PATTERN RECOGNITION

KERNEL PCA

Author

Sina Heydari

Table Of Contents

1 Code Documentation

1.1 sortEigenvaluesAndVectors

This function computes the eigenvalues and eigenvectors of a given square matrix A. It then sorts these eigenvalues in descending order and rearranges the corresponding eigenvectors accordingly.

Inputs:

***A*:** A square matrix

Outputs:

eigenVectorsSorted: Matrix of eigenvectors sorted according to descending eigenvalues.

eigenValuesSorted: Diagonal matrix of eigenvalues sorted in descending order.

The function first calculates the eigenvalues and eigenvectors using MATLAB's eig function. It then sorts the eigenvalues in descending order and rearranges the eigenvectors to match this order.

```
function [eigenVectorsSorted, eigenValuesSorted] =
    sortEigenvaluesAndVectors(A)
    % This function computes the eigenvalues and
        eigenvectors of matrix A
    % and sorts them in descending order of the
        eigenvalues.

    [eigenVectors, eigenValues] = eig(A);
    [~, index] = sort(diag(eigenValues), 'descend');
    eigenValuesSorted = eigenValues(index, index);
    eigenVectorsSorted = eigenVectors(:, index);
end
```

1.2 findPrincipalComponents

This function determines the number of principal components required to retain a specified percentage of the total variance in the data.

Inputs:

***D*:** A diagonal matrix of eigenvalues. percent: The percentage of total variance to retain.

Outputs:

***k*:** The number of principal components required to retain the specified variance.

The function calculates the total variance and iteratively sums the eigenvalues until the cumulative variance reaches the specified percentage.

```
function k = findPrincipalComponents(D, percent)
    % This function determines the number of
       principal components required
    % to retain the given percentage of total
       variance.

    totalVariance = sum(D, 'all') * (percent / 100);
    cumulativeVariance = 0;
    k = 0;

    while cumulativeVariance < totalVariance
        k = k + 1;
        cumulativeVariance = sum(D(:, 1:k), 'all');
    end
end
```

1.3 generateData

This function generates a 3D dataset in the shape of a sphere and assigns labels to the data points using a Gaussian function.

Outputs:

- **featureX:** X-coordinates of the data points.
- **featureY:** Y-coordinates of the data points.
- **featureZ:** Z-coordinates of the data points.
- **samples:** 2D array of X and Y coordinates.
- **labels:** Labels assigned to data points based on their distance from the center.

The function uses MATLAB's sphere function to generate the data and reshapes it into column vectors. Labels are assigned based on a Gaussian kernel function centered at the origin.

```
function [featureX, featureY, featureZ, samples,
    labels] = generateData()
    % This function generates a 3D spherical
        dataset and assigns labels
    % using a Gaussian function.

    % Generate data using sphere
    [featureX, featureY, featureZ] = sphere;
    featureX = reshape(featureX, numel(featureX),
        1);
    featureY = reshape(featureY, numel(featureY),
        1);
    featureZ = reshape(featureZ, numel(featureZ),
        1);
    samples = [featureX, featureY];

    % Assign labels using Gaussian function
    labels = zeros(1, size(featureY, 1));
    % Select nearest to center data
    labels(gKernel(samples, [0; 0]', 0.6) > 0.6) =
        1;
end
```

1.4 gKernel

This function calculates the Gaussian kernel value between two vectors xx and yy using a specified sigma.

Inputs:

- **x:** A vector.
- **y:** Another vector of the same size as x.
- **sigma:** The bandwidth of the Gaussian kernel.

Outputs:

- **kernelValue:** The Gaussian kernel value.

The function computes the squared Euclidean distance between x and y, applies the Gaussian function, and returns the result.

```
function kernelValue = gKernel(x, y, sigma)
    % This function calculates the Gaussian kernel
       value between x and y
    % using the given sigma.

    distance = sum((x - y).^2, 2);
    coefficient = -1 / (2 * (sigma^2));
    kernelValue = exp(distance * coefficient);
end
```

1.5 KernelPCA

This function performs Kernel Principal Component Analysis (Kernel PCA) on the input data to project it into a new feature space.

Inputs:

- **data:** The input data matrix where each column is a data point.
- **percent:** The percentage of total variance to retain in the transformed data.

Outputs:

- **transformedData:** The data projected into the new feature space.
- **selectedEigenVectors:** The eigenvectors corresponding to the retained variance.
- **selectedEigenValues:** The eigenvalues corresponding to the retained variance.

The function constructs a polynomial kernel matrix, centers it, performs eigen decomposition, and selects the principal components based on the specified variance retention percentage. It then projects the data into the new feature space using the selected eigenvectors.

```
function [transformedData, selectedEigenVectors,
   selectedEigenValues] = KernelPCA(data, percent,
   kernelType, kernelParams)
    % Construct the kernel matrix based on the
       specified kernel type
    switch kernelType
        case 'polynomial'
            c = kernelParams(1);
            d = kernelParams(2);
            kernelMatrix = polynomialKernel(data,
               data, c, d);
        case 'gaussian'
            sigma = kernelParams(1);
            kernelMatrix =
               computeGaussianKernel(data, sigma);
        otherwise
            error('Unsupported kernel type');
```

```
    end

    % Kernel centering
    numSamples = size(data, 2);
    unitMatrix = (1 / numSamples) *
       ones(numSamples, numSamples);
    centeredKernelMatrix = kernelMatrix -
       kernelMatrix * unitMatrix - unitMatrix *
       kernelMatrix + unitMatrix * kernelMatrix *
       unitMatrix;

    % Eigen decomposition
    [eigenVectors, eigenValues] =
       sortEigenvaluesAndVectors(centeredKernelMatrix);
    numComponents =
       findPrincipalComponents(eigenValues,
       percent);
    selectedEigenVectors = eigenVectors(:,
       1:numComponents);
    selectedEigenValues = eigenValues(:,
       1:numComponents);

    % Transform data
    transformedData = selectedEigenVectors' *
       centeredKernelMatrix;
end
```

1.6 computeGaussianKernel

This function computes the Gaussian kernel matrix for a given dataset.

