and the closest subcluster should be lesser than the threshold. Otherwise a new subcluster is started. Setting this value to be very low promotes splitting and vice-versa.

branching_factor : *int, default=50*

Maximum number of CF subclusters in each node. If a new samples enters such that the number of subclusters exceed the branching_factor then that node is split into two nodes with the subclusters redistributed in each. The parent subcluster of that node is removed and two new subclusters are added as parents of the 2 split nodes.

n_clusters : *int, instance of sklearn.cluster model, default=3*

Number of clusters after the final clustering step, which treats the subclusters from the leaves as new samples.

- `None` : the final clustering step is not performed and the subclusters are returned as they are.
- `sklearn.cluster` Estimator : If a model is provided, the model is fit treating the subclusters as new samples and the initial data is mapped to the label of the closest subcluster.
- `int` : the model fit is `AgglomerativeClustering` with `n_clusters` set to be equal to the int.

compute_labels : *bool, default=True*

Whether or not to compute labels for each fit.

copy : *bool, default=True*

Whether or not to make a copy of the given data. If set to False, the initial data will be overwritten.

For more solutions, explore the link below:

https://stackabuse.com/hierarchical-clustering-with-python-and-scikit-learn/

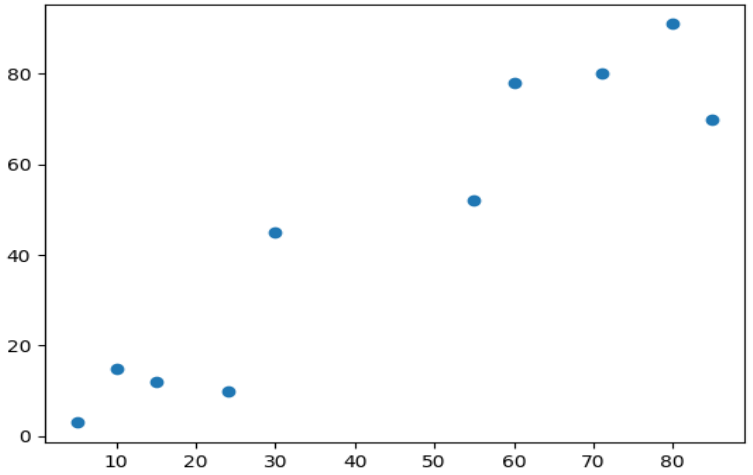
Density based Clustering

- DBSCAN
- OPTICS

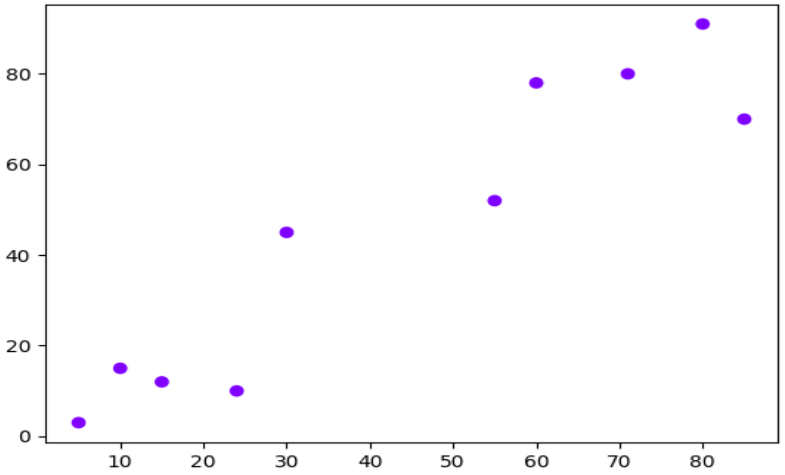
Clustering: DBSCAN

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import DBSCAN

X = np.array([[5,3],
      [10,15],
      [15,12],
      [24,10],
      [30,45],
      [85,70],
      [71,80],
      [60,78],
      [55,52],
      [80,91],])
plt.scatter(X[:,0],X[:,1], label='True Position')
plt.show()

dbscan = DBSCAN(eps=3, min_samples=2).fit(X)

print("Cluster Labels:")
print(dbscan.labels_)

