```
"""
DBSCAN: Density-Based Spatial Clustering of Applications with Noise
"""

# Author: Robert Layton <robertlayton@gmail.com>
#         Joel Nothman <joel.nothman@gmail.com>
#         Lars Buitinck
#
# License: BSD 3 clause

import warnings

import numpy as np

from ..base import BaseEstimator, ClusterMixin
from ..metrics import pairwise_distances
from ..utils import check_array, check_consistent_length
from ..neighbors import NearestNeighbors

from ._dbscan_inner import dbscan_inner


#This is the definition of the internal dbscan function. This is not interfaced for the front-end user.

def dbscan(X, eps=0.5, min_samples=5, metric='minkowski',
           algorithm='auto', leaf_size=30, p=2, sample_weight=None,
           random_state=None):

    """Perform DBSCAN clustering from vector array or distance matrix.

    Parameters
    ----------
    X : array or sparse (CSR) matrix of shape (n_samples, n_features), or \
            array of shape (n_samples, n_samples)
        A feature array, or array of distances between samples if
        ``metric='precomputed'``.

    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 metrics.pairwise.pairwise_distances for its
metric parameter.
If metric is "precomputed", X is assumed to be a distance matrix and
must be square.

#This is the choice of distance metric to be used in the calculation of the distance tensor, all done by the NearestNeighbors module.

algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, optional
The algorithm to be used by the NearestNeighbors module
to compute pointwise distances and find nearest neighbors.
See NearestNeighbors module documentation for details.

#This variable allows us to pass a request in the form of a string into the NN module.

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.

#This is a variable that is part of the data structure this algorithm uses. (not required for grading)

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

A Minkowski Metric in this case refers to the NearestNeighbors calculation. Minkowski metric is the tensor describing the distance relationships among all entities in the clustering calculation. The power is used in the distance calculation. Where p=1 you have manhattan distance. If p=2, you have euclidean distance.

sample_weight : array, shape (n_samples,), optional
Weight of each sample, such that a sample with a weight of at least
``min_samples`` is by itself a core sample; a sample with negative

weight may inhibit its eps-neighbor from being core.
Note that weights are absolute, and default to 1.

#This allows you to take an array of weights allowing certain data points to be labeled (perhaps as noise, perhaps as belonging to another dataset, perhaps as an output of another algorithm….)

random_state: numpy.RandomState, optional
Deprecated and ignored as of version 0.16, will be removed in version
0.18. DBSCAN does not use random initialization.

#I'm not sure why they feel it necessary to have an intake parameter for a random state. It looks like this may be experimental code that has been since forgotten.

Returns
----
core_samples : array [n_core_samples]
Indices of core samples.

#core_samples are the data points that are density connected.

labels : array [n_samples]
Cluster labels for each point. Noisy samples are given the label -1.

#labels are the N data point labels for core, edge, or noise.

Notes

See examples/cluster/plot_dbscan.py for an example.

This implementation bulk-computes all neighborhood queries, which increases
the memory complexity to O(n.d) where d is the average number of neighbors,
while original DBSCAN had memory complexity O(n).

#The bulk computation results from the Nearest Neighbors implementation.

References

Ester, M., H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based
Algorithm for Discovering Clusters in Large Spatial Databases with Noise".

```
    In: Proceedings of the 2nd International Conference on Knowledge Discovery
    and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996
    """
    if not eps > 0.0:
        raise ValueError("eps must be positive.")
```

#you can’t have a negative epsilon value.

```
    if random_state is not None:
        warnings.warn("The parameter random_state is deprecated in 0.16 "
                      "and will be removed in version 0.18. "
                      "DBSCAN is deterministic except for rare border cases.",
                      category=DeprecationWarning)



    X = check_array(X, accept_sparse='csr')
```

#check_array: Input validation on an array, list, sparse matrix or similar, by default the input is converted to a 2-d numpy array. This also densifies sparse matrices (at significant memory cost)

```
    if sample_weight is not None:
        sample_weight = np.asarray(sample_weight)
        check_consistent_length(X, sample_weight)
```

#check_consistent_length: ensures that the dimensions of X and the sample_weight are correct.

```
    # Calculate neighborhood for all samples. This leaves the original point
    # in, which needs to be considered later (i.e. point i is in the
    # neighborhood of point i. While True, its useless information)

    if metric == 'precomputed':
      # If we are using a precomputed Minkowski metric as input, follow this condition.

