```python
# /usr/bin/python
# -*- coding: utf-8 -*-

"""
Kernel matrix approximation methods.

Authors: <alberto.suarez@uam.es>
         Luis Antonio Ortega Andrés
         Antonio Coín Castro
"""

from __future__ import annotations
import warnings
from abc import ABC, abstractmethod
from typing import Callable, Union, Optional

import matplotlib.pyplot as plt
import numpy as np
import scipy as sp

from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.gaussian_process.kernels import RBF


class RandomFeaturesSampler(ABC, BaseEstimator, TransformerMixin):
    """ Base class for random feature samplers. """

    def __init__(self, n_components: int = 100) -> None:
        """
        Initialize a Random Features sampler.

        Parameters
        ----------
        n_components:
            Number of random features to extract. Must be even.
        """
        self.n_components = n_components

        # Initialize default values
        self.w = None

    @abstractmethod
    def fit(
        self,
        X: np.darray,
        y: Optional[np.darray] = None,
    ) -> None:
        """
        Initialize w's for the random features.
```

```
        This should be implemented for each kernel.

        Parameters
        ----------
        X:
            Data matrix of shape (n_instances, n_features).
        y:
            Unused parameter for compatibility with sklearn's interface.
        """
        pass

    def transform(
        self,
        X: np.ndarray,
    ) -> np.ndarray:
        """
        Compute the random features.

        Assumes that the vector of w's has been initialized.

        Parameters
        ----------
        X:
            Data matrix of shape (n_instances, n_features).

        Returns
        -------
        random_features:
            Array of shape (n_instances, self.n_components).
        """
        if self.w is None:
            raise ValueError('Use fit to initialize w.')

        n_instances, n_features = np.shape(X)

        if np.shape(self.w)[1] != n_features:
            raise ValueError('Different # of features for X and w.')

        # Monte Carlo approximation
        random_features = np.empty((n_instances, self.n_components))
        random_features[:, ::2] = np.cos(X@self.w.T)
        random_features[:, 1::2] = np.sin(X@self.w.T)

        # Normalize features
        norm_factor = np.sqrt(self.n_components//2)
        random_features = random_features/norm_factor

        return random_features
```

```python
class RandomFeaturesSamplerRBF(RandomFeaturesSampler):
    """ Random Fourier Features for the RBF kernel. """

    def __init__(
        self,
        n_components: int = 100,
        sigma: float = 1.0,
        random_state: Optional[int] = None,
    ) -> None:
        """
        Initialize a Random Features sampler based on a RBF kernel.

        Parameters
        ----------
        n_components:
            Number of random features to extract.
        sigma:
            Standard deviation of RBF kernel. The covariance matrix will
            be Cov = (1.0/sigma^2)*I.
        random_state:
            Random seed.
        """
        super().__init__(n_components)
        self.sigma = sigma
        self.random_state = random_state

    def fit(
        self,
        X: np.darray,
        y: Optional[np.darray] = None,
    ) -> RandomFeaturesSamplerRBF:
        """
        Compute w's for the random RBF features.

        In this case, the RBF kernel is the characteristic function
        of a certain multivariate normal distribution.

        Parameters
        ----------
        X:
            Data matrix of shape (n_instances, n_features).
        y:
            Unused parameter for compatibility with sklearn's interface.

        Returns
        -------
        self:
            The instance itself.
        """
        n_features = X.shape[1]
```

```
        w_mean = np.zeros(n_features)
        w_cov_matrix = (1.0/self.sigma**2)*np.identity(n_features)

        # Sample from multivariate normal distribution
        rng = np.random.default_rng(seed=self.random_state)
        self.w = rng.multivariate_normal(
            w_mean,
            w_cov_matrix,
            self.n_components//2,
        )

        return self


class RandomFeaturesSamplerMatern(RandomFeaturesSampler):
    """ Random Fourier Features for the Matérn kernel. """

    def __init__(
        self,
        n_components: int = 100,
        scale: float = 1.0,
        nu: float = 1.0,
        random_state: Optional[int] = None,
    ) -> None:
        """
        Initialize a Random Features sampler based on a Matérn Kernel.

