demo_kernel_pca

March 4, 2021

Luis Antonio Ortega Andrés
Antonio Coín Castro

1 Kernel PCA

Adapted by alberto.suarez@uam.es from

Kernel PCA (sklearn)
Authors: Mathieu Blondel, Andreas Mueller

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML

from sklearn import datasets
from sklearn.decomposition import PCA, KernelPCA

import kernel_machine_learning as kpca

%load_ext autoreload
%autoreload 2

seed = 0
np.random.seed(seed)  # for reproducible results
```

1.1 PCA, KPCA and comparison with Sklearn

[2]:
```
# Input data
X, y = datasets.make_moons(n_samples=400,
                           noise=.05,
                           random_state=seed)
```

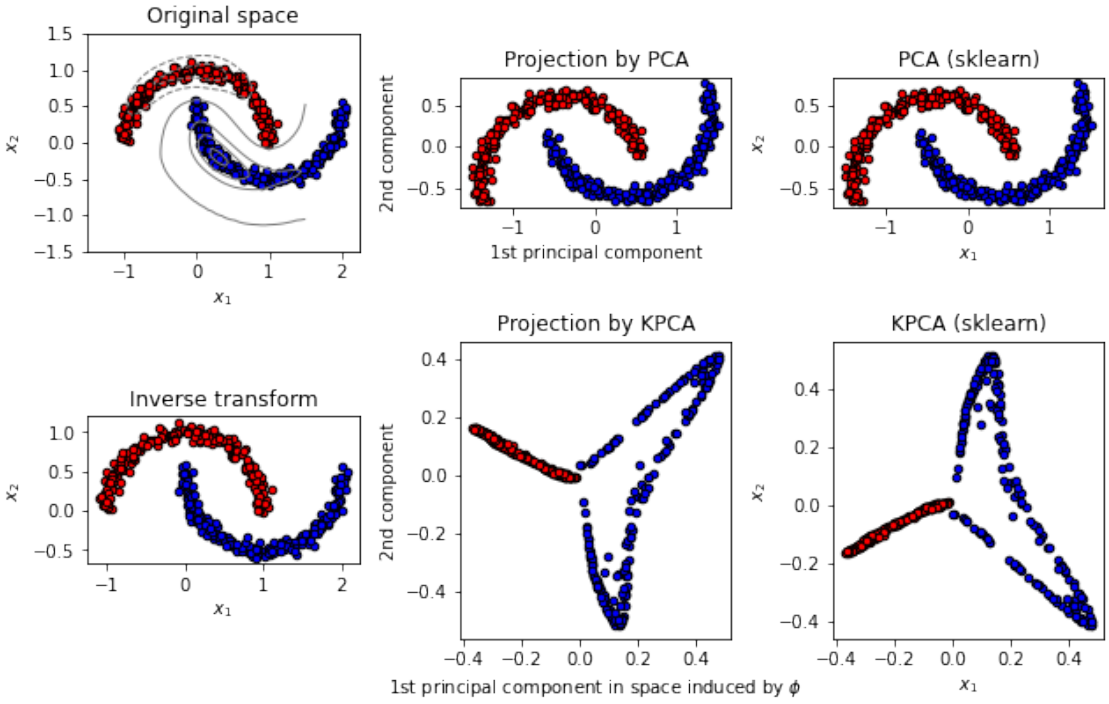
[3]:
```
# Principal componets

# PCA (linear)
X_pca, eigenvals_pca, eigenvecs_pca = \
    kpca.kernel_pca(X, X, kpca.linear_kernel)
```

```
# Kernel PCA
A = 1.0
gamma = 20.0
L = np.sqrt(0.5/gamma)

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


X_kpca, eigenvals_kpca, eigenvecs_kpca = \
    kpca.kernel_pca(X, X, rbf_kernel)