        D = pairwise_distances(X, metric=metric)

            # Returns the minkowski metric tensor D (all distances by all distances)
```

```
        neighborhoods = np.array([np.where(x <= eps)[0] for x in D])
```
#eps is the maximum distance between two points required to form a neighborhood, so the algorithm is selecting out those data points that are definitely in the same neighborhood. This will create lists of points all with D_ij < epsilon. D is organized by row, such that the elements of the row i are the actual distance between that point i and all other points j != i. The where command here will return a list of N lists, each list containing the indices of all other points within the neighborhood of point i.

```
    else:
        neighbors_model = NearestNeighbors(radius=eps, algorithm=algorithm,
                                           leaf_size=leaf_size,
                                           metric=metric, p=p)



        neighbors_model.fit(X)
        # This has worst case O(n^2) memory complexity
```

#This calls the NearestNeighbors module. The same module is also used for K-NN of course. (I would not expect you to annotate the whole module here, but if you did, you'd go find the .pyx files and annotate those. This would add points to your grade.). NN calculates the nearest neighbors of every point and stores everything in an efficient data structure. This adds additional cost to the calculation, of course.

```
        neighborhoods = neighbors_model.radius_neighbors(X, eps,
                                                         return_distance=False)
```

#The neighborhoods NN returns is exactly the same as above.

```
    if sample_weight is None:
        n_neighbors = np.array([len(neighbors) for neighbors in neighborhoods])
    else:
        n_neighbors = np.array([np.sum(sample_weight[neighbors])
                                for neighbors in neighborhoods])
```

#This calculates the number of neighbors for each of the N data points.

```
    # Initially, all samples are noise.
```

```
    labels = -np.ones(X.shape[0], dtype=np.intp)

#Initialize the N labels for each of the data points to be -1 (the flag for noise).

    # A list of all core samples found.
    core_samples = np.asarray(n_neighbors >= min_samples, dtype=np.uint8)

#This returns a mask describing those points that have a minimum of min_samples (minPoints) neighbor points.

    dbscan_inner(core_samples, neighborhoods, labels)

#This is odd. dbscan_inner is an external function. I guess we'll have to look for it in the scikits code and annotate it. Since it is directly related to the algorithm itself, it's necessary to annotate it.

    return np.where(core_samples)[0], labels

# This returns a list of core samples, and labels for the clustering. The labels are enumerated.

_____
#Put some kind of dividing line between different functions.

# Fast inner loop for DBSCAN.
# Author: Lars Buitinck
# License: 3-clause BSD

cimport cython
from libcpp.vector cimport vector
cimport numpy as np
import numpy as np


# Work around Cython bug: C++ exceptions are not caught unless thrown within
# a cdef function with an "except +" declaration.
cdef inline void push(vector[np.npy_intp] &stack, np.npy_intp i) except +:
    stack.push_back(i)
#refer to the above comment

@cython.boundscheck(False)
@cython.wraparound(False)
```

```
#definiton of the dbscan_inner function
def dbscan_inner(np.ndarray[np.uint8_t, ndim=1, mode='c'] is_core,
                 np.ndarray[object, ndim=1] neighborhoods,
                 np.ndarray[np.npy_intp, ndim=1, mode='c'] labels):
    cdef np.npy_intp i, label_num = 0, v #these are cdef of c-type integers

    cdef np.ndarray[np.npy_intp, ndim=1] neighb #this is a cdef of an array of integers. (neighbor labels)
    cdef vector[np.npy_intp] stack #this is a cdef of a vector (stack for handling the order of points) allows us to use vector operations like pop() and push()


    for i in range(labels.shape[0]):
        if labels[i] != -1 or not is_core[i]: # if the label isn't noise or isn't part of a core cluster section, move on to the next point in the label stack..
            continue

        # Depth-first search starting from i, ending at the non-core points.
        # This is very similar to the classic algorithm for computing connected
        # components, the difference being that we label non-core points as
        # part of a cluster (component), but don't expand their neighborhoods.

