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

"""
Kernel matrix approximation methods.

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

from __future__ import annotations
import warnings
import timeit
from typing import (
    Callable, Union, Optional, List, Dict, Tuple
)

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
from sklearn.model_selection import (
    train_test_split, GridSearchCV,
    StratifiedKFold, cross_val_score
)


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

    def __init__(
        self,
        n_components: int = 100,
        sampling_method: str = 'sin+cos',
    ) -> None:
        """
        Initialize a Random Features sampler.

        Parameters
        ----------
        n_components:
            Number of random features to extract.
        sampling_method:
```

```
            Sampling strategy. Must be one of
                - 'sin+cos'
                - 'cos'
        """
        self.n_components = n_components
        self.sampling_method = sampling_method

    def fit(
        self,
        X: np.darray,
        y: None = None,
    ) -> RandomFeaturesSampler:
        """
        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.

        Returns
        -------
        self:
            The instance itself.
        """
        self._w = None

        if self.sampling_method == 'sin+cos':
            self._n_random_samples_w = self.n_components//2
        elif self.sampling_method == 'cos':
            self._n_random_samples_w = self.n_components
        else:
            raise ValueError('Please enter a correct sampling method')

        return self

    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.')

        if self.sampling_method == 'sin+cos':
            random_features = np.empty(
                (n_instances, 2*self._n_random_samples_w)
            )
            random_features[:, ::2] = np.cos(X@self._w.T)
            random_features[:, 1::2] = np.sin(X@self._w.T)
            normalization_factor = np.sqrt(self._n_random_samples_w)

        elif self.sampling_method == 'cos':
            rng = np.random.default_rng(seed=self.random_state)
            b = rng.uniform(0, 2*np.pi, self._n_random_samples_w)
            random_features = np.cos(X@self._w.T + b)
            normalization_factor = np.sqrt(self._n_random_samples_w/2.0)

        else:
            raise ValueError('Please enter a correct sampling method')

        random_features = random_features/normalization_factor

        return random_features


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

    def __init__(
```

```
        self,
        n_components: int = 100,
        sampling_method: str = 'sin+cos',
        sigma_kernel: 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.
        sampling_method:
            Sampling strategy (@see RandomFeaturesSampler.__init__).
        sigma_kernel:
            Standard deviation of RBF kernel. The covariance matrix will
            be Cov = (1.0/sigma^2)*I.
        random_state:
            Random seed.
        """
        super().__init__(n_components, sampling_method)
        self.sigma_kernel = sigma_kernel
        self.random_state = random_state

    def fit(
        self,
        X: np.darray,
        y: None = 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.
            """
            super().fit(X)

            n_features = np.shape(X)[1]
            w_mean = np.zeros(n_features)
            w_cov_matrix = (1.0/self.sigma_kernel**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_random_samples_w,
            )

            return self


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

        def __init__(
            self,
            n_components: int = 100,
            sampling_method: str = 'sin+cos',
            length_scale_kernel: float = 1.0,
            nu_kernel: 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.
            sampling_method:
                Sampling strategy (@see RandomFeaturesSampler.__init__).
            length_scale_kernel:
                Length scale of the Matérn kernel.
            nu_kernel:
                Degrees of freedom of the Matérn kernel.
            random_state:
```

```
        Random seed.
    """
    super().__init__(n_components, sampling_method)
    self.length_scale_kernel = length_scale_kernel
    self.nu_kernel = nu_kernel
    self.random_state = random_state

def fit(
    self,
    X: np.darray,
    y: None = 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.
    """
    super().fit(X)

    n_features = np.shape(X)[1]
    w_mean = np.zeros(n_features)
    w_cov_matrix = (1.0/self.length_scale_kernel**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_kernel,
                self._n_random_samples_w,
                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 RandomFeaturesSamplerExp(RandomFeaturesSampler):
        """ Random Fourier Features for the exponential kernel. """

        def __init__(
            self,
            n_components: int = 100,
            sampling_method: str = 'sin+cos',
            length_scale_kernel: float = 1.0,
            random_state: Optional[int] = None,
        ) -> None:
            """
            Initialize a Random Features sampler based on an exponential kernel.

            Parameters
            ----------
            n_components:
                Number of random features to extract.
            sampling_method:
                Sampling strategy (@see RandomFeaturesSampler.__init__).
            length_scale_kernel:
                Length scale of the kernel.
            random_state:
                Random seed.
            """
            super().__init__(n_components, sampling_method)
            self.length_scale_kernel = length_scale_kernel
            self.random_state = random_state

        def fit(
            self,
            X: np.darray,
            y: None = None,
```

```
) -> RandomFeaturesSamplerExp:
    """
    Compute w's for the random exponential 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.
    """
    super().fit(X)

    n_features = np.shape(X)[1]
    self._w = self._sample_exp_kernel(
        n_features,
        self._n_random_samples_w
    )

    return self

def _sample_exp_kernel(
    self,
    dim: int,
    n_samples: int,
) -> np.ndarray:
    """
    Sample points from the mulidimensional pdf of the exponential kernel.

