Assignment 4

CSCD01 Winter 2021

Spaghetti Code

Julian Barker, Adam Ah-Chong, Andrew Gao, Jesse Francispillai, Laphonso (Poncie) Reyes, Saad Ali

Table of Contents

Introduction **2**
Feature #14214 **2**
Description 2
Design Details 2
Initial State 2
Goal State 3
Feature (Enhancement) #15336 **5**
Description 5
Design Details 5
Initial State 5
Goal State 5
Notes: 6
Example of Use: 6
Implementation of Feature #14214 **7**
Acceptance Testing 7
User Guide 9
Team Organization **15**
Works Cited **16**

Introduction

To begin the assignment, each team member chose one hard issue to recommend to the team. Everyone on the team would then read through each issue then vote for the 2 issues that they believed should be worked on. The 2 issues with the most votes were chosen. After the initial design details were done, a team meeting was held to decide which feature to implement. Each team member assigned themselves tasks on JIRA, informing the rest of the group what they were working on and any issues or roadblocks they encountered.

Issues selected:
Team decision: https://github.com/scikit-learn/scikit-learn/issues/14214
Team decision: https://github.com/scikit-learn/scikit-learn/issues/15336

Feature #14214

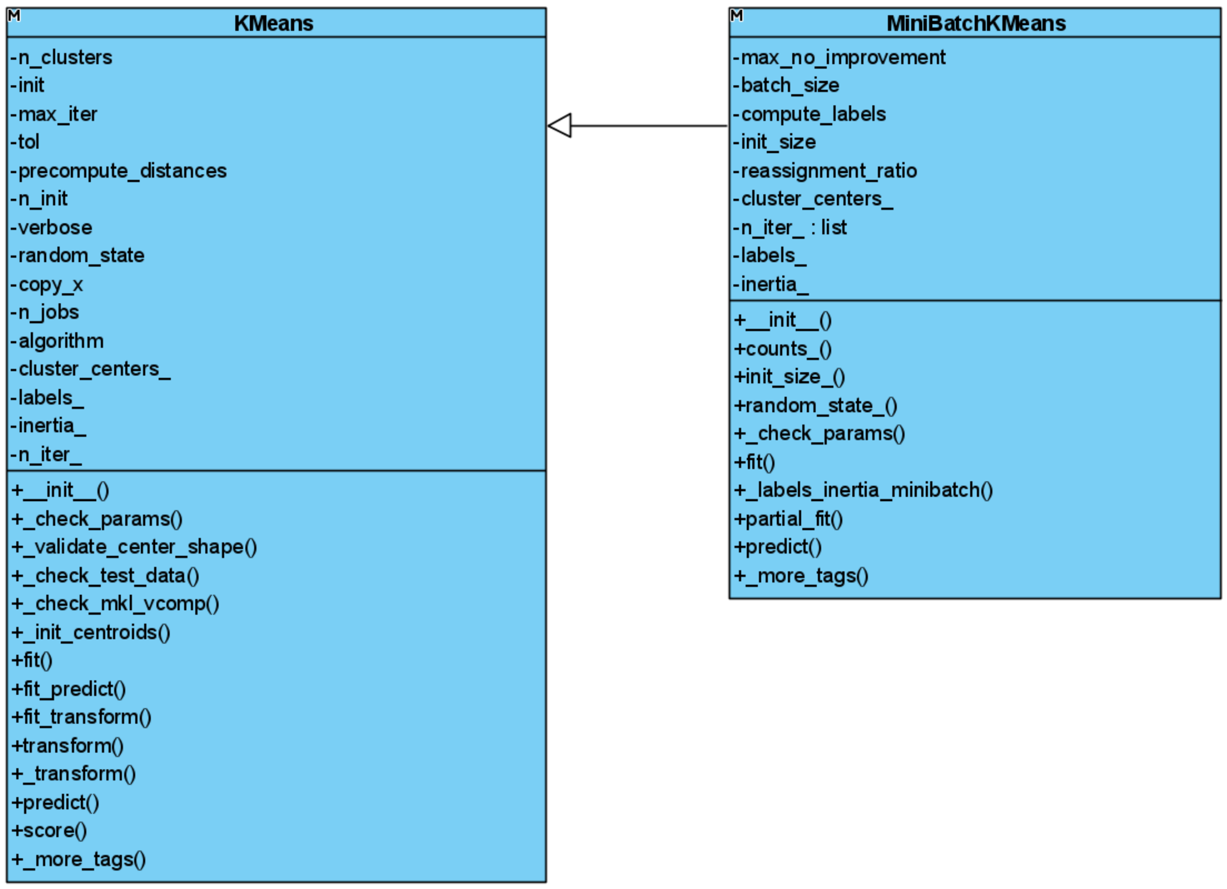
Description