# PCA (sklearn)
pca = PCA()
X_pca_sk = pca.fit_transform(X)

# Kernel PCA (sklearn)
kernel_pca = KernelPCA(kernel='rbf',
                       fit_inverse_transform=True,
                       gamma=gamma)
X_kpca_sk = kernel_pca.fit_transform(X)
X_back = kernel_pca.inverse_transform(X_kpca_sk)
```

[4]:
```
# Plot results

plt.figure(figsize=(9, 6))
plt.subplot(2, 3, 1, aspect='equal')
plt.title('Original space')
reds = y == 0
blues = y == 1

# Original dataset
plt.scatter(X[reds, 0], X[reds, 1], c='red',
            s=20, edgecolor='k')
plt.scatter(X[blues, 0], X[blues, 1], c='blue',
            s=20, edgecolor='k')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Projection on the first principal component (in the phi space)
X1, X2 = np.meshgrid(np.linspace(-1.5, 1.5, 50),
                     np.linspace(-1.5, 1.5, 50))
X_grid = np.array([np.ravel(X1), np.ravel(X2)]).T
Z_grid = kernel_pca.transform(X_grid)[:, 0].reshape(X1.shape)
plt.contour(X1, X2, Z_grid, colors='grey', linewidths=1, origin='lower')
```

```
# Inverse transform
plt.subplot(2, 3, 4, aspect='equal')
plt.scatter(X_back[reds, 0], X_back[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_back[blues, 0], X_back[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Inverse transform")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

# PCA and KPCA projections
plt.subplot(2, 3, 2, aspect='equal')
plt.scatter(X_pca[reds, 0], X_pca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_pca[blues, 0], X_pca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by PCA")
plt.xlabel("1st principal component")
plt.ylabel("2nd component")

plt.subplot(2, 3, 5, aspect='equal')
plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("Projection by KPCA")
plt.xlabel(r"1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")

plt.subplot(2, 3, 3, aspect='equal')
plt.scatter(X_pca_sk[reds, 0], X_pca_sk[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_pca_sk[blues, 0], X_pca_sk[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("PCA (sklearn)")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

plt.subplot(2, 3, 6, aspect='equal')
plt.scatter(X_kpca_sk[reds, 0], X_kpca_sk[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca_sk[blues, 0], X_kpca_sk[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title("KPCA (sklearn)")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
```

