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

"""
Kernel methods in Machine Learning.

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

from typing import Callable, Tuple, Optional
import matplotlib

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as anim
from scipy.spatial import distance

from sklearn.utils.extmath import svd_flip


def linear_kernel(
    X: np.ndarray,
    X_prime: np.ndarray,
) -> np.ndarray:
    """
    Parameters
    ----------
    X:
        Data matrix
    X_prime:
        Data matrix

    Returns
    -------
    kernel matrix
    """
    return X@X_prime.T


def exponential_kernel(
    X: np.ndarray,
    X_prime: np.ndarray,
    A: float,
    ls: float,
) -> np.ndarray:
    """
    Parameters
    ----------
    X:
        Data matrix
    X_prime:
        Data matrix
    A:
        Output variance
    ls:
        Kernel lengthscale

    Returns
    -------
    kernel matrix
    """
    d = distance.cdist(X, X_prime, metric='minkowski', p=1.0)
    return A*np.exp(-d/ls)


def rbf_kernel(
    X: np.ndarray,
    X_prime: np.ndarray,
    A: float,
    ls: float,
) -> np.ndarray:
    """
    Parameters
    ----------
    X:
        Data matrix
    X_prime:
        Data matrix
    A:
        Output variance
    ls:
        Kernel lengthscale

    Returns
    -------
    kernel matrix

    Notes
    -------
    Alternative parametrization (e.g. in sklearn)
    gamma = 0.5 / ls**2

    Example
    -------
    >>> import numpy as np
    >>> import matplotlib.pyplot as plt
    >>> import gaussian_process_regression as gp
    >>> X = np.array([[1,2], [3, 4], [5,6]])
    >>> X_prime = np.array([[1,2], [3, 4]])
    >>> A, l = 3, 10.0
    >>> kernel_matrix = gp.rbf_kernel(X, X_prime, A, l)
    >>> print(kernel_matrix)
    """
    d = distance.cdist(X, X_prime, metric='euclidean')
    return A*np.exp(-0.5*(d/ls)**2)


def compute_centered_gram_matrix(
    K1: np.ndarray,
    K2: Optional[np.ndarray] = None,
) -> np.ndarray:
    """
    Compute Gram matrix of centered kernel.

    Parameters
    ----------
    K1:
        An NxN kernel Gram matrix from training data.
    K2:
        An LxN kernel Gram matrix from test data.

    Notes
    -------
    If K2=None, it computes the Gram matrix of the
    centered kernel for training data. Otherwise it computes the
    kernel Gram matrix for test data.
    """
    N = K1.shape[0]
    L = K2.shape[0] if K2 is not None else N
    K2 = K1 if K2 is None else K2
    ones = np.ones((N, N))
    ones_prime = ones if K2 is None else np.ones((L, N))

    return K2 - 1/N*(K2@ones) - 1/N*(ones_prime@K1) \
        + 1/(N**2)*(ones_prime@K1@ones)


def kernel_pca(
    X: np.ndarray,
    X_test: np.ndarray,
    kernel: Callable,
    flip: bool = False,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Parameters
    ----------
    X:
        Data matrix
```

```
X_test:
    Data matrix
kernel:
    Kernel function
flip:
    Whether to impose sklearn's deterministic
    choice of normalized eigenvector signs.

Returns
-------
X_test_hat:
    Projection of X_test on the principal components corresponding
    to non-zero eigenvalues.
lambda_eigenvals:
    Eigenvalues of the centered kernel.
alpha_eigenvecs:
    Principal components. These are the eigenvectors
    of the centered kernel with the RKHS normalization.

