```
"""
Kernel Density Estimation
-------------------------
"""
# Author: Jake Vanderplas <jakevdp@cs.washington.edu>

import numpy as np
from scipy.special import gammainc
from ..base import BaseEstimator
from ..utils import check_array, check_random_state
from ..utils.extmath import row_norms
from .ball_tree import BallTree, DTYPE
from .kd_tree import KDTree


VALID_KERNELS = ['gaussian', 'tophat', 'epanechnikov', 'exponential', 'linear',
                 'cosine']
TREE_DICT = {'ball_tree': BallTree, 'kd_tree': KDTree}


# TODO: implement a brute force version for testing purposes
# TODO: bandwidth estimation
# TODO: create a density estimation base class?
      KernelDensity(BaseEstimator):
    """Kernel Density Estimation

    Parameters
    ----------
    bandwidth : float
        The bandwidth of the kernel.

    algorithm : string
        The tree algorithm to use.  Valid options are
        ['kd_tree'|'ball_tree'|'auto'].  Default is 'auto'.

    kernel : string
        The kernel to use.  Valid kernels are
        ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine']
        Default is 'gaussian'.

    metric : string
        The distance metric to use.  Note that not all metrics are
        valid with all algorithms.  Refer to the documentation of
        :class:`BallTree` and :class:`KDTree` for a description of
        available algorithms.  Note that the normalization of the density
        output is correct only for the Euclidean distance metric. Default
        is 'euclidean'.

    atol : float
        The desired absolute tolerance of the result.  A larger tolerance will
        generally lead to faster execution. Default is 0.

    rtol : float
        The desired relative tolerance of the result.  A larger tolerance will
        generally lead to faster execution.  Default is 1E-8.

    breadth_first : boolean
        If true (default), use a breadth-first approach to the problem.
        Otherwise use a depth-first approach.

    leaf_size : int
        Specify the leaf size of the underlying tree.  See :class:`BallTree`
        or :class:`KDTree` for details.  Default is 40.

    metric_params : dict
        Additional parameters to be passed to the tree for use with the
        metric.  For more information, see the documentation of
        :class:`BallTree` or :class:`KDTree`.
    """
        __init__(self, bandwidth=1.0, algorithm='auto',
                 kernel='gaussian', metric="euclidean", atol=0, rtol=0,
                 breadth_first=True, leaf_size=40, metric_params=None):
```



```
        self.algorithm = algorithm
        self.bandwidth = bandwidth
        self.kernel = kernel
        self.metric = metric
        self.atol = atol
        self.rtol = rtol
        self.breadth_first = breadth_first
        self.leaf_size = leaf_size
        self.metric_params = metric_params

        # run the choose algorithm code so that exceptions will happen here
        # we're using clone() in the GenerativeBayes classifier,
        # so we can't do this kind of logic in __init__
        self._choose_algorithm(self.algorithm, self.metric)

           bandwidth <= 0:
                ValueError("bandwidth must be positive")
           kernel        VALID_KERNELS:
                ValueError("invalid kernel: '{0}'".format(kernel))

        _choose_algorithm(self, algorithm, metric):
        # given the algorithm string + metric string, choose the optimal
        # algorithm to compute the result.
           algorithm == 'auto':
            # use KD Tree if possible
               metric    KDTree.valid_metrics:
                       'kd_tree'
                  metric    BallTree.valid_metrics:
                       'ball_tree'
                 :
                      ValueError("invalid metric: '{0}'".format(metric))
            algorithm    TREE_DICT:
              metric        TREE_DICT[algorithm].valid_metrics:
                      ValueError("invalid metric for {0}: "
                                 "'{1}'".format(TREE_DICT[algorithm],
                                                metric))
                  algorithm
            :
                ValueError("invalid algorithm: '{0}'".format(algorithm))

        fit(self, X, y=None):
        """Fit the Kernel Density model on the data.

        Parameters
        ----------
        X : array_like, shape (n_samples, n_features)
            List of n_features-dimensional data points.  Each row
            corresponds to a single data point.
        """
        algorithm = self._choose_algorithm(self.algorithm, self.metric)
        X = check_array(X, order='C', dtype=DTYPE)

        kwargs = self.metric_params
           kwargs    None:
            kwargs = {}
        self.tree_ = TREE_DICT[algorithm](X, metric=self.metric,
                                          leaf_size=self.leaf_size,
                                          **kwargs)
               self

        score_samples(self, X):
        """Evaluate the density model on the data.

        Parameters
        ----------
        X : array_like, shape (n_samples, n_features)
            An array of points to query.  Last dimension should match dimension
            of training data (n_features).

        Returns
        -------
        density : ndarray, shape (n_samples,)
```

```
        The array of log(density) evaluations.
    """
    # The returned density is normalized to the number of points.
    # For it to be a probability, we must scale it.  For this reason
    # we'll also scale atol.
    X = check_array(X, order='C', dtype=DTYPE)
    N = self.tree_.data.shape[0]
    atol_N = self.atol * N
    log_density = self.tree_.kernel_density(
        X, h=self.bandwidth, kernel=self.kernel, atol=atol_N,
        rtol=self.rtol, breadth_first=self.breadth_first, return_log=True)
    log_density -= np.log(N)
           log_density

    score(self, X, y=None):
    """Compute the total log probability under the model.

    Parameters
    ----------
    X : array_like, shape (n_samples, n_features)
        List of n_features-dimensional data points.  Each row
        corresponds to a single data point.

    Returns
    -------
    logprob : float
        Total log-likelihood of the data in X.
    """
           np.sum(self.score_samples(X))

    sample(self, n_samples=1, random_state=None):
    """Generate random samples from the model.

    Currently, this is implemented only for gaussian and tophat kernels.

    Parameters
    ----------
    n_samples : int, optional
        Number of samples to generate. Defaults to 1.

    random_state : RandomState or an int seed (0 by default)
        A random number generator instance.

    Returns
    -------
    X : array_like, shape (n_samples, n_features)
        List of samples.
    """
    # TODO: implement sampling for other valid kernel shapes
       self.kernel        ['gaussian', 'tophat']:
              NotImplementedError()

    data = np.asarray(self.tree_.data)

    rng = check_random_state(random_state)
    i = rng.randint(data.shape[0], size=n_samples)

       self.kernel == 'gaussian':
              np.atleast_2d(rng.normal(data[i], self.bandwidth))

         self.kernel == 'tophat':
        # we first draw points from a d-dimensional normal distribution,
        # then use an incomplete gamma function to map them to a uniform
        # d-dimensional tophat distribution.
        dim = data.shape[1]
        X = rng.normal(size=(n_samples, dim))
        s_sq = row_norms(X, squared=True)
        correction = (gammainc(0.5 * dim, 0.5 * s_sq) ** (1. / dim)
                      * self.bandwidth / np.sqrt(s_sq))
              data[i] + X * correction[:, np.newaxis]
```