CS 331

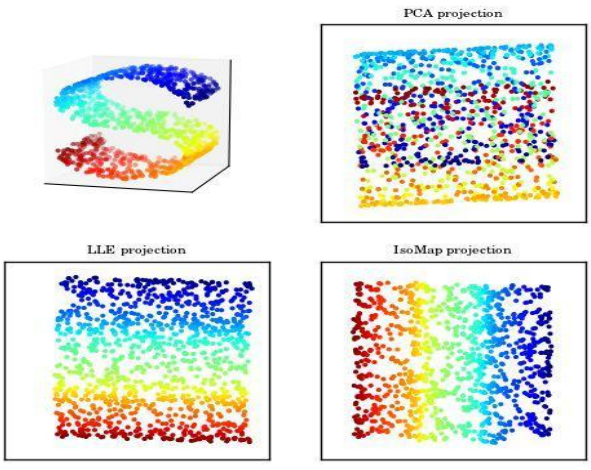
KERNEL PRINCIPAL COMPONENT ANALYSIS

- ArAkNiKhTuMa

Kernel Principal Component Analysis (KPCA) is a non-linear dimensionality reduction technique. It is an extension of Principal Component Analysis (PCA) - which is a linear dimensionality reduction technique - to a high dimensionality dimensional feature space using the “kernel trick”. The algorithm can efficiently extract up to n (number of samples) nonlinear principal components without expensive computations.

Why non-linear dimensionality reduction?

Sometimes the structure of the data is nonlinear, and the principal components will not give us the optimal dimensionality reduction, so we use non-linear methods like KPCA. For example, in the figure below, PCA will look for the surface that will cross the S-shaped data with minimal reconstruction error, but no matter what surface we choose, there will be some loss in information. This is because some points from different colors would be mapped to one point. Unlike PCA, non-linear methods will give us more optimal projections as shown in the last two figures.



Intuition behind KPCA

The idea of KPCA relies on the intuition that many datasets, which are not linearly separable in their space, can be made linearly separable by projecting them into a higher dimensional space. The added dimensions are just simple arithmetic operations performed on the original data dimensions.

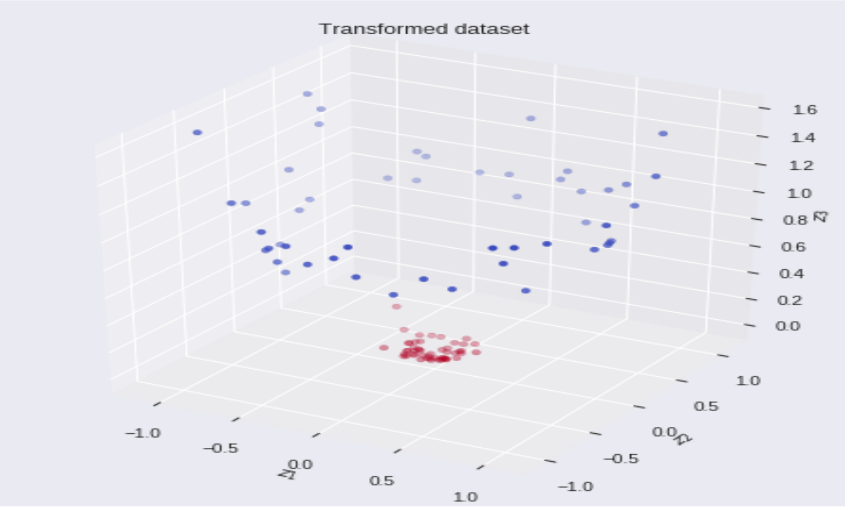
So we project our dataset into a higher dimensional feature space, and because they become linearly separable, then we can apply PCA to this new dataset. Performing this linear dimensionality reduction in that space will be equivalent to a non-linear dimensionality reduction in the original space.

For example, consider this dataset where the red and blue points are not linearly separable in two-dimensional space.

We define the following linear mapping:

$$T(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

where this mapping function will project the data from a lower-dimensional (2D) to a higher dimensional (3D) space as shown below.



.

