1 Kernel PCA for nonlinear dimensionality reduction

Most of the techniques in statistics are linear by nature, so in order to capture nonlinearity, we might need to apply some transformation. PCA is, of course, a linear transformation. In this recipe, we'll look at applying nonlinear transformations, and then apply PCA for dimensionality reduction.

1.0.1 Getting ready

- Life would be so easy if data was always linearly separable, but unfortunately it's not. Kernel PCA can help to circumvent this issue. Data is first run through the kernel function that projects the data onto a different space; then PCA is performed.
- To familiarize yourself with the kernel functions, it will be a good exercise to think of how to generate data that is separable by the kernel functions available in the kernel PCA.
- Here, we'll do that with the cosine kernel. This recipe will have a bit more theory than the previous recipes.

How to do it

The cosine kernel works by comparing the angle between two samples represented in the feature space. It is useful when the magnitude of the vector perturbs the typical distance measure used to compare samples.

As a reminder, the cosine between two vectors is given by the following:

In Euclidean space, a Euclidean vector is a geometrical object that possesses both a magnitude and a direction. A vector can be pictured as an arrow. Its magnitude is its length, and its direction is the direction that the arrow points. The magnitude of a vector A is denoted by $\|\mathbf{A}\|$. The dot product of two Euclidean vectors A and

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

Figure 1

B is defined by[2]

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$$

, where θ is the angle between A and B. This means that the cosine between A and B is the dot product of the two vectors normalized by the product of the individual norms. The magnitude of vectors A and B have no influence on this calculation. So, let's generate some data and see how useful it is. First, we'll imagine there are two different underlying processes; we'll call them A and B:

```
>>> import numpy as np
>>> A1_mean = [1, 1]
>>> A1_cov = [[2, .99], [1, 1]]
>>> A1 = np.random.multivariate_normal(A1_mean, A1_cov, 50)
>>> A2_mean = [5, 5]
>>> A2_cov = [[2, .99], [1, 1]]
>>> A2 = np.random.multivariate_normal(A2_mean, A2_cov, 50)
>>> A = np.vstack((A1, A2))
>>> B_mean = [5, 0]
>>> B_cov = [[.5, -1], [-.9, .5]]
>>> B = np.random.multivariate_normal(B_mean, B_cov, 100)
```

Once plotted, it will look like the following: By visual inspection, it seems that the two classes are from different processes, but separating them in one slice might be difficult. So, we'll use the kernel PCA with the cosine kernel discussed earlier:

```
>>> kpca = decomposition.KernelPCA(kernel='cosine', n_components=1)
>>> AB = np.vstack((A, B))
>>> AB_transformed = kpca.fit_transform(AB)
```

Visualized in one dimension after the kernel PCA, the dataset looks like the following: Contrast this with PCA without a kernel: Clearly, the kernel PCA does a much better job. How it works... There are several different kernels available as well as the cosine kernel. You can even write your own kernel function. The available kernels are:

- poly (polynomial)
- rbf (radial basis function)
- sigmoid
- cosine
- precomputed

There are also options contingent of the kernel choice. For example, the degree argument will specify the degree for the poly, rbf, and sigmoid kernels; also, gamma will affect the rbf or poly kernels.

The recipe on SVM will cover the rbf kernel function in more detail. A word of caution: kernel methods are great to create separability, but they can also cause overfitting if used without care.

Using truncated SVD to reduce dimensionality

Truncated Singular Value Decomposition (SVD) is a matrix factorization technique that factors a matrix M into the three matrices U, $\sum$, and V. This is very similar to PCA, excepting that the factorization for SVD is done on the data matrix, whereas for PCA, the factorization is done on the covariance matrix. Typically, SVD is used under the hood to find the principle components of a matrix.

1.0.2 Getting ready

Truncated SVD is different from regular SVDs in that it produces a factorization where the number of columns is equal to the specified truncation. For example, given an n x n matrix, SVD will produce matrices with n columns, whereas truncated SVD will produce matrices with the specified number of columns. This is how the dimensionality is reduced. Here, we'll again use the iris dataset so that you can compare this outcome against the PCA outcome:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> iris_data = iris.data
>>> iris_target = iris.target
```

How to do it... This object follows the same form as the other objects we've used. First, we'll import the required object, then we'll fit the model and examine the results:

```
>>> from sklearn.decomposition import TruncatedSVD
>>> svd = TruncatedSVD(2)
>>> iris_transformed = svd.fit_transform(iris_data)
```

```
>>> iris_data[:5]
array([[ 5.1, 3.5, 1.4, 0.2],
       [ 4.9, 3. , 1.4, 0.2],
       [ 4.7, 3.2, 1.3, 0.2],
       [ 4.6, 3.1, 1.5, 0.2],
       [ 5. , 3.6, 1.4, 0.2]])
>>> iris_transformed[:5]
array([[ 5.91220352, -2.30344211],
       [ 5.57207573, -1.97383104],
       [ 5.4464847 , -2.09653267],
       [ 5.43601924, -1.87168085],
       [ 5.87506555, -2.32934799]])
```

The output will look like the following: How it works... Now that we've walked through how TruncatedSVD is performed in scikit-learn, let's look at how we can use only scipy, and learn a bit in the process. First, we need to use linalg of scipy to perform SVD:

```
>>> from scipy.linalg import svd
>>> D = np.array([[1, 2], [1, 3], [1, 4]])
>>> D
array([[1, 2],
[1, 3],
[1, 4]])
>>> U, S, V = svd(D, full_matrices=False)
>>> U.shape, S.shape, V.shape
((3, 2), (2,), (2, 2))
```

We can reconstruct the original matrix D to confirm U, S, and V as a decomposition:

```
>>> np.dot(U.dot(np.diag(S)), V)
array([[1, 2],
[1, 3],
[1, 4]])
```

The matrix that is actually returned by TruncatedSVD is the dot product of the U and S matrices. If we want to simulate the truncation, we will drop the smallest singular values and the corresponding column vectors of U. So, if we want a single component here, we do the following:

```
>>> new_S = S[0]
>>> new_U = U[:, 0]
>>> new_U.dot(new_S)
array([-2.20719466, -3.16170819, -4.11622173])
```

In general, if we want to truncate to some dimensionality, for example, t, we drop N-t singular values. There's more... TruncatedSVD has a few miscellaneous things that are worth noting with respect to the method. Sign flipping There's a "gotcha" with truncated SVDs. Depending on the state of the random number generator, successive fittings of TruncatedSVD can flip the signs of the output. In order to avoid this, it's advisable to fit TruncatedSVD once, and then use transforms from then on. Another good reason for Pipelines! To carry this out, do the following:

```
>>> tsvd = TruncatedSVD(2)
>>> tsvd.fit(iris_data)
```

```
>>> tsvd.transform(iris_data)
```

Sparse matrices

One advantage of TruncatedSVD over PCA is that TruncatedSVD can operate on sparse matrices while PCA cannot. This is due to the fact that the covariance matrix must be computed for PCA, which requires operating on the entire matrix.