```
plt.tight_layout()
plt.show()
```



1.2 KPCA animation

We make an animation that shows the evolution of the projections onto the first two KPCA components for $0.002 \leq \gamma \leq 20000.0$.

[5]:
```
animation = kpca.AnimationKPCA(n_frames=100,
                               xlims=(-1, 1),
                               ylims=(-1, 1)).animate(X, X, y)
plt.close()
HTML(animation.to_jshtml())
```

[5]: <IPython.core.display.HTML object>

1.3 Questions

1. Why do the projections onto the first two KPCA principal components look different for the Sklearn and our implementation? Is any of the two incorrect?

Both representations are correct and valid; what makes them different is the decision made when normalizing the eigenvectors. The eigenvectors returned via specialized functions such as `eig` and `eigh` (for symmetric cases) are normalized in $\mathbb{R}^N$, that is, they verify $\|\alpha_i\|_{\mathcal{L}_2} = 1$. However, what we really want is that the eigenvectors induced by these α_i be normalized in the subjacent Hilbert

space. As we saw in the class lectures, this can be achieved by ensuring that their $\mathcal{L}_2$-norm in $\mathbb{R}^N$ equals the inverse of the square root of their corresponding eigenvalues, $1/\sqrt{\lambda_i}$. Since the α_i are unitary, the procedure is simply to divide each eigenvector by this quantity. It is in this step that the discrepancy arises, as one may divide by $-1/\sqrt{\lambda_i}$ and still get the same $\mathcal{L}_2$-norm.

Sklearn's implementation uses an auxiliary function `svd_flip` that deterministically assigns a sign for each normalized eigenvector (see Sklearn's source code). We have implemented this option as a parameter for our function in order to reproduce Sklearn's output perfectly.

In our case, not using this function leads to a reflection of the second principal component in the Kernel PCA graph with respect to Sklearn's (all values of this vector are multiplied by -1). There is nothing special about the second principal component, only that the corresponding eigenvector had the sign flipped. Linear PCA does not show discrepancies by pure coincidence.

2. Vary the parameters of the kernel and comment on the behavior of the projections onto the first two KPCA components for the different values considered (e.g. $\gamma \in \{0.02, 0.2, 2.0, 20.0, 200.0, 2000.0\}$). In particular,

- *What is the behavior in the limit in which the width of the kernel approaches ∞?. Explain why one should expect such behavior.*