plt.scatter(X[:,0],X[:,1], c=dbscan.labels_, cmap='rainbow')
plt.show()
```

Output:



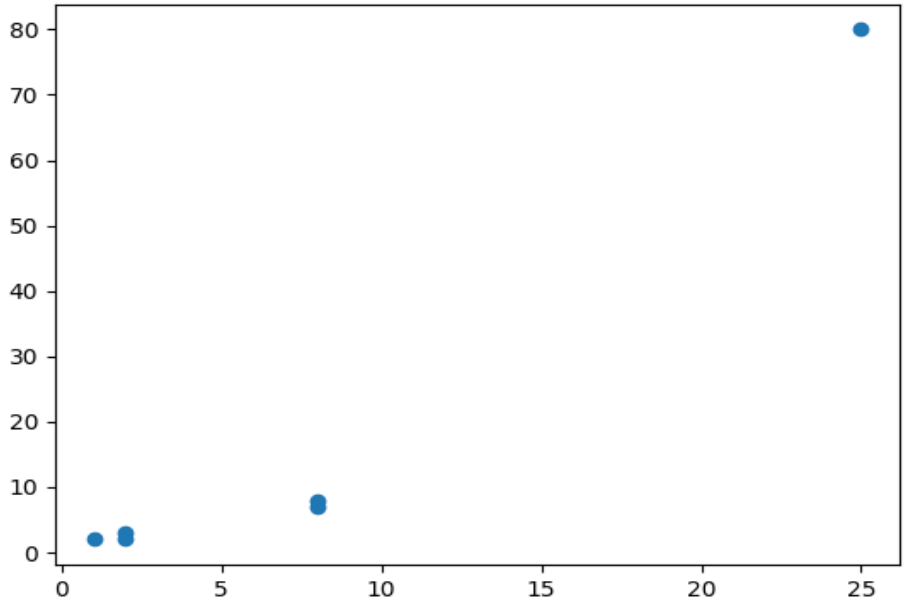

```
Cluster Labels:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```



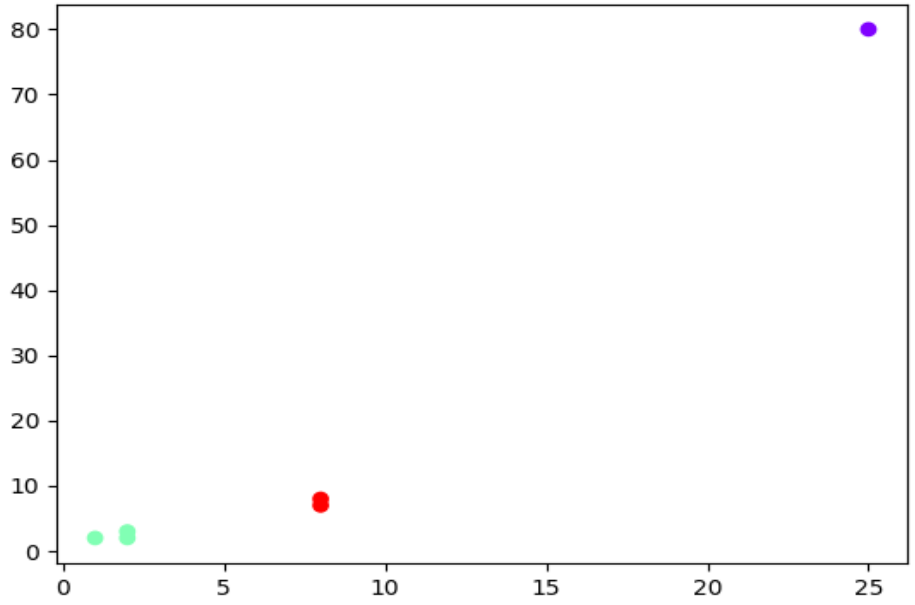
Clustering: DBSCAN

```
X = np.array([[1, 2],
              [2, 2],
              [2, 3],
              [8, 7],
              [8, 8],
              [25, 80]])
```

Output:




```
Cluster Labels:
[ 0  0  0  1  1 -1]
```



Clustering (Parameters): DBSCAN

DBSCAN - Density-Based Spatial Clustering of Applications with Noise. Finds core samples of high density and expands clusters from them. Good for data which contains clusters of similar density.

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*) [sourc

Clustering (Parameters): DBSCAN

eps : *float, default=0.5*

The maximum distance between two samples for one to be considered as in the neighborhood of the other. This is not a maximum bound on the distances of points within a cluster. This is the most important DBSCAN parameter to choose appropriately for your data set and distance function.

min_samples : *int, default=5*

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 : *str, or callable, default='euclidean'*

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 Glossary, in which case only "nonzero" elements may be considered neighbors for DBSCAN.

Clustering (Parameters): DBSCAN

metric_params : *dict, default=None*

Additional keyword arguments for the metric function.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

The algorithm to be used by the NearestNeighbors module to compute pointwise distances and find nearest neighbors. See NearestNeighbors module documentation for details.

leaf_size : *int, 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, default=None*

The power of the Minkowski metric to be used to calculate distance between points. If None, then `p=2` (equivalent to the Euclidean distance).