        Parameters
        ----------
        n_components:
            Number of random features to extract.
        scale:
            Length scale of the Matérn kernel.
        nu:
            Degrees of freedom of the Matérn kernel.
        random_state:
            Random seed.
        """
        super().__init__(n_components)
        self.scale = scale
        self.nu = nu
        self.random_state = random_state

    def fit(
        self,
        X: np.darray,
        y: np.darray = Optional[None],
    ) -> RandomFeaturesSamplerMatern:
        """
        Compute w's for the random Matérn features.
```

```
        The Fourier transform of the Matérn kernel is a Student's t
        distribution with twice the degrees of freedom.

            (Ref.) Chapter 4 of Carl Edward Rasmussen and Christopher K. I.
            Williams. 2005. Gaussian Processes for Machine Learning
            (Adaptive Computation and Machine Learning). The MIT Press.

        [There is probably a mistake with the scale factor.]

        Parameters
        ----------
        X:
            Data matrix of shape (n_instances, n_features).
        y:
            Unused parameter for compatibility with sklearn's interface.

        Returns
        -------
        self:
            The instance itself.
        """
        n_features = X.shape[1]

        # Scale of the Fourier transform of the kernel
        w_mean = np.zeros(n_features)
        w_cov_matrix = (1.0/self.scale**2)*np.identity(n_features)

        # Sample from multivariate student t distribution
        self.w = random_multivariate_student_t(
            w_mean,
            w_cov_matrix,
            2.0*self.nu,
            self.n_components//2,
            self.random_state
        )

        return self


def random_multivariate_student_t(
    mean: np.ndarray,
    cov_matrix: np.ndarray,
    df: float,
    n_samples: int,
    random_state: Optional[int] = None,
) -> np.ndarray:
    """
    Generate samples from a multivariate Student's t distribution.
```

```
        (Ref.) https://en.wikipedia.org/wiki/Multivariate_t-distribution

    This is a helper function for the RandomFeaturesSamplerMatern class.

    Parameters
    ----------
    mean:
        Mean vector of the distribution.
    cov_matrix:
        Covariance matrix of the distribution.
    df:
        Degrees of freedom.
    n_samples:
        Number of samples to generate.
    random_state:
        Random seed.

    Returns
    -------
    X:
        Array of shape (n_samples, len(mean)) with the generated samples.
    """
    # Dimensions of multivariate Student's t distribution.
    D = len(mean)

    # Formula for generating samples of a Student's t
    rng = np.random.default_rng(seed=random_state)
    x = rng.chisquare(df, n_samples)/df
    Z = rng.multivariate_normal(
        np.zeros(D),
        cov_matrix,
        n_samples,
    )
    X = mean + Z/np.sqrt(x)[:, np.newaxis]

    return X


class NystroemFeaturesSampler(BaseEstimator, TransformerMixin):
    """ Sample features following the Nyström method. """

    def __init__(
        self,
        n_components: int = 100,
        kernel: Callable[[np.ndarray, np.ndarray], np.ndarray] = RBF(),
        random_state: Optional[int] = None,
    ) -> None:
        """
        Initialize Nyström Features sampler.
```

```
        Parameters
        ----------
        n_components:
            Number of features to extract.
        kernel:
            Underlying kernel function.
        random_state:
            Random seed.
        """
        self.n_components = n_components
        self.kernel = kernel
        self.random_state = random_state

        # Initialize default values
        self.component_indices = None
        self.X_reduced = None
        self.reduced_kernel_matrix = None
        self.sqrtm_pinv_reduced_kernel_matrix = None

    def fit(
        self,
        X: np.ndarray,
        y: Optional[np.darray] = None,
    ) -> NystroemFeaturesSampler:
        """
        Precompute auxiliary matrix (W+)^1/2 for Nyström features.

        Parameters
        ----------
        X:
            Data matrix of shape (n_instances, n_features), ideally
            verifying that n_features >= self.n_components.
        y:
            Unused parameter for compatibility with sklearn's interface.

        Returns
        -------
        self:
            The instance itself.
        """
        n_instances = len(X)
        if self.n_components > n_instances:
            n_components = n_instances
            warnings.warn("n_components > n_samples, so n_components was set"
                          "to n_samples, which results in an inefficient"
                          " evaluation of the full kernel.")

