ODSC - Open Data Science Follow
Dec 10, 2019 · 6 min read · Listen



# Implementing a Kernel Principal Component Analysis in Python

*In this article, we discuss implementing a kernel Principal Component Analysis in Python, with a few examples.*

Many machine learning algorithms make assumptions about the linear separability of the input data. The perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) SVM to just name a few. However, if we are dealing with nonlinear problems, which we may encounter rather frequently in real-world applications, linear transformation techniques for dimensionality reduction, such as PCA and LDA, may not be the best choice.

*[Related Article: **Web Scraping News Articles in Python**]*

In this section, we will take a look at a kernelized version of PCA, or KPCA. Using KPCA, we will learn how to transform data that is not linearly separable onto a new, lower-dimensional subspace that is suitable for linear classifiers.

*This article is an excerpt from the book Python Machine Learning, Third Edition by Sebastian Raschka and Vahid Mirjalili. This book is a comprehensive guide to machine learning and deep learning with Python. This new third edition is updated for TensorFlow 2.0 and the latest additions to scikit-learn. In this article, we are going to implement an RBF KPCA in Python.*

Using some SciPy and NumPy helper functions, we will see that implementing a KPCA is actually really simple:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np
def rbf_kernel_pca(X, gamma, n_components):
    """
    RBF kernel PCA implementation.
    Parameters
    ------------
    X: {NumPy ndarray}, shape = [n_examples, n_features]
    gamma: float
        Tuning parameter of the RBF kernel
    n_components: int
        Number of principal components to return
    Returns
    ------------
    X_pc: {NumPy ndarray}, shape = [n_examples, k_features]
        Projected dataset
    """
    # Calculate pairwise squared Euclidean distances
    # in the MxN dimensional dataset.
    sq_dists = pdist(X, 'sqeuclidean')
    # Convert pairwise distances into a square matrix.
    mat_sq_dists = squareform(sq_dists)
    # Compute the symmetric kernel matrix.
    K = exp(-gamma * mat_sq_dists)
    # Center the kernel matrix.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)
    # Obtaining eigenpairs from the centered kernel matrix
```