- *What is the behavior in the limit in which the width of the kernel approaches 0?. Explain why one should expect such behavior.*

Consider a generic set of points X, with N samples, and observe that in the code we only ever compute $\mathcal{K}(X, X)$, that is, the test set is the same as the training set.. First of all, to study the limit behavior of the projections we are using the explicit RBF kernel formula:

$$\mathcal{K}(x, y) = A \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right),$$

where A is the output variance and σ is the kernel's width. In our case, $A = 1.0$ and the parameter γ is inversely proportional to σ. Using this formula, and given two fixed input values $x \neq y$, we have:

$$\lim_{\gamma \to 0^+} = \lim_{\sigma \to \infty} \mathcal{K}(x, y) = A = 1.$$

$$\lim_{\gamma \to \infty} = \lim_{\sigma \to 0^+} \mathcal{K}(x, y) = 0.$$

Let us begin by analyzing the first limit. In this case, if $x = y$ we have $\mathcal{K}(x, y) = A = 1$ as well, so all in all we have:

$$\lim_{\gamma \to 0^+} \mathcal{K}(X, X) = 1,$$

where \$ 1\$ is a matrix with the same dimension as $K \equiv \mathcal{K}(X, X)$ containing all ones. Now observe that when we center this kernel, the expression in the limit is:

$$\hat{K} = 1 - (2/N)11 + (1/N^2)111 = 0,$$

that is, we get the null matrix. As a result, the limit case of the projection reduces to the matrix operation

$$0 \begin{pmatrix} e_1 & \dots e_N \end{pmatrix} = 0,$$

where $e_1, \ldots, e_N$ are the computed eigenvectors of a null matrix, i.e, any orthonormal basis in $\mathbb{R}^N$. When $\hat{K}$ is not entirely a null matrix but it is close to it, redoing the above calculations we may approximate it as

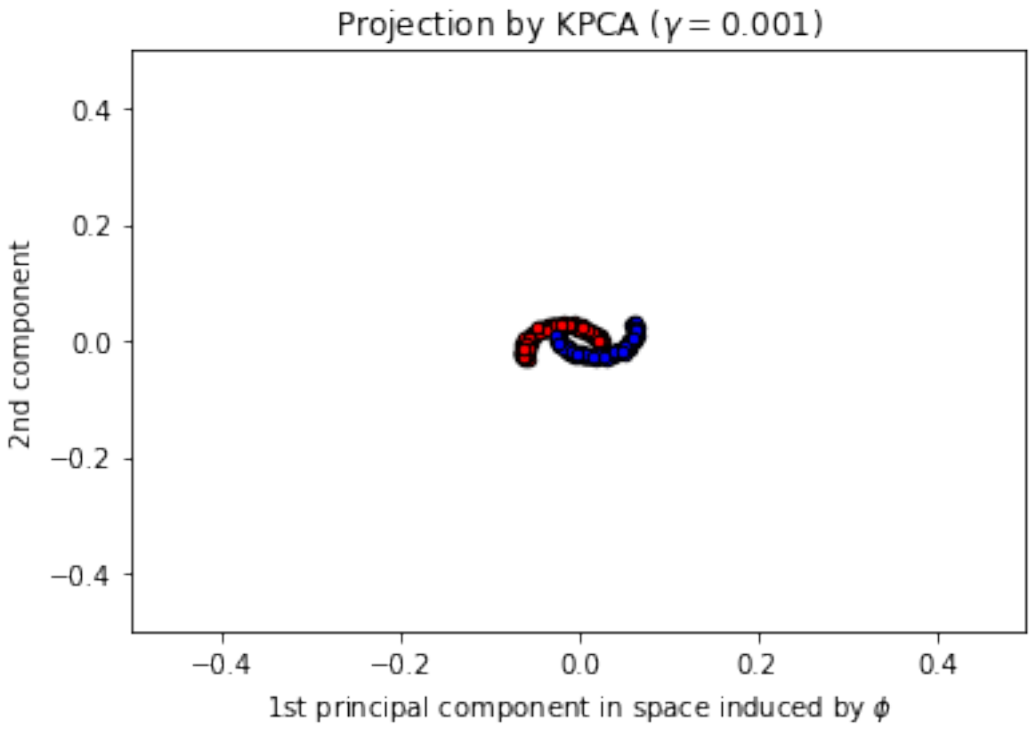
$$\hat{K} = \varepsilon \mathbf{1}, \quad \varepsilon << 1,$$

where $\mathbf{1}$ is a matrix with all ones. It is immediate to see that the normalized eigenvectors of such a matrix are the same for each value of ε, and as a result, the matrix multiplication consists of scaling via ε the desired projection:

$$\hat{K} \begin{pmatrix} e_1 & \ldots e_N \end{pmatrix} = \varepsilon \mathbf{1} \begin{pmatrix} e_1 & \ldots e_N \end{pmatrix},$$

but the basis of eigenvectors $\{e_1, \ldots, e_N\}$ is the same as the one obtained when ε is larger (but still within the bounds of our approximation). This behaviour can be seen in the animation above (or in the example below), where as $\gamma \to 0$ (i.e, $\varepsilon \to 0$), the projection structure is preserved, but reduced in scale.

[65]:
```
gamma = 1e-3
L = np.sqrt(0.5/gamma)