        else:
            n_components = self.n_components
```

```
        # Sample subset of training instances
        rng = np.random.default_rng(seed=self.random_state)
        self.component_indices = rng.choice(
            range(n_instances),
            size=n_components,
            replace=False,
        )
        self.X_reduced = X[self.component_indices, :]

        # Compute reduced kernel matrix
        self.reduced_kernel_matrix = self.kernel(
            self.X_reduced,
            self.X_reduced
        )

        # Enforce symmetry of kernel matrix
        self.reduced_kernel_matrix = (
            self.reduced_kernel_matrix + self.reduced_kernel_matrix.T
        )/2.0

        # Compute the matrix (W+)^1/2
        self.sqrtm_pinv_reduced_kernel_matrix = sp.linalg.sqrtm(
            np.linalg.pinv(
                self.reduced_kernel_matrix,
                rcond=1.0e-6,
                hermitian=True
            )
        )

        # Check that complex part is negligible and eliminate it
        if np.iscomplexobj(self.sqrtm_pinv_reduced_kernel_matrix):
            threshold_imaginary_part = 1.0e-6
            max_imaginary_part = np.max(
                np.abs(np.imag(self.sqrtm_pinv_reduced_kernel_matrix))
            )

            if max_imaginary_part > threshold_imaginary_part:
                warnings.warn(
                    'Maximum imaginary part is {}'.format(max_imaginary_part)
                )

            self.sqrtm_pinv_reduced_kernel_matrix = np.real(
                self.sqrtm_pinv_reduced_kernel_matrix
            )

        return self

    def approximate_kernel_matrix(
        self,
        X: np.ndarray,
```

```
        X_prime: Optional[np.ndarray] = None,
    ) -> np.ndarray:
        """
        Approximate a kernel matrix using Nyström features.

        Parameters
        ----------
        X:
            Data matrix of shape (N, D).
        X_prime:
            Optional data matrix of shape (L, D).

        Returns
        -------
        The approximated kernel matrix of k(X_prime, X) if X_prime is
        present, or else the approximated kernel matrix of k(X, X).
        """
        if X_prime is None:
            X_prime = X
        X_features = self.fit_transform(X)
        X_prime_features = self.transform(X_prime)
        return X_prime_features@X_features.T

    def transform(
        self,
        X: np.ndarray,
    ) -> np.ndarray:
        """
        Compute Nyström features using fitted quantities.

        Parameters
        ----------
        X:
            Data matrix.

        Returns
        -------
        Array of Nyström features of X.
        """
        J = self.kernel(X, self.X_reduced)
        return J@self.sqrtm_pinv_reduced_kernel_matrix


def demo_kernel_approximation_features(
    X: np.ndarray,
    kernel: Callable[[np.ndarray, np.ndarray], np.ndarray],
    features_sampler: Union[RandomFeaturesSampler, NystroemFeaturesSampler],
    n_random_features: np.array,
) -> None:
    """
```

```
    Kernel approximation using Random Fourier features (RFF) or Nyström method.

    It shows a graph of each approximated kernel and also the mean and max
    absolute error of the approximation.

    Parameters
    ----------
    X:
        Data matrix.
    kernel:
        Kernel function that represents the kernel matrix to approximate.
    features_sampler:
        Object representing the sampling strategy initialized with the number
        of features to extract.
    n_random_features:
        Array with a collection of numbers of random features to sample.