    This pdf is the inverse Fourier transform of the exponential kernel,
    which factorizes as the product of one-dimensional pdfs. We perform
    the sampling using the method of the inverse.

    Parameters
    ----------
    dim:
        Dimension of the kernel.
```

```
        n_samples:
            Number of samples to generate.
        Returns
        -------
        X:
            Array of shape (n_samples, dim) with the generated samples.
        """
        # Function that defines the corresponding inverse cdf
        def exp_kernel_inverse_cdf(p, gamma):
            return np.tan(np.pi*(p - 0.5))/gamma

        # Apply the method of the inverse
        rng = np.random.default_rng(seed=self.random_state)
        U = rng.uniform(size=(n_samples, dim))  # U ~ U([0, 1]^D)
        X = exp_kernel_inverse_cdf(U, self.length_scale_kernel)

        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

    def fit(
```

```
    self,
    X: np.ndarray,
    y: None = 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_instances, so n_components was set"
                      "to n_instances, 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
        -------
        kernel_matrix_approx:
            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)
        kernel_matrix_approx = X_prime_features@X_features.T

        return kernel_matrix_approx

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

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

        Returns
        -------
        X_nystroem:
            Array of Nyström features of X.
        """
        reduced_kernel_matrix_columns = self.kernel(X, self._X_reduced)

        X_nystroem = (
            reduced_kernel_matrix_columns
            @ self._sqrtm_pinv_reduced_kernel_matrix
        )

        return X_nystroem


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,
    fig_num: int = 1,
) -> 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.
    fig_num:
        Matplotlib internal parameter for figure number.
    """
    # Set plot options
    n_plots = len(n_random_features) + 1
    fig, axes = plt.subplots(1, n_plots, num=fig_num)
    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:]):
        # 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 demo_cv_search(
    X: np.ndarray,
    y: np.ndarray,
    models: List[Tuple],
    parameters: List[Dict],
    n_train: float = 0.5,
    n_folds: int = 5,
    n_runs: int = 1,
    verbose: bool = False,
    seed: Optional[int] = None,
) -> List[np.ndarray]:
    """
    Perform cross-validation to find optimal hyperparameters.

    The data is first divided on training and test sets. Then a
    cross-validation grid search is performed, and finally the
    refitted optimal estimator is evaluated on the test set.
    The process can be repeated with different random training/test
    partitions to increase stability.

    Parameters
    ----------
    X:
```

```
    Data matrix.
y:
    Class labels.
models:
    List of tuples representing different models, each containing
    a sklearn.pipeline.Pipeline and a name.
parameters:
    List of dictionaries containing the search space for each model.
n_train:
    Fraction of the data set for training.
n_folds:
    Number of (stratified) folds in CV.
n_runs:
    Number of repetitions with different train/test partitions.
verbose:
    Whether to print the the training/test error and optimal
    hyperparameters for each model.
seed:
    Seed for randomness. Must be a non-negative integer.

Returns
-------
stats_dict:
    Dictionary of arrays with statistical information regarding
    the whole process.
"""
n_models = len(models)

# Initialize statistics arrays
cv_times = np.zeros((n_runs, n_models))
train_times = np.zeros((n_runs, n_models))
test_times = np.zeros((n_runs, n_models))
cv_errors = np.zeros((n_runs, n_models))
train_errors = np.zeros((n_runs, n_models))
test_errors = np.zeros((n_runs, n_models))
best_params = []

# Repeat to increase stability
for i in range(n_runs):
    best_params.append([])

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        X,
        y,
```

```
        train_size=n_train,
        stratify=y,
        random_state=(i + 1)*(seed + 1)  # Different splits across runs
    )

    # Set CV parameters
    folds = StratifiedKFold(
        n_folds,
        shuffle=True,
        random_state=seed
    )

    # Perform grid search
    search_space = zip(models, parameters)
    for j, ((model, name), param_grid) in enumerate(search_space):
        clf = GridSearchCV(
            model,
            param_grid,
            cv=folds,
            refit=True,
            n_jobs=-1,  # Use all available threads
            return_train_score=True
        )

        # Find best classifier and refit on the whole training set
        start = timeit.default_timer()
        clf.fit(X_train, y_train)
        cv_time = timeit.default_timer() - start - clf.refit_time_

        # Compute training score of best model (already refitted)
        train_score = clf.score(X_train, y_train)

        # Compute test score of best model
        start = timeit.default_timer()
        test_score = clf.score(X_test, y_test)
        test_time = timeit.default_timer() - start