We can see from the above figure that the dataset becomes linearly separable. The optimal surface that PCA will find will be in the middle between blue and red points. And if we inverse this transform then this surface will correspond to a nonlinear curve in the original 2D space. And by projecting on that curve, we will find the optimal dimensionality reduction.

But doing PCA in high dimensional space will need a lot of computations, so in order to solve this problem, we use kernel methods.

Kernel 'Trick'

As mentioned before, doing PCA in the transformed dataset will need a lot of computations. But with the help of kernel methods, we can perform these computations in the original state space. This kernel method or 'trick' is defined as follows:

$$K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

Given any algorithm that can be expressed solely in terms of dot products, this trick allows us to construct different nonlinear versions of it. The above can be achieved with the help of kernel functions that give us a way of computing the dot product, between two vectors - in high dimensional space - in our original space. Some examples of kernel functions are polynomial, Radial Basis Function (RBF), and Gaussian kernels.

Mathematical Proof

In the images below, we can see the mathematical derivation behind the KPCA algorithm.

Page No.

Converting data from $\mathbb{R}^d$ to $\mathbb{R}^K$ where $K<d$.

Take $D = \{x^{(1)}, x^{(2)} \cdots x^{(n)}\}$ Set of mean-centered points

Suppose $\{u_1, \cdots u_d\}$ is orthonormal basis to $\mathbb{R}^d$.

We will choose K-dimension subspace $\{u_1 \cdots u_K\}$

Given, $x = x^T u_1 u_1 + \cdots x^T u_d u_d$

Transformed, $\tilde{x} = x^T u_1 u_1 + \cdots x^T u_K u_K$

Problem: $\min_{u_1 \cdots u_d} \sum_{i=1}^{n} \| x^{(i)} - \tilde{x}^{(i)} \|^2$

s.t. $u_i^T u_i = 1 \quad \forall\ i = 1 \cdots d$

$u_i^T u_j = 0 \quad \forall\ i \neq j$

Objective can be written as: $\sum_{i=1}^{n} \| \sum_{j=1}^{d} (x^{(i)T} u_j u_j) - \sum_{j=1}^{K} x^{(i)T} u_j u_j \|_2^2$

$$= \sum_{i=1}^{n} \sum_{j=K+1}^{d} (x_i^T u_j)^T (x_i^T u_j)$$

$$= \sum_{i=1}^{n} \sum_{j=K+1}^{d} u_j^T x_i x_i^T u_j$$

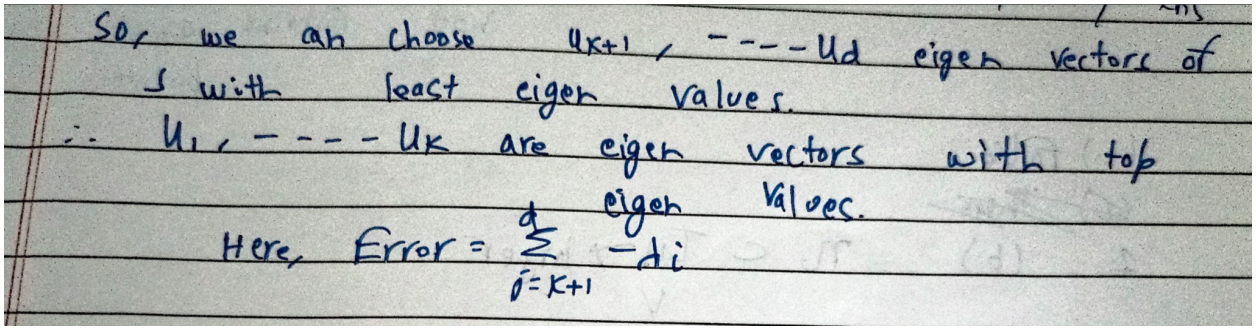
$$= \sum_{j=K+1}^{d} u_j^T \underbrace{\sum_{i=1}^{n} x_i x_i^T}_{\to S} u_j$$

$$= \sum_{j=K+1}^{d} u_j^T S u_j$$

Here, lagrangian, $L = \sum_{j=K+1}^{d} u_j^T S u_j + \sum_{j=1}^{d} \lambda_j (u_j^T u_j - 1)$