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


X_kpca, _, _ = kpca.kernel_pca(X, X, rbf_kernel)

plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title(r"Projection by KPCA ($\gamma=$" + f"{gamma})")
plt.xlabel(r"1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")
plt.xlim((-0.5, 0.5))
plt.ylim((-0.5, 0.5))
plt.show()
```



Next we consider the second case, where the kernel tends to $ 0 $ on any pair of non equal points. This does not apply when computing $\lim_{\gamma\to\infty} \mathcal{K}(x,x)$, as $\mathcal{K}(x,x) = 1$ is constant for any given value of γ. Considering this, the kernel matrix converges to

$$\lim_{\gamma\to\infty} \mathcal{K}(X,X) = I,$$

where I is the identity matrix of order N. In this case, centering the kernel amounts to a small perturbation of $1/N$, so it can be ignored for our purposes. The situation in this case is this: the application of the kernel to any set of points results in an identity matrix, which has all of its eigenvalues equal to $ 1 $, and its eigenvector can be any orthonormal basis in $\mathbb{R}^N$. There is no need for such base to be the usual basis of $\mathbb{R}^N$, but it is the case when using Numpy's `eigh` function.

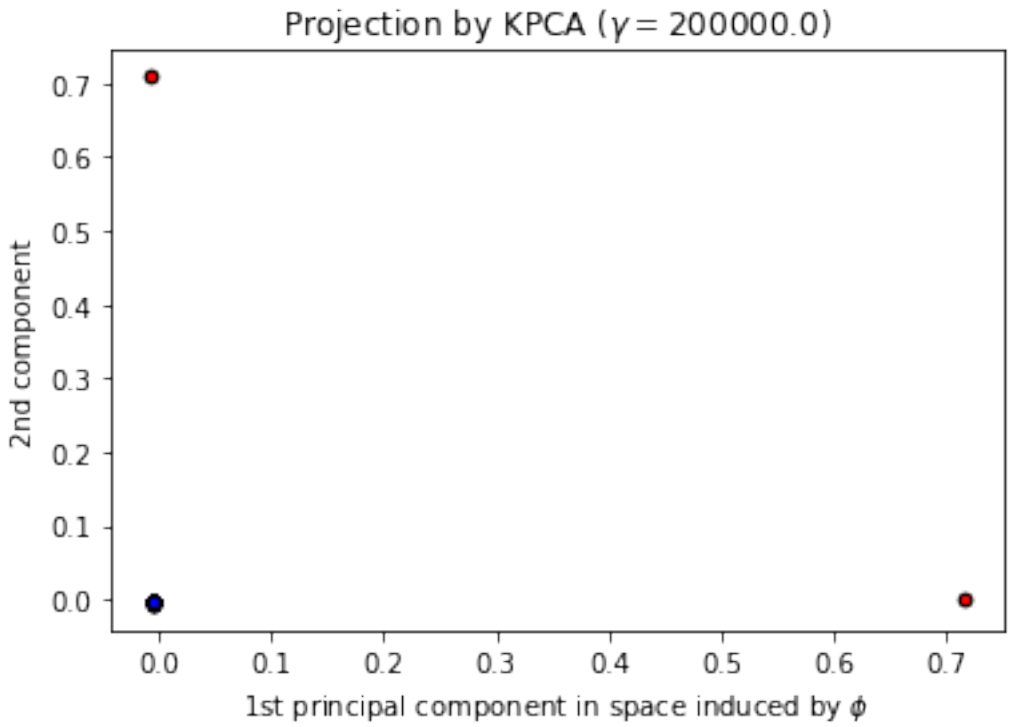
As a result, the kernel method performs the following matrix operation in the limit:

$$\mathcal{K}(X,X)\begin{pmatrix} e_1 & \dots & e_N \end{pmatrix} = II = I.$$

Given this, plotting a projection over the first two principal components leads to the degenerate case of projecting a point in $ (1,0) $ and another point in $ (0,1) $, while the rest are mapped to the origin. This phenomenon can approximately be seen if we set a large enough value of γ:

[57]:
```
gamma = 2e5
L = np.sqrt(0.5/gamma)

def rbf_kernel(X, X_prime):
```

```
    return kpca.rbf_kernel(X, X_prime, A, L)


X_kpca, _, _ = kpca.kernel_pca(X, X, rbf_kernel)

plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title(r"Projection by KPCA ($\gamma=$" + f"{gamma})")
plt.xlabel(r"1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")
plt.show()
```



However, there is an interesting behaviour of the `eigh` function when its argument is almost an identity matrix, but not exactly (that is, when γ is big but not that big). The returned eigenvalues are roughly equal to 1 and the eigenvectors are nearly an orthonormal basis of $\mathbb{R}^N$, but they are not close to being the usual base. As a result, the plot leads to the projection over $\mathbb{R}^2$ of a basis in $\mathbb{R}^N$, which corresponds to the "spiked" appearance of the projection, in which the lines that are formed seem to be the projection of perpendicular axes from a higher dimensional space.

[61]:
```
gamma = 2e2
L = np.sqrt(0.5/gamma)
```

```
def rbf_kernel(X, X_prime):
    return kpca.rbf_kernel(X, X_prime, A, L)


X_kpca, eigenvals_kpca, eigenvecs_kpca = \
    kpca.kernel_pca(X, X, rbf_kernel)

plt.scatter(X_kpca[reds, 0], X_kpca[reds, 1], c="red",
            s=20, edgecolor='k')
plt.scatter(X_kpca[blues, 0], X_kpca[blues, 1], c="blue",
            s=20, edgecolor='k')
plt.title(r"Projection by KPCA ($\gamma=$" + f"{gamma})")
plt.xlabel(r"1st principal component in space induced by $\phi$")
plt.ylabel("2nd component")
plt.show()
```