        while True: #run the loop to completion; will run indefinitely until there's nothing on the stack
            #i and label_num were initialized to 0 above
            if labels[i] == -1: #if the point is noise
                labels[i] = label_num #Set the point to the current label_num (cluster number)
                if is_core[i]:
                    neighb = neighborhoods[i] #Add the point's neighborhood to the neighb list
                    for i in range(neighb.shape[0]): #for all the points in the current point's neighborhood
                        v = neighb[i] #take all the neighbors of the current neighbor point
                        if labels[v] == -1: #if these neighbor-neighbors are currently labeled as noise put them onto the stack!
                            push(stack, v)

            if stack.size() == 0: #if we reach the bottom of the stack, exit!
                break
            i = stack.back() # this provides a reference to the next point off the bottom of the stack! http://www.cplusplus.com/reference/vector/vector/back/
            stack.pop_back() #this removes the item from the stack after we've referenced it.
```

label_num += 1 #if we have exited the neighborhood loop, increment the cluster index. (Start a new cluster)

---

class DBSCAN(BaseEstimator, ClusterMixin):

#This is the wrapper class that provides interface to the user.

"""Perform DBSCAN clustering from vector array or distance matrix.
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.

Parameters

#just intake parameters

---
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 metrics.pairwise.calculate_distance for its
metric parameter.
If metric is "precomputed", X is assumed to be a distance matrix and
must be square.
algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, optional
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, 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.
random_state: numpy.RandomState, optional
    Deprecated and ignored as of version 0.16, will be removed in version
    0.18. DBSCAN does not use random initialization.

Attributes
----------
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.

Notes
-----
See examples/cluster/plot_dbscan.py for an example.

This implementation bulk-computes all neighborhood queries, which increases
the memory complexity to O(n.d) where d is the average number of neighbors,
while original DBSCAN had memory complexity O(n).

References
----------
Ester, M., H. P. Kriegel, J. Sander, and X. Xu, "A Density-Based
Algorithm for Discovering Clusters in Large Spatial Databases with Noise".
In: Proceedings of the 2nd International Conference on Knowledge Discovery
and Data Mining, Portland, OR, AAAI Press, pp. 226-231. 1996
"""

def __init__(self, eps=0.5, min_samples=5, metric='euclidean',
             algorithm='auto', leaf_size=30, p=None, random_state=None):
    self.eps = eps
    self.min_samples = min_samples
    self.metric = metric
    self.algorithm = algorithm
    self.leaf_size = leaf_size
```

```
        self.p = p
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        """Perform DBSCAN clustering from features or distance matrix.

#This function provides an interface for the fitting aspect.

        Parameters
        ----------
        X : array or sparse (CSR) matrix of shape (n_samples, n_features), or \
            array of shape (n_samples, n_samples)
            A feature array, or array of distances between samples if
            ``metric='precomputed'``.
        sample_weight : array, shape (n_samples,), optional
            Weight of each sample, such that a sample with a weight of at least
            ``min_samples`` is by itself a core sample; a sample with negative
            weight may inhibit its eps-neighbor from being core.
            Note that weights are absolute, and default to 1.
        """
        X = check_array(X, accept_sparse='csr')

        clust = dbscan(X, sample_weight=sample_weight, **self.get_params())
        self.core_sample_indices_, self.labels_ = clust

#This is as described above.

        if len(self.core_sample_indices_):
            # fix for scipy sparse indexing issue
            self.components_ = X[self.core_sample_indices_].copy()

             #makes a copy of the core samples as taken from the original data and sets it to
“components”

        else:
            # no core samples
            self.components_ = np.empty((0, X.shape[1]))

        return self

    def fit_predict(self, X, y=None, sample_weight=None):
        """Performs clustering on X and returns cluster labels.
```

#This function is here simply to provide parity with other scikits functions…..it doesn't do anything other than the 'fit' implementation, and returning the labels.

```
        Parameters
        ----------
        X : array or sparse (CSR) matrix of shape (n_samples, n_features), or \
            array of shape (n_samples, n_samples)
            A feature array, or array of distances between samples if
            ``metric='precomputed'``.
        sample_weight : array, shape (n_samples,), optional
            Weight of each sample, such that a sample with a weight of at least
            ``min_samples`` is by itself a core sample; a sample with negative
            weight may inhibit its eps-neighbor from being core.
            Note that weights are absolute, and default to 1.

        Returns
        -------
        y : ndarray, shape (n_samples,)
            cluster labels
        """
        self.fit(X, sample_weight=sample_weight)
```

#provides fitting to the current sample weights and returns the labels.

```
        return self.labels_
```