        # Save statistical information
        cv_times[i, j] = cv_time
        train_times[i, j] = clf.refit_time_
        test_times[i, j] = test_time
        cv_errors[i, j] = 1.0 - clf.best_score_
        train_errors[i, j] = 1.0 - train_score
        test_errors[i, j] = 1.0 - test_score
        best_params[i].append(clf.best_params_)
```

```
            if verbose:
                print("--", name, "--")
                print(f"  Grid search time: {cv_time:.3f}s")
                print(f"  Test score: {test_score:.3f}")
                print("  Best parameters:\n", clf.best_params_)
                print("")

    stats_dict = {
        'cv_times': cv_times,
        'train_times': train_times,
        'test_times': test_times,
        'cv_errors': cv_errors,
        'train_errors': train_errors,
        'test_errors': test_errors,
        'best_params': best_params
    }

    return stats_dict


def plot_features_evolution(
    X: np.ndarray,
    y: np.ndarray,
    models: List[Tuple],
    n_features: np.ndarray,
    n_train: float = 0.5,
    n_folds: int = 5,
    n_runs: int = 1,
    fig_num: int = 1,
    seed: Optional[int] = None,
) -> None:
    """
    Plot a curve with the evolution of the error as a function of n_features.

    The data is randomly divided on a training and test set.
    The curve plotted tracks the dependence on the number of features
    sampled for the cross-validation, training and test errors, while
    keeping the rest of parameters fixed.

    Parameters
    ----------
    X:
        Data matrix.
    y:
```

```
    Class labels.
models:
    List of tuples representing different models, each containing
    a sklearn.pipeline.Pipeline and a name. The models in question
    must have at least one transformer with a 'n_components' attribute.
n_features:
    Array with a collection of numbers of features to sample.
n_train:
    Fraction of the data set for training.
n_folds:
    Number of (stratified) folds in CV.
n_runs:
    Number of repetitions with different train/test partitions.
fig_num:
    Matplotlib internal parameter for figure number.
seed:
    Seed for randomness. Must be a non-negative integer.
"""
n_models = len(models)
n_features_len = len(n_features)

# Initialize results arrays
cv_errors = np.zeros((n_runs, n_models, n_features_len))
train_errors = np.zeros((n_runs, n_models, n_features_len))
test_errors = np.zeros((n_runs, n_models, n_features_len))

# Repeat to increase stability
for i in range(n_runs):
    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        X,
        y,
        train_size=n_train,
        stratify=y,
        random_state=(i + 1)*(seed + 1)
    )

    # Set CV parameters
    folds = StratifiedKFold(
        n_folds,
        shuffle=True,
        random_state=seed
    )

    # Measure each model
```

```
    for j, (model, name) in enumerate(models):
        # Use different n_features
        for k, n in enumerate(n_features):
            model[0].set_params(n_components=n)

            # Compute CV score for each fold
            cv_score = cross_val_score(
                model,
                X_train,
                y_train,
                cv=folds,
                n_jobs=-1,
            )

            # Train model on the whole training set
            model.fit(X_train, y_train)

            # Copute train and test scores
            train_score = model.score(X_train, y_train)
            test_score = model.score(X_test, y_test)

            cv_errors[i, j, k] = 1.0 - cv_score.mean()
            train_errors[i, j, k] = 1.0 - train_score
            test_errors[i, j, k] = 1.0 - test_score

# Plot curves
fig, axs = plt.subplots(1, 3, num=fig_num,
                        figsize=(15, 5), sharey=True)

for j, (model, name) in enumerate(models):
    axs[0].set_xlabel("n_features")
    axs[0].set_title("CV error")
    axs[1].set_xlabel("n_features")
    axs[1].set_title("Train error")
    axs[2].set_xlabel("n_features")
    axs[2].set_title("Test error")

    # Compute mean and std
    mean_cv = cv_errors[:, j, :].mean(axis=0)
    std_cv = cv_errors[:, j, :].std(axis=0)
    mean_train = train_errors[:, j, :].mean(axis=0)
    std_train = train_errors[:, j, :].std(axis=0)
    mean_test = test_errors[:, j, :].mean(axis=0)
    std_test = test_errors[:, j, :].std(axis=0)
```

```
        # Plot mean values
        axs[0].plot(n_features, mean_cv, label=name)
        axs[1].plot(n_features, mean_train, label=name)
        axs[2].plot(n_features, mean_test, label=name)

        # Fill with +-2 std
        axs[0].fill_between(
            n_features,
            mean_cv - 2*std_cv,
            mean_cv + 2*std_cv,
            alpha=0.2
        )
        axs[1].fill_between(
            n_features,
            mean_train - 2*std_train,
            mean_train + 2*std_train,
            alpha=0.2
        )
        axs[2].fill_between(
            n_features,
            mean_test - 2*std_test,
            mean_test + 2*std_test,
            alpha=0.2
        )

        axs[0].legend(loc='best')
        axs[1].legend(loc='best')
        axs[2].legend(loc='best')

plt.show()
```