```
    # scipy.linalg.eigh returns them in ascending order
    eigvals, eigvecs = eigh(K)
    eigvals, eigvecs = eigvals[::-1], eigvecs[:, ::-1]
    # Collect the top k eigenvectors (projected examples)
    X_pc = np.column_stack([eigvecs[:, i]
                            for i in range(n_components)])
    return X_pc
```

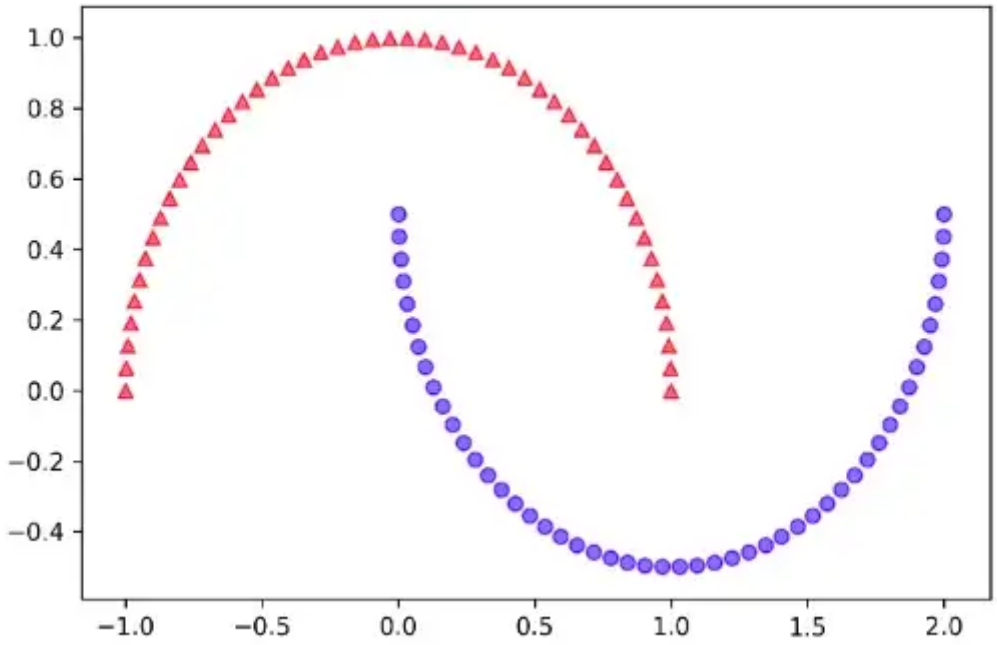
One downside of using an RBF KPCA for dimensionality reduction is that we have to specify the parameter a priori. Finding an appropriate value for requires experimentation and is best done using algorithms for parameter tuning, for example, performing a grid search.

## Example — separating half-moon shapes

Now, let us apply our rbf_kernel_pca on some nonlinear example datasets. We will start by creating a two-dimensional dataset of 100 example points representing two half-moon shapes:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_examples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...     color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...     color='blue', marker='o', alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

For the purposes of illustration, the half-moon of triangle symbols will represent one class, and the half-moon depicted by the circle symbols will represent the examples from another class:

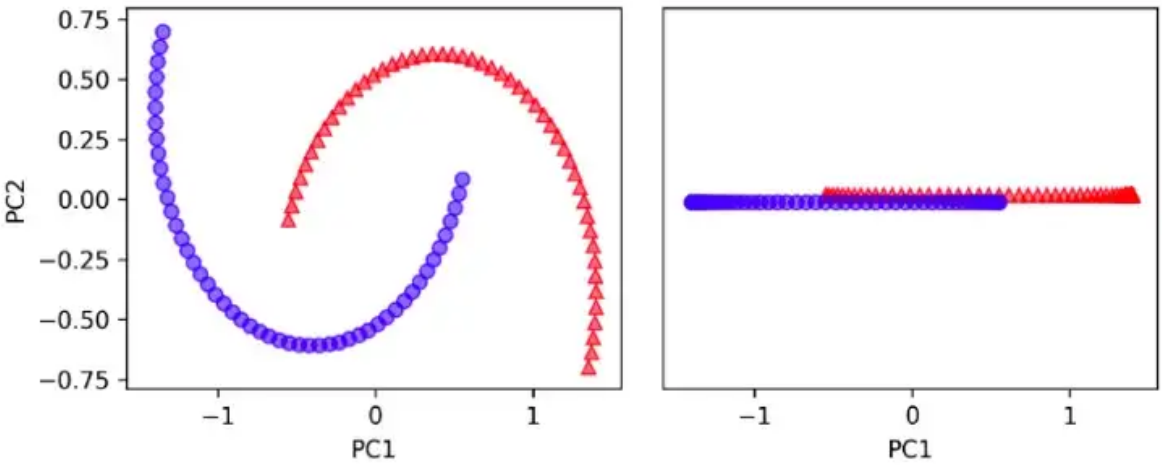



Clearly, these two half-moon shapes are not linearly separable, and our goal is to *unfold* the half-moons via KPCA so that the dataset can serve as a suitable input for a linear classifier. But first, let's see how the dataset looks if we project it onto the principal components via standard PCA:

```
>>> from sklearn.decomposition import PCA

>>> scikit_pca = PCA(n_components=2)

>>> X_spca = scikit_pca.fit_transform(X)

>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))

>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)

>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)

>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)

>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)

>>> ax[0].set_xlabel('PC1')

>>> ax[0].set_ylabel('PC2')

>>> ax[1].set_ylim([-1, 1])

>>> ax[1].set_yticks([])

