Lecture 14: Dimensionality Reduction – Kernel PCA

Textbook: Chapter 5

Dr. Huiping Cao

Kernel PCA

- In real applications, we may need to deal with nonlinear problems.
- For nonlinear problems, linear transformation techniques for dimensionality reduction (e.g., PCA and LDA) may not be the best choice.
- Kernelized version of PCA (or KPCA): transform data that is not linearly separable onto a new, higher-dimensional subspace that is suitable for linear classifiers. Or, it extends conventional principal component analysis (PCA) to a high dimensional feature space using the “kernel trick”.

Mapping function

- A nonlinear mapping function ϕ: $\mathbb{R}^d \rightarrow \mathbb{R}^k (k \gg d)$
- ϕ can be thought of as a function that creates **nonlinear combinations** of the features in the original *d*-dimensional dataset onto a larger *k*-dimensional feature space.
  E.g., **x** = $[x_1, x_2]$ can be transformed to **z** = $\left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]$.
- New feature space has *k* dimensions. It is denoted as $\mathcal{Z}$ space or Φ space.

How to use Kernel PCA

- Perform a **nonlinear mapping** via kernel PCA that transforms data to a higher dimensional space.
- Use **standard PCA** in this higher dimensional space to project the data back onto a lower-dimensional space where the samples can be separated by a **linear classifier**.
  - This lower-dimensional space (after running PCA) is different from the original X space.

Procedure

- PCA analysis: calculate covariance matrix

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{x}^{(i)}\right)\left(\mathbf{x}^{(i)}\right)^{T}$$

- In the $\mathcal{Z}$ space

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\phi\left(\mathbf{x}^{(i)}\right)\phi\left(\mathbf{x}^{(i)}\right)^{T}$$

- Then, we need to obtain the eigen vector of Σ. This process is computationally very expensive. (Detailed analysis see the extra slides provided.)
- This is where we use the **kernel trick**. Using the kernel trick, we can compute the similarity between two high-dimension feature vectors in the original feature space.

Kernel trick

- Using the kernel trick, we can avoid calculating the pairwise dot products of the samples **x** under φ. Instead, we use a kernel function κ

$$\kappa\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \phi\left(\mathbf{x}^{(i)}\right)^T \phi\left(\mathbf{x}^{(j)}\right)$$

- Different types of kernel functions
  - Polynomial kernel: $\kappa\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \left(\left(\mathbf{x}^{(i)}\right)^T \mathbf{x}^{(j)} + \theta\right)^p$
    - θ is a threshold. *p* is the power specified by the user.
  - Hyperbolic tangent (sigmoid) kernel: $\kappa\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \tanh\left(\eta \left(\mathbf{x}^{(i)}\right)^T \mathbf{x}^{(j)} + \theta\right)$
  - **Radial Basis Function (or Gaussian kernel)**: $\kappa\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$ or $\kappa\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right) = \exp\left(-\gamma \| \mathbf{x}^{(i)} - \mathbf{x}^{(j)} \|^2\right)$ where $\gamma = \frac{1}{2\sigma^2}$