$\nabla L_{u_j} = 2 S u_j + 2 \lambda_j u_j \quad \forall\ j = K+1 \cdots d$

$\nabla L_{u_j} = 0 \Rightarrow S u_j = -\lambda_j u_j$



Kernel PCA: Pseudocode

The Kernel PCA algorithm can be summarized as follows:

1) Initially, the kernel functions $k(x_i, x_j)$ are chosen and let T be any transformation to a higher dimension.
2) Similar to PCA, we compute the covariance matrix of our data. But here, the kernel function is used to calculate this matrix. Thus, the finally computed kernel matrix is the matrix that results from applying the kernel function to all pairs of data.

$$K = T(X) * T(X)^T$$

3) We then center our kernel matrix (this step is equivalent to subtracting the mean of the transformed data and dividing by standard deviations)
4) Then, the eigenvectors and eigenvalues of this matrix are computed.
5) Eigenvectors are then sorted based on their corresponding eigenvalues in decreasing order.
6) We then choose the number of dimensions that we want our reduced dataset to be, calling it m. Then we will choose the first m eigenvectors and concatenate them in one matrix.
7) Finally, we calculate the product of that matrix with our data. This result will be our new reduced dataset.

The pseudocode for the algorithm is as follows:

Algorithm 1 Kernel PCA Algorithm

procedure K - PCA(X)

Given Input: $X_{N\times M} \leftarrow [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_M]$

Centralize : $X_{centered} \leftarrow X_{N\times M}$

Kernel Matrix : $K_{M\times M} : k_{ij} \leftarrow k(\mathbf{x}_i, \mathbf{x}_j)$

Centralization in F space :

$K : K \leftarrow K - I_M K/M - K I_M/M + I_M K I_M/M^2$

Extracting eigenvectors : $M\lambda\boldsymbol{\alpha} = K\boldsymbol{\alpha}$

Normalization :

$\boldsymbol{\alpha} \leftarrow \frac{\boldsymbol{\alpha}}{mod(\boldsymbol{\alpha})\sqrt{M\lambda}}$

loop: $i \leftarrow 1 : p$

$$P_i(x) = \sum_{i=1}^{M} \alpha_{ij}\kappa(\mathbf{x}_i, \mathbf{x})$$

goto *top.*

Steps for the Algorithm

```
class kernel:
    def __init__(self, gamma = 1, sigma = 1, c = 0):
        self.gamma = gamma
        self.sigma = sigma
        self.c = c

    def linear(self, x, y):
        return np.matmul(x.T, y) + self.c

    def rbf(self, x, y):
        return np.exp(- self.gamma * (np.linalg.norm(x-y)**2))

    def exp(self, x, y):
        return np.exp(- (1/ (2*self.sigma**2)) * np.linalg.norm(x-y))
```

In the code above, we declared a class named ‘kernel’ to define different kernel functions such as linear, radial basis, and exponential.

```
class KPCA:
    def __init__(self, X, kernel, d):
        self.X = X
        self.kernel = kernel
        self.d = d

    def is_pos_semidef(self, x):
        return np.all(x >= 0)

    def kernel_matrix(self):
        K = []
        r, c = self.X.shape
        for fil in range(c):
            k_aux = []
            for col in range(c):
                k_aux.append(self.kernel(self.X[:, fil], self.X[:, col]))
            K.append(k_aux)
        K = np.array(K)
        # Centering K
        ones = np.ones(K.shape)/c
        K = StandardScaler().fit_transform(K)
        return K

    def decomposition(self):
        self.K = self.kernel_matrix()
        temp=np.zeros((self.K.shape[1],self.K.shape[1]))
        for i in range(len(self.K)):
            x=self.K[i].reshape((self.K.shape[1],1))
            temp+=np.dot(x,x.T)
        self.K=temp
        eigval, eigvec = np.linalg.eig(self.K)
        if not self.is_pos_semidef(eigval):
            warnings.warn("The matrix K is not positive semidefinite")
        # Normalize eigenvectors and compute singular values of K
        eigen = [(eigval[i], eigvec[:,i]/np.linalg.norm(eigvec[:,i]) ) for i in range(len(eigval))]
        eigen.sort(key=lambda x: x[0], reverse=True)
        return eigen

    def project(self):
        self.eigen = self.decomposition()
        eigen_dim = self.eigen[:self.d]
        self.sigma = np.diag([i[0] for i in eigen_dim])
        self.v = np.array([list(j[1]) for j in eigen_dim]).T
        self.sigma = np.real_if_close(self.sigma, tol=1)
        self.v = np.real_if_close(self.v, tol=1)
        self.scores = np.matmul(self.sigma, self.v.T)
        return self.scores
```

