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*) [source]

»

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.

max_iter : int, default: 300

Maximum number of iterations of the k-means algorithm for a single run.

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.

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.

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

tol : float, default: 1e-4

Relative tolerance with regards to inertia to declare convergence

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.

random_state : integer or numpy.RandomState, optional

The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

verbose : int, default 0

Verbosity mode.

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.

Attributes:

cluster_centers_ : array, [n_clusters, n_features]

Coordinates of cluster centers

labels_ : :

Labels of each point

inertia_ : float

Sum of 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 to 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.

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[, y])	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*) [source]

fit(*X*, *y=None*) [source]

Compute k-means clustering.

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

fit_predict(*X*, *y=None*) [source]

Compute cluster centers and predict cluster index for each sample.

Convenience method; equivalent to calling fit(X) followed by predict(X).

`fit_transform`(*X*, *y=None*) [source]

Compute clustering and transform X to cluster-distance space.

Equivalent to fit(X).transform(X), but more efficiently implemented.

`get_params`(*deep=True*) [source]

Get parameters for this estimator.

Parameters: **deep: boolean, optional** :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns: **params** : mapping of string to any

Parameter names mapped to their values.

`predict`(*X*) [source]

Predict the closest cluster each sample in X belongs to.

In the vector quantization literature, *cluster_centers_* is called the code book and each value returned by *predict* is the index of the closest code in the code book.

Parameters: **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data to predict.

Returns: **labels** : array, shape [n_samples,]

Index of the cluster each sample belongs to.

`score`(*X*, *y=None*) [source]

Opposite of the value of X on the K-means objective.

Parameters: **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data.

Returns: **score** : float

Opposite of the value of X on the K-means objective.

`set_params`(***params*) [source]

Set the parameters of this estimator.

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

Returns: **self** :

`transform`(*X*, *y=None*) [source]

Transform X to a cluster-distance space.

In the new space, each dimension is the distance to the cluster centers. Note that even if X is sparse, the array returned by *transform* will typically be dense.

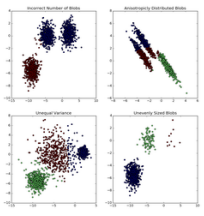
Parameters: **X** : {array-like, sparse matrix}, shape = [n_samples, n_features]

New data to transform.

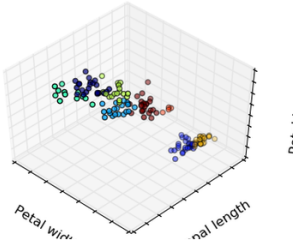
Returns: **X_new** : array, shape [n_samples, k]

X transformed in the new space.

Examples using `sklearn.cluster.KMeans`

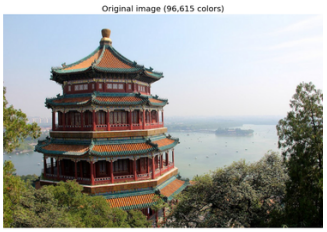


Demonstration of k-means assumptions

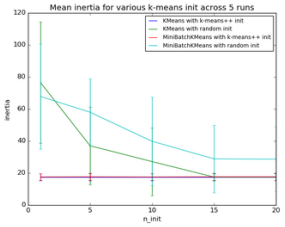


K-means Clustering

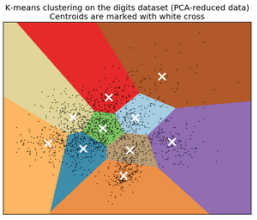
Vector Quantization Example



Color Quantization using K-Means

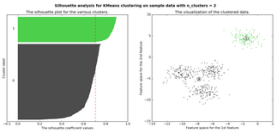


Empirical evaluation of the impact of k-means initialization



A demo of K-Means clustering on the handwritten digits data

Comparison of the K-Means and MiniBatchKMeans clustering algorithms



Selecting the number of clusters with silhouette analysis on KMeans clustering

Clustering text documents using k-means