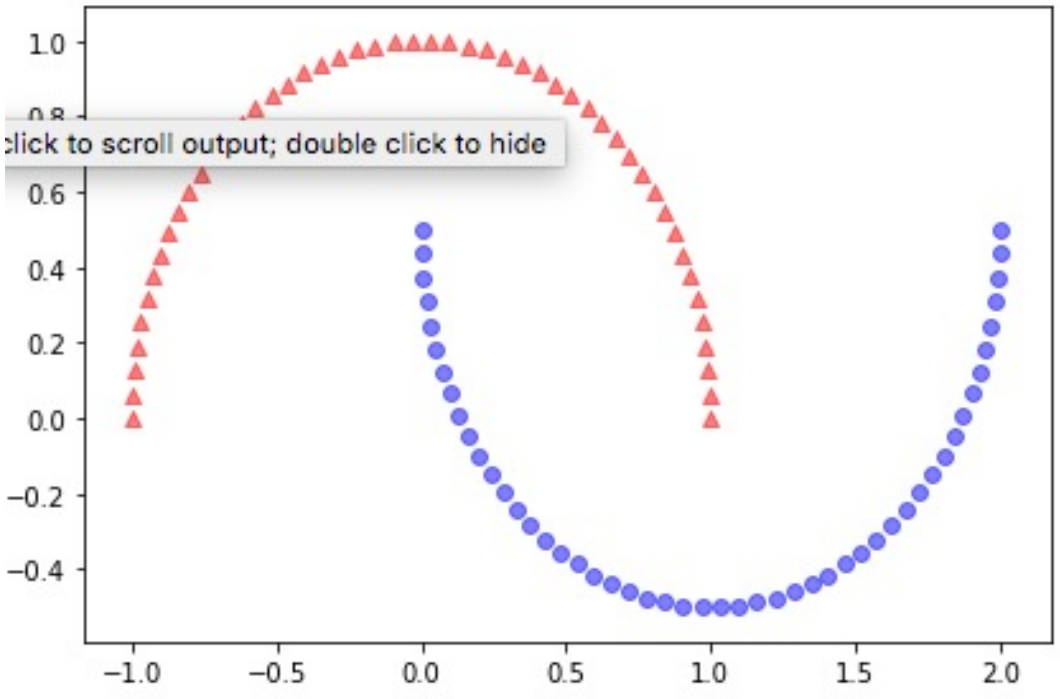
Scikit-learn kernel PCA

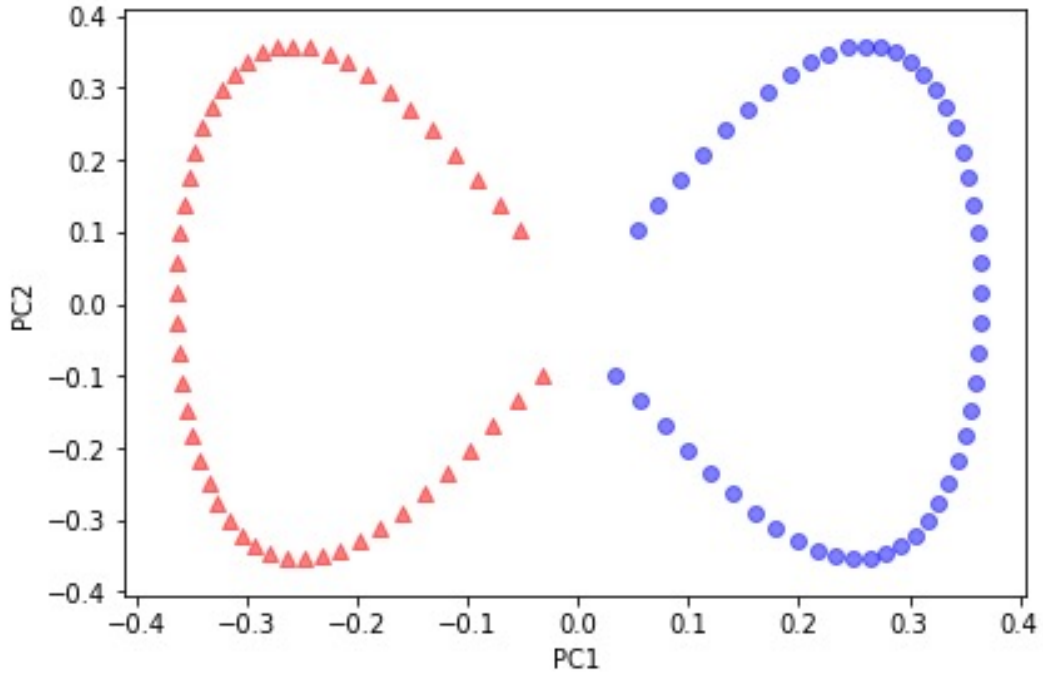
- **KernelPCA** function in **sklearn.decomposition** module
- Parameter “n_components”: Number of components. If None, all non-zero components are kept.
- Parameter ”kernel”.

```
from sklearn.decomposition import KernelPCA

scikit_kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_kpca = scikit_kpca.fit_transform(X)
```

Example 1

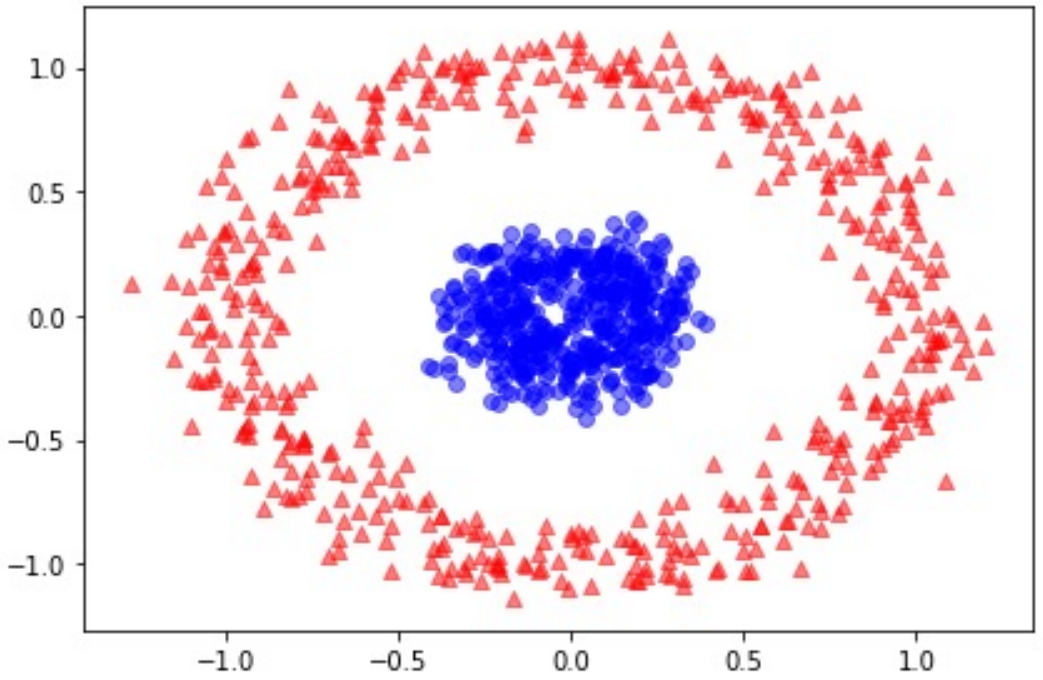
```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, random_state=123)

# plotting X with different colors for different class labels
# X_kpca  = apply KPCA operation on X with n_components=2, kernel='rbf', gamma=15
# plotting X_kpca with different colors for different class labels
```