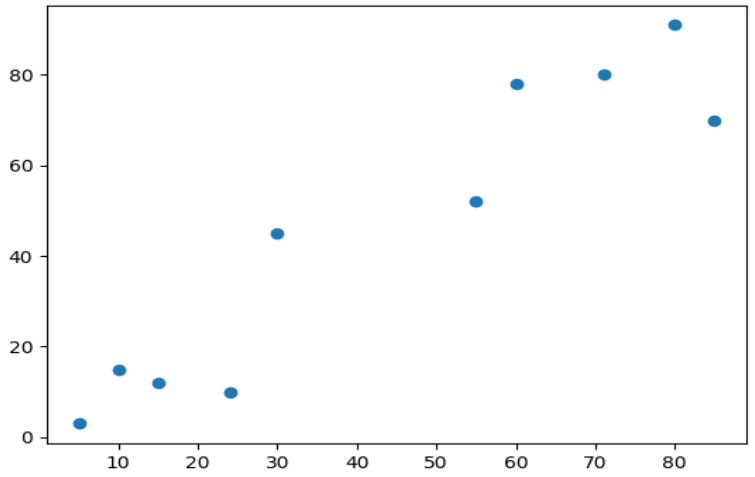
n_jobs : *int, default=None*

The number of parallel jobs to run. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

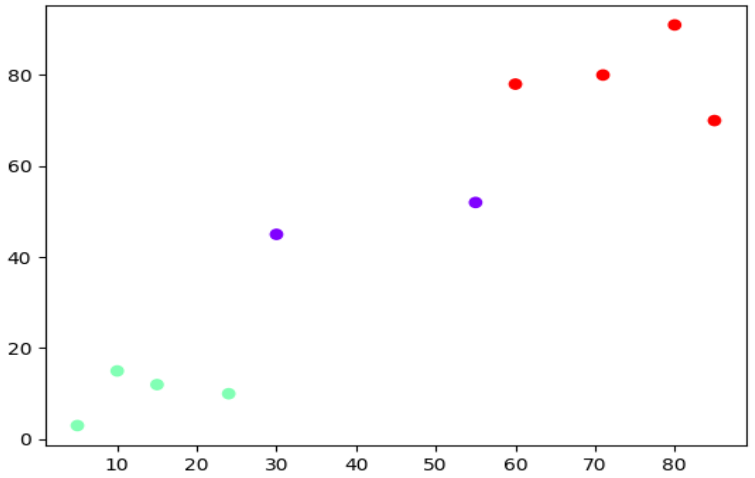
Clustering: OPTICS

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import OPTICS

X = np.array([[5,3],
      [10,15],
      [15,12],
      [24,10],
      [30,45],
      [85,70],
      [71,80],
      [60,78],
      [55,52],
      [80,91],])
plt.scatter(X[:,0],X[:,1], label='True Position')
plt.show()

optics = OPTICS(min_samples=2).fit(X)

print("Cluster Labels:")
print(optics.labels_)

plt.scatter(X[:,0],X[:,1], c=optics.labels_, cmap='rainbow')
plt.show()
```

Output:




```
Cluster Labels:
[ 0  0  0  0 -1  1  1  1 -1  1]
```



Clustering (Parameters): OPTICS

OPTICS (Ordering Points To Identify the Clustering Structure), closely related to DBSCAN, finds core sample of high density and expands clusters from them [1]. Unlike DBSCAN, keeps cluster hierarchy for a variable neighborhood radius. Better suited for usage on large datasets than the current sklearn implementation of DBSCAN.

Clusters are then extracted using a DBSCAN-like method (cluster_method = 'dbscan') or an automatic technique proposed in [1] (cluster_method = 'xi').

This implementation deviates from the original OPTICS by first performing k-nearest-neighborhood searches on all points to identify core sizes, then computing only the distances to unprocessed points when constructing the cluster order. Note that we do not employ a heap to manage the expansion candidates, so the time complexity will be O(n^2).

class sklearn.cluster.OPTICS(**, min_samples=5, max_eps=inf, metric='minkowski', p=2, metric_params=None, cluster_method='xi', eps=None, xi=0.05, predecessor_correction=True, min_cluster_size=None, algorithm='auto', leaf_size=30, memory=None, n_jobs=None*) [sourc

Clustering (Parameters): OPTICS

min_samples : *int > 1 or float between 0 and 1, default=5*

The number of samples in a neighborhood for a point to be considered as a core point. Also, up and down steep regions can't have more than `min_samples` consecutive non-steep points. Expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2).

max_eps : *float, default=np.inf*

The maximum distance between two samples for one to be considered as in the neighborhood of the other. Default value of `np.inf` will identify clusters across all scales; reducing `max_eps` will result in shorter run times.

metric : *str or callable, default='minkowski'*

Metric to use for distance computation. Any metric from scikit-learn or scipy.spatial.distance can be used.

If metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays as input and return one value indicating the distance between them. This works for Scipy's metrics, but is less efficient than passing the metric name as a string. If metric is "precomputed", X is assumed to be a distance matrix and must be square.

Valid values for metric are:

- from scikit-learn: ['cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan']
- from scipy.spatial.distance: ['braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule']

Clustering (Parameters): OPTICS

p : *int, default=2*

Parameter for the Minkowski metric from `pairwise_distances`. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric_params : *dict, default=None*

Additional keyword arguments for the metric function.

cluster_method : *str, default='xi'*

The extraction method used to extract clusters using the calculated reachability and ordering. Possible values are "xi" and "dbscan".

eps : *float, default=None*

The maximum distance between two samples for one to be considered as in the neighborhood of the other. By default it assumes the same value as `max_eps`. Used only when `cluster_method='dbscan'`.

xi : *float between 0 and 1, default=0.05*

Determines the minimum steepness on the reachability plot that constitutes a cluster boundary. For example, an upwards point in the reachability plot is defined by the ratio from one point to its successor being at most 1-xi. Used only when `cluster_method='xi'`.

predecessor_correction : *bool, default=True*

Correct clusters according to the predecessors calculated by OPTICS [2]. This parameter has minimal effect on most datasets. Used only when `cluster_method='xi'`.

Clustering (Parameters): OPTICS

min_cluster_size : *int > 1 or float between 0 and 1, default=None*

Minimum number of samples in an OPTICS cluster, expressed as an absolute number or a fraction of the number of samples (rounded to be at least 2). If `None`, the value of `min_samples` is used instead. Used only when `cluster_method='xi'`.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method. (default)

Note: fitting on sparse input will override the setting of this parameter, using brute force.

Clustering (Parameters): OPTICS

leaf_size : *int, default=30*

Leaf size passed to `BallTree` or `KDTree`. 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.

memory : *str or object with the joblib.Memory interface, default=None*

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.

n_jobs : *int, default=None*

The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

Clustering Performance Evaluation

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import metrics

labels_true = [0, 0, 0, 1, 1, 1]
labels_pred = [0, 0, 1, 1, 2, 2]

print("Clustering Performance Scores:")
print("-------------------------------------")
print("-------------------------------------")

score=metrics.rand_score(labels_true, labels_pred)
print("Rand Score: ", score)

print("-------------------------------------")

score=metrics.adjusted_rand_score(labels_true, labels_pred)
print("Adjusted Rand Score: ",score)

print("-------------------------------------")

score=metrics.mutual_info_score(labels_true, labels_pred)
print("Mutual Info Score: ",score)

print("-------------------------------------")

score=metrics.adjusted_mutual_info_score(labels_true, labels_pred)
print("Adjusted Mutual Info Score: ",score)

print("-------------------------------------")

score=metrics.normalized_mutual_info_score(labels_true, labels_pred)
print("Normalized Mutual Info Score: ",score)

print("-------------------------------------")
```

Output:

```
Clustering Performance Scores:
-------------------------------------
-------------------------------------
Rand Score:  0.6666666666666666
-------------------------------------
Adjusted Rand Score:  0.24242424242424243
-------------------------------------
Mutual Info Score:  0.4620981203732969
-------------------------------------
Adjusted Mutual Info Score:  0.2987924581708901
-------------------------------------
Normalized Mutual Info Score:  0.5158037429793889
-------------------------------------
```

Clustering Performance Evaluation

```
score=metrics.homogeneity_score(labels_true, labels_pred)
print("Homogeneity Score: ",score)

print("-------------------------------------")

score=metrics.completeness_score(labels_true, labels_pred)
print("Completeness Score: ",score)

print("-------------------------------------")

score=metrics.v_measure_score(labels_true, labels_pred)
print("V-measure Score: ",score)

print("-------------------------------------")
```

Output:

```
Homogeneity Score:  0.6666666666666669
-------------------------------------
Completeness Score:  0.42061983571430 5
-------------------------------------
V-measure Score:  0.5158037429793889
-------------------------------------
```

To know more about clustering implementations & evaluations, explore the link below:

https://scikit-learn.org/stable/modules/clustering.html#