Notes
-------
In the corresponding method of sklearn the eigenvectors
are normalized in l2.
"""
# Gram matrix of kernel
K = kernel(X, X)

# Gram matrix of centered kernel
K_hat = compute_centered_gram_matrix(K)

# Compute eigenvectors and eigenvalues (in ascending order)
lambda_eigenvals, alpha_eigenvecs = np.linalg.eigh(K_hat)

# Set negligible eigenvalues to zero
lambda_eigenvals[lambda_eigenvals < 1.0e-6] = 0.0

# Order eigenvalues and eigenvectors in descending order
lambda_eigenvals = lambda_eigenvals[::-1]
alpha_eigenvecs = alpha_eigenvecs[:, ::-1]

# Compute (centered) projection matrix
K_test = kernel(X_test, X)
K_test_hat = compute_centered_gram_matrix(K, K_test)

# Choose sign of eigenvectors in a deterministic way
if flip:
    alpha_eigenvecs, _ = svd_flip(alpha_eigenvecs,
                                  np.zeros_like(alpha_eigenvecs).T)

# RKHS normalization of eigenvectors, ignoring null components
```

```
    non_zero = np.flatnonzero(lambda_eigenvals)
    alpha_eigenvecs[:, non_zero] = (alpha_eigenvecs[:, non_zero]
                                    / np.sqrt(lambda_eigenvals[non_zero]))

    # Project principal components of non-zero eigenvalues
    X_test_hat = K_test_hat@alpha_eigenvecs[:, non_zero]

    return X_test_hat, lambda_eigenvals, alpha_eigenvecs


class AnimationKPCA:
    """ Animation of KPCA projection varying the
        width parameter of an RBF kernel. """

    def __init__(
        self,
        xlims: Tuple[float, float],
        ylims: Tuple[float, float],
        n_frames: int = 50,
    ) -> None:
        """
        Set initial parameters for the animation.

        Parameters
        ----------
        xlims:
            Limits for the x-axis.
        ylims:
            Limits for the y-axis.
        n_frames:
            Number of frames (i.e. different parameter values).
        """
        self.n_frames = n_frames
        self.gammas = 2*np.logspace(-3, 4, n_frames)
        self.A = 1.0
        self.L = 1.0
        self.xlims = xlims
        self.ylims = ylims

    def _init_plot(
        self,
        ax: matplotlib.axes.Axes,
        gamma: float,
    ) -> None:
        """
        Initialize axis labels, titles and limits.
        Also clear any previous plots.
        """
        ax.clear()
        ax.set_title(
            r"Projection by KPCA ($\gamma=$" + f"{gamma:.3f})")
        ax.set_xlabel(
            r"1st principal component in space induced by $\phi$")
        ax.set_ylabel("2nd principal component")
        ax.set_xlim(self.xlims)
        ax.set_ylim(self.ylims)

    def _update_plot(
        self,
        i: int,
        ax: matplotlib.axes.Axes,
        X: np.ndarray,
        X_test: np.ndarray,
        reds: np.ndarray,
        blues: np.ndarray,
    ) -> None:
        """
        Update output in animation by advancing frames.

        Parameters
        -----------
        i:
            Frame number.
        ax:
            Axis in which to plot.
        X:
            Training data matrix.
        X_test:
            Test data matrix.
        reds:
            Boolean array of test data with label 0.
        blues:
            Boolean array of test data with label 1.
        """
        gamma = self.gammas[i]
        self.L = np.sqrt(0.5/gamma)
        self._init_plot(ax, gamma)

        X_kpca, _, _ = kernel_pca(X, X_test,
                                  self.kernel,
                                  flip=True)

        ax.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
                   s=20, edgecolor='k')
        ax.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
                   s=20, edgecolor='k')

    def animate(
        self,
        X: np.ndarray,
        X_test: np.ndarray,
        y_test: np.ndarray,
    ) -> matplotlib.animation.FuncAnimation:
        """
        Make an animation with a given dataset.

        Parameters
        -----------
        X:
            Training data matrix.
        X_test:
            Test data matrix.
        y_test:
            Test data labels.
        """
        fig = plt.figure(figsize=(8, 5))
        ax = fig.add_subplot(111)

        def kernel(X, X_prime):
            return rbf_kernel(X, X_prime, self.A, self.L)

        self.kernel = kernel
        reds = y_test == 0
        blues = y_test == 1

        return anim.FuncAnimation(
            fig,
            self._update_plot,
            frames=self.n_frames,
            repeat=False,
            fargs=(ax, X, X_test, reds, blues,)
        )
```