>>> ax[1].set_xlabel('PC1')
```

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

Clearly, we can see in the resulting figure that a linear classifier would be unable to perform well on the dataset transformed via standard PCA:



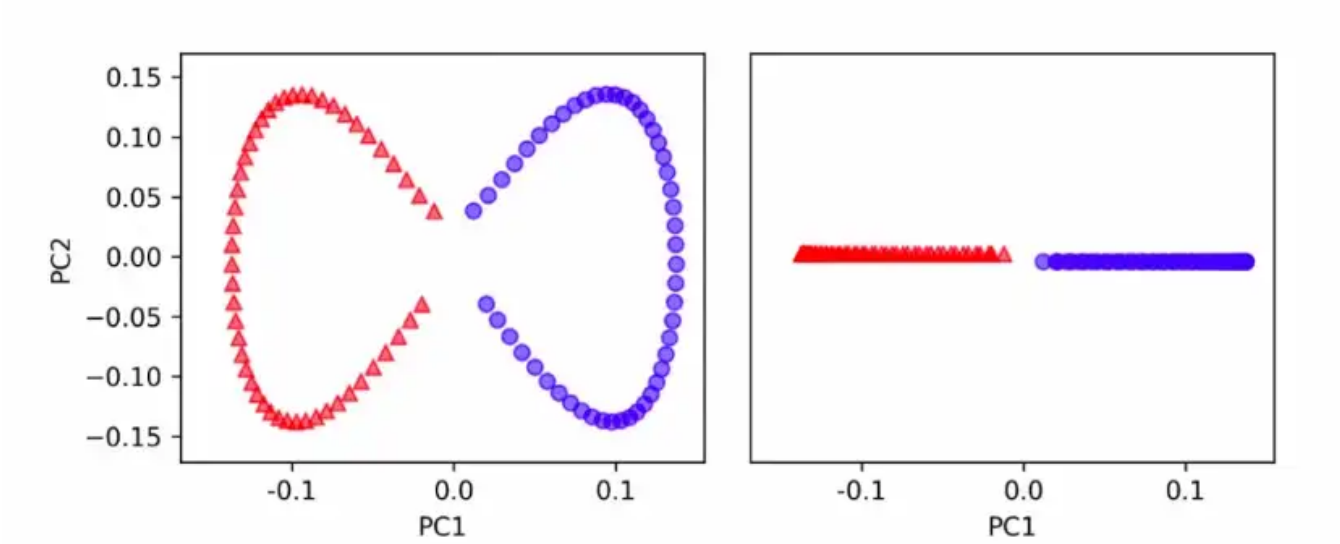

Note that when we plotted the first principal component only (right subplot), we shifted the triangular examples slightly upwards and the circular examples slightly downwards to better visualize the class overlap. As the left subplot shows, the original half-moon shapes are only slightly sheared and flipped across the vertical center — this transformation would not help a linear classifier in discriminating between circles and triangles. Similarly, the circles and triangles corresponding to the two half-moon shapes are not linearly separable if we project the dataset onto a one-dimensional feature axis, as shown in the right subplot.

Now, let's try out our kernel PCA function, rbf_kernel_pca, which we implemented in the previous subsection:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7, 3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
```

```
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)

>>> ax[0].set_xlabel('PC1')

>>> ax[0].set_ylabel('PC2')

>>> ax[1].set_ylim([-1, 1])

>>> ax[1].set_yticks([])

>>> ax[1].set_xlabel('PC1')

>>> plt.tight_layout()

>>> plt.show()
```

We can now see that the two classes (circles and triangles) are linearly well separated so that we have a suitable training dataset for linear classifiers:




Unfortunately, there is no universal value for the tuning parameter, , that works well for different datasets. Finding a value that is appropriate for a given problem requires experimentation. Here, I will use values for that I have found to produce good results.

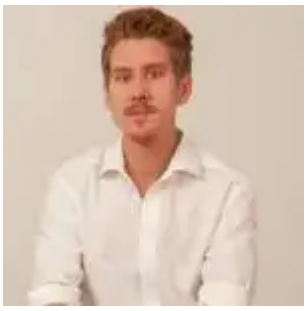
*[Related Article: **Local Regression in Python**]*

## Summary of Principal Component Analysis in Python

In this article, you learned about Principal Component Analysis in Python, KPCA. Using the kernel trick and a temporary projection into a higher-dimensional feature space, you were ultimately able to compress datasets consisting of nonlinear features onto a lower-dimensional subspace where the classes became linearly

separable. *Python Machine Learning, Third Edition* is a comprehensive guide to machine learning and deep learning with Python.

**About the Authors**



Sebastian Raschka has many years of experience with coding in Python, and he has given several seminars on the practical applications of data science, machine learning, and deep learning, including a machine learning tutorial at SciPy — the leading conference for scientific computing in Python. He is currently an Assistant Professor of Statistics at UW-Madison focusing on machine learning and deep learning research.

His work and contributions have recently been recognized by the departmental outstanding graduate student award 2016–2017, as well as the ACM Computing Reviews' Best of 2016 award. In his free time, Sebastian loves to contribute to open source projects, and the methods that he has implemented are now successfully used in machine learning competitions, such as Kaggle.

Vahid Mirjalili obtained his PhD in mechanical engineering working on novel methods for large-scale, computational simulations of molecular structures. Currently, he is focusing his research efforts on applications of machine learning in

various computer vision projects at the Department of Computer Science and Engineering at Michigan State University.

While Vahid's broad research interests focus on deep learning and computer vision applications, he is especially interested in leveraging deep learning techniques to extend privacy in biometric data such as face images so that information is not revealed beyond what users intend to reveal. Furthermore, he also collaborates with a team of engineers working on self-driving cars, where he designs neural network models for the fusion of multispectral images for pedestrian detection.

*Original post here.*

*Read more data science articles on OpenDataScience.com, including tutorials and guides from beginner to advanced levels! Subscribe to our weekly newsletter here and receive the latest news every Thursday.*

Machine Learning Python Data Science Artificial Intelligence

Component Analysis