```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806
```

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Returns **self** :

transform(*X*)
Transform a new matrix using the built clustering

Parameters **X** : array-like, shape = [n_samples, n_features] or [n_features]

A M by N array of M observations in N dimensions or a length M array of M one-dimensional observations.

Returns **Y** : array, shape = [n_samples, n_clusters] or [n_clusters]

The pooled values for each feature cluster.

Examples using sklearn.cluster.FeatureAgglomeration

- *Feature agglomeration*
- *Feature agglomeration vs. univariate selection*

sklearn.cluster.KMeans

class sklearn.cluster.**KMeans**(*n_clusters=8*, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *verbose=0*, *random_state=None*, *copy_x=True*, *n_jobs=1*, *algorithm='auto'*)
K-Means clustering

Read more in the *User Guide*.

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.

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

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

verbose : int, default 0

Verbosity mode.

random_state : int, RandomState instance or None, optional, default: None

If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

copy_x : boolean, default True

When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True, 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.

n_jobs : int

The number of jobs to use for the computation. This works by computing each of the n_init runs in parallel.

If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

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

labels_ : :

Labels of each point

inertia_ : float

Sum of squared distances of samples to their closest cluster center.

See also:

MiniBatchKMeans Alternative online implementation that does incremental updates of the centers positions using mini-batches. For large scale learning (say n_samples > 10k) MiniBatchKMeans is probably much faster than the default batch implementation.

Notes

The k-means problem is solved using Lloyd's algorithm.

The average complexity is given by O(k n T), were n is the number of samples and T is the number of iteration.

The worst case complexity is given by O(n^(k+2/p)) with n = n_samples, p = n_features. (D. Arthur and S. Vassilvitskii, 'How slow is the k-means method?' SoCG2006)

In practice, the k-means algorithm is very fast (one of the fastest clustering algorithms available), but it falls in local minima. That's why it can be useful to restart it several times.

Examples

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

Methods

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

__init__ (*n_clusters=8*, *init='k-means++'*, *n_init=10*, *max_iter=300*, *tol=0.0001*, *precompute_distances='auto'*, *verbose=0*, *random_state=None*, *copy_x=True*, *n_jobs=1*, *algorithm='auto'*)

fit (*X*, *y=None*)

Compute k-means clustering.

Parameters **X** : array-like or sparse matrix, shape=(n_samples, n_features)

Training instances to cluster.

y : Ignored

fit_predict (*X*, *y=None*)

Compute cluster centers and predict cluster index for each sample.