As shown in the code snippet above, we have declared a class named 'KPCA' consisting of all the necessary functions to carry out the Kernel PCA implementation. The input for the class are input data (dimension: dxn), kernel function choice, and the number of principal components/features to be chosen.

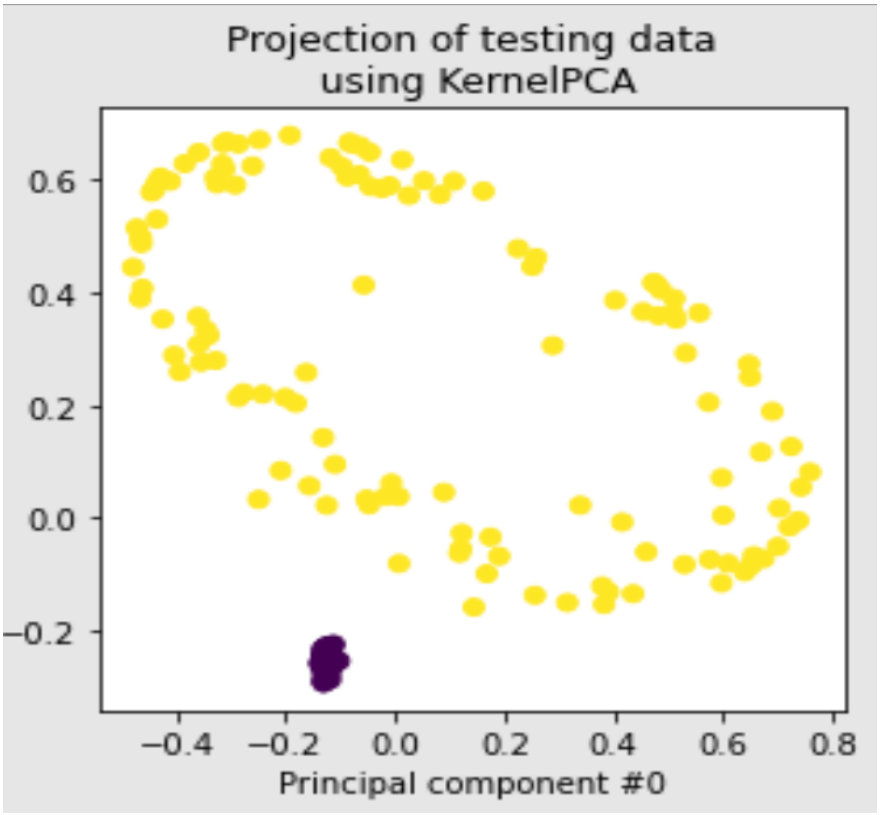
The function ***'kernel_matrix'*** results in a final centered matrix (dimension: nxn) obtained from implementing the chosen kernel function on our input data.

The function ***'decomposition'*** first computes the normalized eigenvalues and eigenvectors of the kernel matrix and then sort the eigenvectors in decreasing order based on their corresponding eigenvalues.