This issue proposes to create a new K-Means clustering class in Scikit-learn for the bisecting K-Means algorithm. The bisecting K-Means algorithm combines the idea of K-Means clustering with hierarchy clustering. The github issue makes reference to the conducted research "A Comparison of Document Clustering Techniques" by Steinbach et al., illustrating how the bisecting K-Means technique exceeds the standard K-Means by measuring a variety of cluster evaluation metrics. Hence, there has been a popularity in demand with regard to implementing this estimator into Scikit-learn.
Sources: http://www.philippe-fournier-viger.com/spmf/bisectingkmeans.pdf

Design Details

Initial State

The current state of the relevant architecture is the _KMeans package that includes currently implemented K-Means variants, like Mini-Batch K-Means for example.



Goal State

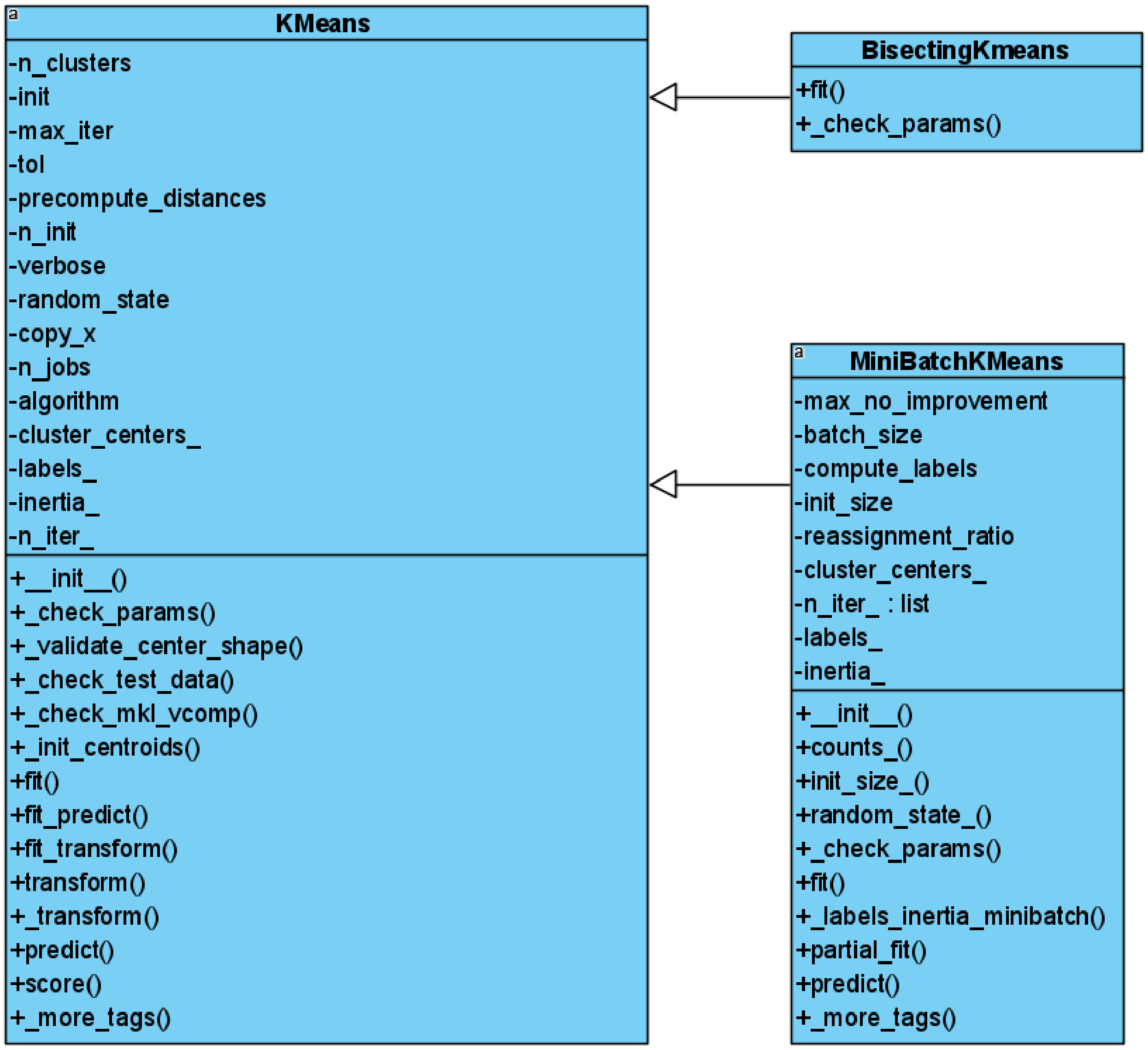
The Bisecting K-means feature will require a new class `BisectingKMeans` in `cluster/_kmeans.py`, which will extend the functionality of the abstracted `KMeans` class which specializes as a K-Means clustering method. This new `BisectingKMeans` class will require some additional implementations. As stated by the research study A Comparison of Document Clustering Techniques (Steinbach et al., 2016), the following is the general algorithm for computing Bisecting k-Means for finding K clusters:

1. Pick a cluster to split.
2. Find 2 sub-clusters using the basic K-means algorithm. (Bisecting step)
3. Repeat step 2, the bisecting step, for ITER times and take the split that produces the clustering with the highest overall similarity.
4. Repeat steps 1, 2 and 3 until the desired number of clusters is reached

In order to accomplish this in Scikit-learn, we will have to integrate it with respect to the existing architecture of the `KMeans` class.

In class `BisectingKMeans`, the entirety of this algorithm would be implemented in the overridden methods `def fit()` and `def predict()`.

- `def fit()`: Implements the above algorithm with use of the helper function `def k_means` to iteratively find the 2 sub-clusters with the most similarity. We repeat this iterative process until we reach the desired number of clusters, as stated in the above algorithm.
- `def predict()`: Predicts the closest cluster for each cluster sample to determine their general groupage using the already defined function in this module def `_labels_intertia()` that specializes in generalized KMeans labels and inertia computations.
- Utilize the helper functions already defined
    - `def _labels_intertia()` to compute the labels and the inertia of clusters.
    - `def k_means()` to facilitate dividing samples in groups of equal variance, minimizing the inertia or within-cluster sum-of-squares (source: https://scikit-learn.org/stable/modules/clustering.html#k-means).



Feature (Enhancement) #15336

Description

Add sparse matrix support for the HistGradientBoostingClassifier.

Sparse matrices using the *scipy* library do not store 0-value-elements in memory, allowing for large arrays with many 0 values to be stored in potentially much less memory space than normal dense arrays.

Design Details

Initial State

The HistGradientBoostingClassifier goes through the following steps in order to run its fitness model:

1. Validate parameters
2. Bin training data
3. Create a number of tree growers for estimation
4. Estimate using binned data and tree classifiers

Currently, the HistGradientBoostingClassifier does not support sparse matrices for any of these steps. The class simply raises an error should the training data inputted be a sparse matrix.