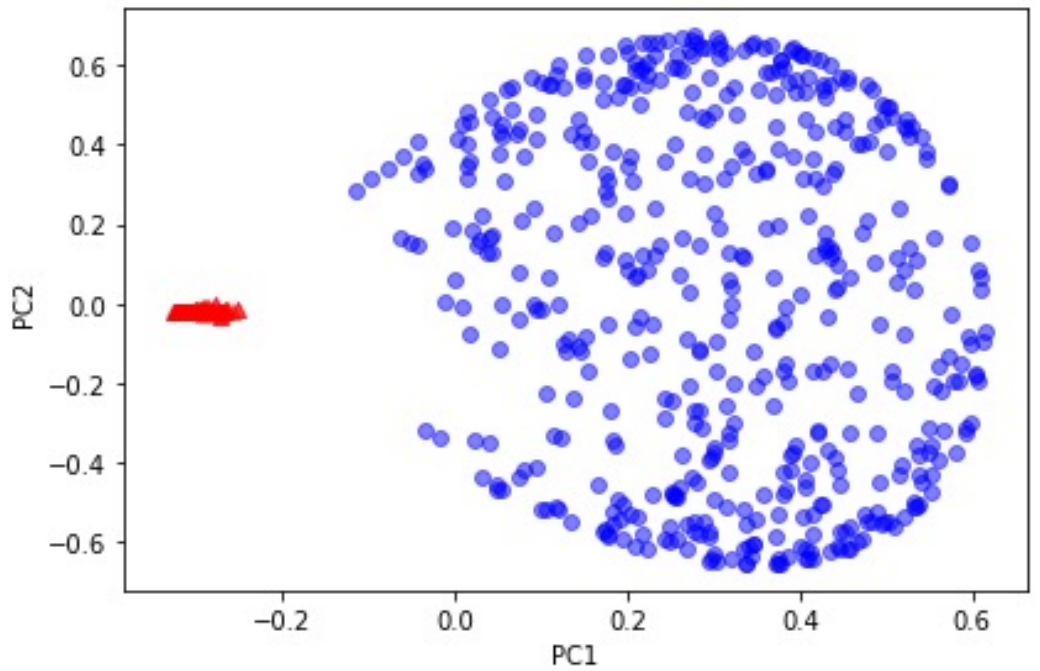




Example 2

```
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=1000, random_state=123, noise=0.1, factor=0.2)

# plotting X with different colors for different class labels
# X_skernpca  = apply KPCA operation on X with n_components=2, kernel='rbf', gamma=15
# plotting X_skernpca with different colors for different class labels
```





Use Kernel PCA

```
from sklearn.decomposition import KernelPCA

X, y = make_circles(n_samples=1000, random_state=123, noise=0.1, factor=0.2)
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3,stratify=y,random_state=0)

kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_train_kpca = kpca.fit_transform(X_train)
X_test_kpca = kpca.transform(X_test)

ppn1 = Perceptron(eta0=0.1, random_state=1)
ppn1.fit(X_train, y_train)
y_pred1 = ppn.predict(X_test)
print("accuracy (no kpca) = ", sum(y_test==y_pred1)/y_test.shape[0])

ppn2 = Perceptron(eta0=0.1, random_state=1)
ppn2.fit(X_train_kpca, y_train)
y_pred2 = ppn.predict(X_test_kpca)
print("accuracy (with kpca) ", sum(y_test==y_pred2)/y_test.shape[0])
```

```
accuracy (no kpca) =  0.6433333333333333
accuracy (with kpca)  1.0
```

Discussions

- Run linear classification algorithms after Kernel PCA process
- Transform the data **only once,** but can utilize it multiple times in different classification algorithms.