    """
    # Set plot options
    n_plots = len(n_random_features) + 1
    fig, axes = plt.subplots(1, n_plots)
    fig.set_size_inches(15, 4)
    font = {'fontname': 'arial', 'fontsize': 18}

    # Plot original kernel
    kernel_matrix = kernel(X, X)
    axes[0].imshow(kernel_matrix, cmap=plt.cm.Blues)
    axes[0].set_title('Exact kernel', **font)
    axes[0].set_xticks([])
    axes[0].set_yticks([])

    # Plot kernel approximations
    for n, ax in zip(n_random_features, axes[1:]):
        # print('[Kernel approximation] # of features = ', n)

        # Get kernel matrix approximation
        X_features = features_sampler.set_params(
            n_components=n
        ).fit_transform(X)
        kernel_matrix_approx = X_features@X_features.T

        # Plot approximation
        ax.imshow(kernel_matrix_approx, cmap=plt.cm.Blues)

        # Compute and plot approximation errors
        err_approx = kernel_matrix - kernel_matrix_approx
        err_mean = np.mean(np.abs(err_approx))
        err_max = np.max(np.abs(err_approx))
        ax.set_xlabel('err (mean) = {:.4f} \n err (max) = {:.4f}'.format(
            err_mean,
            err_max
```

```
        ), **font)

        ax.set_title('{} features'.format(n), **font)
        ax.set_xticks([])
        ax.set_yticks([])
        plt.tight_layout()
    plt.show()


def plot_mean_approx_err(
    X: np.ndarray,
    kernel: Callable[[np.ndarray, np.ndarray], np.ndarray],
    features_sampler: Union[RandomFeaturesSampler, NystroemFeaturesSampler],
    max_features: int,
    start: int = 2,
    step: int = 2,
) -> np.ndarray:
    """
    Explore the dependence of mean approximation error w.r.t number of features.

    Parameters
    ----------
    X:
        Data matrix.
    kernel:
        Kernel function that represents the kernel matrix to approximate.
    features_sampler:
        Object representing the sampling strategy initialized with the number
        of features to extract.
    max_features:
        Sets the final number of random features.
    start:
        Sets the initial number of random features.
    step:
        Controls how the number of features increases on each iteration.

    Returns
    -------
    features_range:
        Array with all the number of random features tried.
    """
    K = kernel(X, X)
    mean_errs = []
    features_range = np.arange(start, max_features + 1, step)

    # Compute array of mean approximation errors for each n
    for n in features_range:
        X_features = features_sampler.set_params(
            n_components=n
        ).fit_transform(X)
```

```
        K_hat = X_features@X_features.T

        mean_errs.append(np.mean(np.abs(K - K_hat)))

    # Plot error vs n_components
    plt.figure(figsize=(7, 4))
    plt.title("Evolution of mean approximation error")
    plt.xlabel("n_features_sampled")
    plt.ylabel("mean absolute error")
    plt.plot(features_range, mean_errs, label="Empirical error")

    return features_range


if __name__ == '__main__':
    """
    from sklearn import datasets, svm
    from sklearn.kernel_approximation import RBFSampler
    from sklearn.metrics.pairwise import rbf_kernel

    # A not so simple 2-D problem
    X, Y = datasets.make_moons(n_samples=100, noise=0.3, random_state=0)

    # Compute grid of points for plotting the decision regions
    grid_x, grid_y = np.meshgrid(
        np.linspace(-3, 3, 50),
        np.linspace(-3, 3, 50)
    )
    grid_X = np.c_[grid_x.ravel(), grid_y.ravel()]

    # Kernel matrix
    gamma = 0.5
    def kernel(X, Y):
        return rbf_kernel(X, Y, gamma=gamma)

    # Nyström features
    n_nystroem_features = 20
    nystroem_sampler = NystroemFeaturesSampler(n_nystroem_features, kernel)
    nystroem_features = nystroem_sampler.fit_transform(X)
    nystroem_features_grid = nystroem_sampler.transform(grid_X)

    # Classifier
    clf = svm.SVC(kernel='linear')
    # clf = svm.NuSVC(gamma='auto')
    clf.fit(nystroem_features, Y)
    """

    from sklearn import datasets
    from sklearn.metrics.pairwise import rbf_kernel
```

```
# 3-D data
n_instances = 1000
X, t = datasets.make_s_curve(n_instances, noise=0.1)
X = X[np.argsort(t)]

# 2-D data
# X, y = datasets.make_moons(n_samples=400, noise=.05)
# X = X[np.argsort(y)]

# Reshape if necessary
if (X.ndim == 1):
    X = X[:, np.newaxis]

# Kernel parameters
sigma = 1.0
gamma = 1.0/(2.0*sigma**2)

# Kernel function
def kernel(X, Y):
    return rbf_kernel(X, Y, gamma=gamma)

# Nyström features
n_nystroem_features = [10, 100, 1000]
nystroem_sampler = NystroemFeaturesSampler(n_nystroem_features, kernel)

demo_kernel_approximation_features(
    X,
    kernel,
    nystroem_sampler,
    n_nystroem_features
)
```