Validation processing uses the base validation processes of scikit-learn.

The binning, tree classification, and estimation processes all use code and classes made specifically for Histogram Gradient estimators. All of the classes that are involved with binning, trees growing, and estimation are contained within the *ensemble* package, made specifically with the Gradient estimators in mind.

Goal State

Firstly, we wish to modify the validation parameters to include sparse matrices, likely of the *csr_matrix* type, as that is the most common, simple, and widely used matrix type within scikit-learn.

Next, the binning, trees growing, and estimation processes need to have sparse matrix support added to them.

The binning process uses a class called *_BinMapper*, which takes in a matrix and transforms it into a new matrix where each value at any index is instead an integer representing which bin the value at the same index in the original matrix is in. In order to make this work with our feature, we will not only have to make sure that the *_BinMapper* can take in sparse matrices as an input, but that it can also output sparse matrices, as outputting a dense matrix would defeat the purpose of the sparse matrix functionality, since the input matrix and output matrix are of the same sizes and dimensions.

Note that the KBinsDiscretizer class may be useful for sparse matrix binning operations. It does not support sparse matrix, but it does output sparse matrices as a result of its binning process.

Tree growing within the estimator is done with a class called *TreeGrower*. The *HistGradientBoostingClassifier* uses several *TreeGrower* instances and consolidates their results to create a final estimation for its fitness function. For this feature, we will need to add sparse matrix functionality to the *TreeGrower* class, allowing it to input sparse matrices and use them for creating its estimations.

The final step of estimation is essentially taking the results of the *TreeGrower* classes and consolidating them appropriately according to the initial parameters set. Should the rest of the sparse matrix compatibility work well, this step will simply require us to make sure that the correct kind of data is outputted once everything else is complete.

Notes:

- Modify validation at line 202 to allow sparse matrices
    - Add parameter *accept_sparse="(sparse matrix type)"*
        - Type is generally *csr_matrix*
    - EASY
- Find a way to bin sparse matrix data *(_bin_data() at line 292)*
    - Binning array X creates an array X_binned of elements 0-255, where X_binned[i]==bin number of X[i]
    - HARD - BLOCKER
- Add sparse matrix compatibility to TreeGrower *(line 447)*
    - MEDIUM
- Final checks with sparse matrices
    - EASY

Example of Use:

```
import scipy
from scipy import sparse
from scipy.sparse import rand
from scipy.sparse import csr_matrix
import numpy as np
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingClassifier

X_data = scipy.sparse.rand(10,10,0.5)     # sparse training data
y_data = np.array([0]*9 + [1])      # target data doesn't have to be sparse
```

```
clf = HistGradientBoostingClassifier()

# dense arrays give results
clf.fit(X_data.toarray(), y_data)
clf.score(X_data.toarray(), y_data)

# sparse arrays raise errors
clf.fit(X_data, y_data)
clf.score(X_data, y_data)
```

Implementation of Feature #14214

Acceptance Testing

1. Let us import numpy and the newly implemented BisectingKmeans to set up the environment first

```
import numpy as np

from sklearn.cluster import BisectingKMeans

from sklearn.datasets import make_blobs
```

2. Now let us set up some variable to start using the BisectingKmeans algorthim we will create a new dataset with some sample weight and centers as following:

```
X = np.array([[0, 0], [0.5, 0], [0.5, 1], [1, 1]], dtype=np.float32)
```

Here we created a new arrays need to the Bisecting Kmeans and set the dtype which is the format of the output.