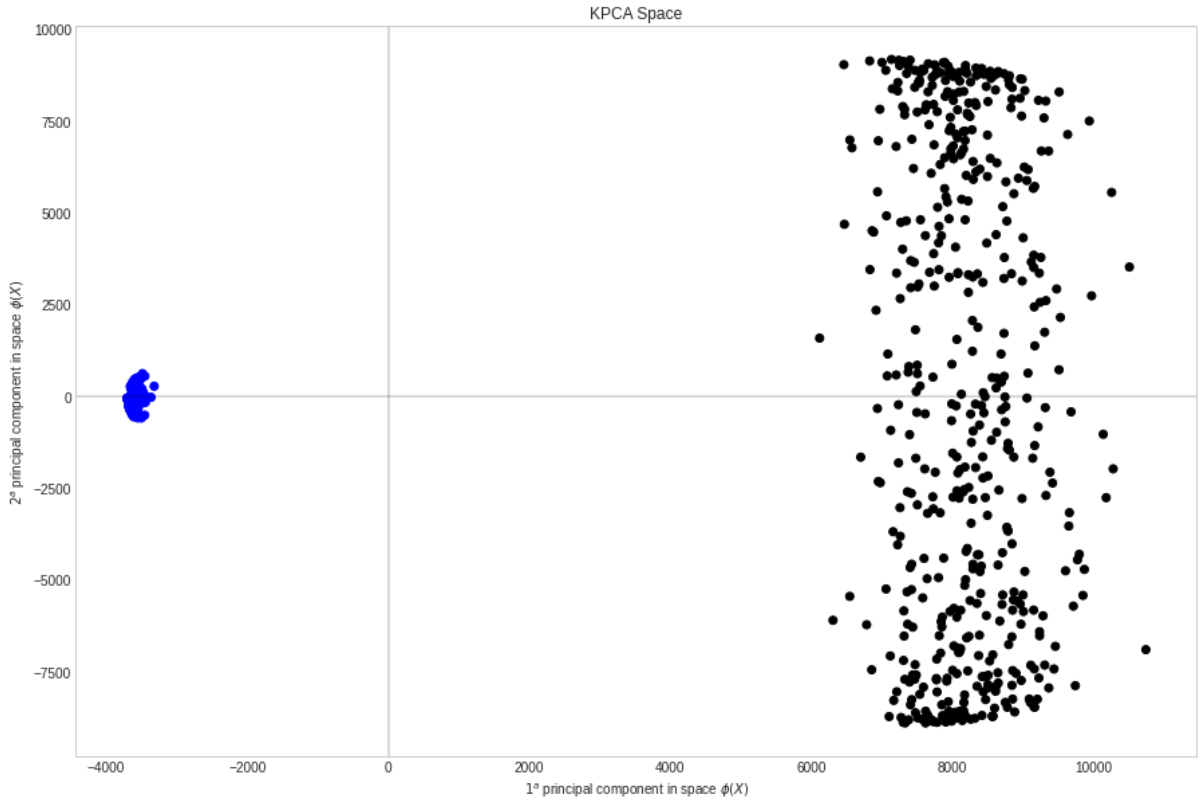
Let's call our reduced dataset m and choose the number of dimensions we want it to have. Then we'll concatenate our first m eigenvectors into a single matrix. Finally, multiply your data by the product of that matrix. Your new, smaller dataset will be the result. This will be done in the function **"project"**.

The function ***'project'*** consists of concatenating the first m eigenvectors into a single matrix and then multiplying the reduced dataset with this matrix. This results in a smaller dataset.

Comparison with the library implementation



Implementation using direct sklearn libraries



Implementation from scratch

Advantages and Disadvantages of KPCA over PCA

Advantages of KPCA over PCA	Disadvantages of KPCA over PCA
1) M features can be extracted (where M is the number of observations)	1) The projection on higher dimensions does not necessarily have a pre-image
2) It effectively analyses non-linear variance	2) It becomes tough to predict the contour lines intuitively
3) The classifier has the opportunity to train itself better as the extracted feature now depends on the number of observations	3) Clustering (or data separation) does not necessarily work better as extracted feature are abstract in nature

Conclusion

From the above plots that depict the implementation of Kernel PCA using imported libraries and from scratch, we can observe that there is a similar pattern in the way the data points are separated. Hence, assuring the correctness of our scratch implementation. When it comes to the practical application, one should study the dataset very well, in order to select what algorithm to use to reduce your dimensions. One should also consider the fact that KPCA takes more time than PCA.

Bibliography

1) https://scikit-learn.org/stable/auto_examples/decomposition/plot_kernel_pca.html
2) https://ml-explained.com/blog/kernel-pca-explained
3) https://iq.opengenus.org/kernal-principal-component-analysis/