3. Now lets call BisectingKmeans and proceed to run fit() with the algorithm “full”

```
bkmeans = BisectingKMeans(n_clusters=2, n_init=1, algorithm="full")

bkmeans.fit(X)
```

4. Now we can take a look at the data stored within bkmeans, we can view the labels the inertia and the cluster_centers which is the result of our BisectingKmeans algorithm. The results should look as following. The array may be reverse depending on how the BisectingKmeans algorithm picked its clustering for the labels.

bkmeans.labels_ = [1, 1, 0, 0] or [0, 0, 1, 1]

bkmeans.inertia_ = 0.25

bkmeans.cluster_centers_ = [0.75, 1],[0.25, 0] or [0.25,0],[0.75, 1]

5. BisectingKmeans has an alternate algorithm that can be used to calculate the clustering, which is the “elkan” algorithm, we can rerun steps 3 and 4 this time specifying the algorithm to “elkan”. The results should be the same as using the “full” algorithm.

```
bkmeans = BisectingKMeans(n_clusters=2, n_init=1, algorithm="elkan")

bkmeans.fit(X)
```

6. Let us change the parameters of the BisectingKmeans, let us change the number of clusters to 3 and call fit.

```
bkmeans = BisectingKMeans(n_clusters=3, n_init=1, algorithm="elkan")

bkmeans.fit(X)
```

We should get:

bkmeans.labels_ = [0, 0, 2, 1] or [1, 2, 0, 0]

bkmeans.inertia_ = 0.125

bkmeans.cluster_centers_ = [[0.25, 0.],[1. , 1.],[0.5 , 1.]]

or

[[0.5 , 1.], [1. , 1.], [0.25, 0.]]

7. Let us try setting the clusters to 5.

```
bkmeans = BisectingKMeans(n_clusters=5, n_init=1, algorithm="elkan")

bkmeans.fit(X)
```

This time the function should error as following, this as intended as the initial array has a sample size of 4, thus it is not possible to create 5 clusters out of 4 points.

ValueError: n_samples=4 should be >= n_clusters=5.

8. We can also use the BisectingKmeans to predict a model as well to do this let us create some sample data: Here we create a sample of size 10 with 2 centers and 2 features

```
X = make_blobs(n_samples=10, n_features=2, centers=2,

random_state=random)[0].astype(np.float32, copy=False)

X = np.asarray(X)
```

9. Now we set up the BisectingKmeans algorthim and using the np library we will choose random states

```
random = np.random.RandomState(1)

bkmeans = BisectingKMeans(algorithm="full", n_clusters=10,

random_state=random,tol=1e-7,max_iter=2)
```

10. Now we can predict the model by calling fit_predict or we can call fit than predict. Both the following work

```
bkmeans.fit_predict(X)
bkmeans.fit(X).predict(X)
```

11. The result should be similar to what is below, the values will not be the same since we are choosing the states randomly.

```
>>> bkmeans.fit_predict(X)

array([8, 6, 3, 9, 2, 4, 7, 1, 0, 5], dtype=int32)
```

User Guide

sklearn.cluster.BisectingKMeans

(This section is new and everything in it is a new addition to the user guide)

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

Bisecting K-Means clustering.

Read more in the User Guide.

Parameters:

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, 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 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 times 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 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.

Deprecated since version 0.23: ‘precompute_distances’ was deprecated in version 0.22 and will be removed in 1.0 (renaming of 0.25). It has no effect.
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.

Deprecated since version 0.23: n_jobs was deprecated in version 0.23 and will be removed in 1.0 (renaming of 0.25).
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 compatibility) chooses “elkan” but it might change in the future for a better heuristic.

Changed in version 0.18: Added Elkan algorithm

Attributes:
cluster_centers_ : *ndarray of shape (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_ : *ndarray of shape (n_samples,)*
Labels of each point
inertia_ : *float*
Sum of squared distances of samples to their closest cluster center.
n_iter_ : *int*
Number of iterations run.

Methods

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

Compute bisecting k-means clustering.

Parameters:
X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
Training instances to cluster. It must be noted that the data will be converted to C ordering, which will cause a memory copy if the given data is not C-contiguous. If a sparse matrix is passed, a copy will be made if it’s not in CSR format.
y : *Ignored*

Not used, present here for API consistency by convention.
sample_weight : *array-like of shape (n_samples,), default=None*
The weights for each observation in X. If None, all observations are assigned equal weight.
New in version 0.20.

Returns:
self
Fitted estimator.

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

Compute cluster centers and predict cluster index for each sample.

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

Parameters:
X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
New data to transform.
y : *Ignored*
Not used, present here for API consistency by convention.
sample_weight : *array-like of shape (n_samples,), default=None*
The weights for each observation in X. If None, all observations are assigned equal weight.

Returns:
labels : *ndarray of shape (n_samples,)*
Index of the cluster each sample belongs to.

`fit_transform`(*X*, *y=None*, *sample_weight=None*)

Compute clustering and transform X to cluster-distance space.

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

Parameters:
X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
New data to transform.
y : *Ignored*
Not used, present here for API consistency by convention.
sample_weight : *array-like of shape (n_samples,), default=None*
The weights for each observation in X. If None, all observations are assigned equal weight.

Returns:
X_new : *ndarray of shape (n_samples, n_clusters)*
X transformed in the new space.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters:
deep : *bool, default=True*
If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns:
params : *dict*
Parameter names mapped to their values.

`predict`(*X*, *sample_weight=None*)

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} of shape (n_samples, n_features)*
New data to predict.
sample_weight : *array-like of shape (n_samples,), default=None*
The weights for each observation in X. If None, all observations are assigned equal weight.

Returns:
labels : *ndarray of shape (n_samples,)*
Index of the cluster each sample belongs to.

`score`(*X*, *y=None*, *sample_weight=None*)

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

Parameters:
X : *{array-like, sparse matrix} of shape (n_samples, n_features)*
New data.
y : *Ignored*
Not used, present here for API consistency by convention.
sample_weight : *array-like of shape (n_samples,), default=None*

The weights for each observation in X. If None, all observations are assigned equal weight.

Returns:
score : *float*
Opposite of the value of X on the K-means objective.

`set_params`(***params*)

Set the parameters of this estimator.

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

Parameters:
****params : *dict***
Estimator parameters.

Returns:
self : *estimator instance*
Estimator instance.

`transform`(*X*)

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} of shape (n_samples, n_features)*
New data to transform.

Returns:
X_new : *ndarray of shape (n_samples, n_clusters)*
X transformed in the new space.

Team Organization

Meeting 1: March 21, 2021

1. Examine the list of requested features/serious bug fixes in scikit-learn issue list in GitHub. From the issue list, select at least two features/bug fixes to examine further. These should not be small fixes: we are looking for substantial new development here. The best way to ensure that you are working on acceptable features/fixes is to contact your TA very early for a confirmation. You will need to select issues which are at hard difficulty level.
   Team decision:
   https://github.com/scikit-learn/scikit-learn/issues/14214
   https://github.com/scikit-learn/scikit-learn/issues/15336

Meeting 2: March 27, 2021

2. Briefly and clearly describe the selected features/bug fixes in your report.
3. For each of the selected features/bug fixes, investigate what parts of the existing code base you would need to modify in order to implement the feature. Produce designs for the selected feature, describing your plans for the organization of new code, as well as all interactions between new code and existing code. You guessed it: UML diagrams would be very helpful here.

Meeting 3: March 30, 2021

4. Select one of these features/bug fixes for implementation, and briefly explain your decision.
5. For the selected feature/bug fix, produce a suite of acceptance tests that will demonstrate that the feature/bug fix has been implemented correctly. Design these as ”customer acceptance” tests: i.e. a description of the steps a user needs to carry out to check that the program works as expected.
6. For the selected feature/bug fix produce a unit test suite that will demonstrate that the feature/bug fix has been implemented correctly.

Works Cited

Steinbach, M., Karypis, G., & Kumar, V. (2016, July 27). A Comparison of Document Clustering Techniques. Retrieved from http://www.stat.cmu.edu/~rnugent/PCMI2016/papers/DocClusterComparison.pdf