Inputs:

- **grid:** The input data matrix where each column is a data point.
- **sigma:** The bandwidth of the Gaussian kernel.

Outputs:

- **gaussianKernelMatrix:** The Gaussian kernel matrix.

The function calculates the Gaussian kernel values for each pair of data points in the input matrix and stores them in the kernel matrix.

```
function gaussianKernelMatrix =
   computeGaussianKernel(grid, sigma)
    % This function computes the Gaussian kernel
       matrix for the input grid.

    [~, numColumns] = size(grid);
    gaussianKernelMatrix = zeros(numColumns,
       numColumns);

    for j = 1:numColumns
        baseVector = grid(:, j);
        for k = 1:numColumns
            kernelValue = gKernel(baseVector',
               grid(:, k)', sigma);
            gaussianKernelMatrix(j, k) =
               kernelValue;
        end
    end
end
```

1.7 polynomialKernel

This function calculates the polynomial kernel value between two vectors x and y using a specified offset and degree.

Inputs:

- **x:** A vector.
- **y:** Another vector of the same size as x.
- **offset:** The offset to be added to the inner product.
- **degree:** The degree of the polynomial.

Outputs:

- **kernelValue:** The polynomial kernel value.

The function computes the inner product of x and y, adds the offset, raises the result to the specified degree, and returns the kernel value.

```
function kernelValue = polynomialKernel(x, y,
   offset, degree)
    % This function calculates the polynomial
       kernel value between x and y
    % using the given offset and degree.

    innerProduct = x' * y;
    kernelValue = (innerProduct + offset) .^ degree;
end
```

1.8 main

To visualize different shapes and structures with Polynomial Kernel PCA and Gaussian Kernel PCA, we can adjust the parameters passed to these kernels.

For Polynomial Kernel PCA:

- **Offset (c)**: This parameter controls the offset in the polynomial kernel. Increasing it generally makes the transformation less sensitive to smaller differences.
- **Degree (d):** This parameter controls the degree of the polynomial. Higher degrees can capture more complex structures but may also lead to overfitting.

For Gaussian Kernel PCA:

- **Sigma(σ):** This parameter controls the width of the Gaussian kernel. Smaller values of σ make the kernel more sensitive to closer points, while larger values make it more smooth and general.

Polynomial Kernel PCA:

- **polynomialParams1 = [1, 2]**: This sets a lower offset and degree, which might capture less complex structures.
- **polynomialParams2 = [5, 3]:** This sets a higher offset and degree, which might capture more complex structures.

Gaussian Kernel PCA:

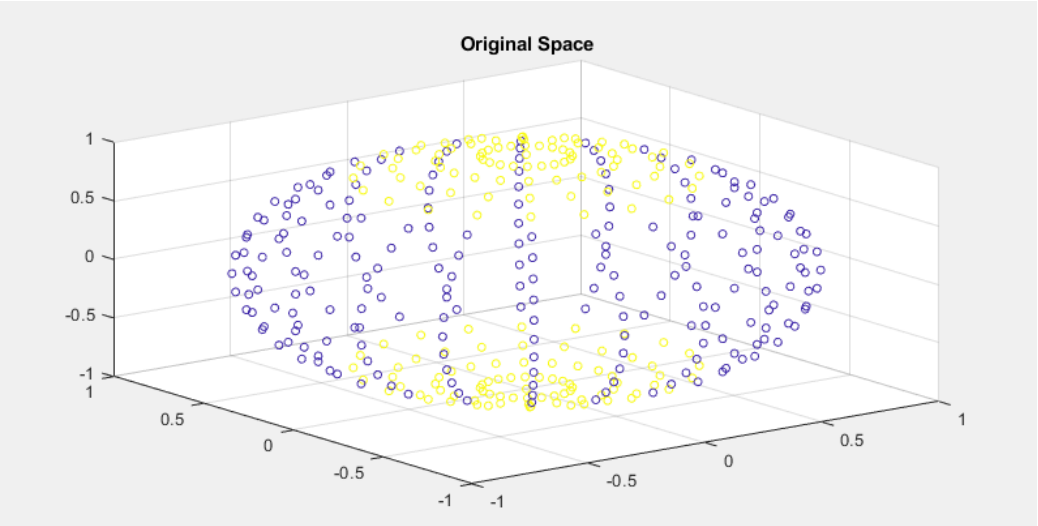
- **gaussianParams1 = [5]:** This sets a lower sigma, making the kernel more sensitive to closer points.
- **gaussianParams2 = [20]:** This sets a higher sigma, making the kernel smoother and less sensitive to small differences.

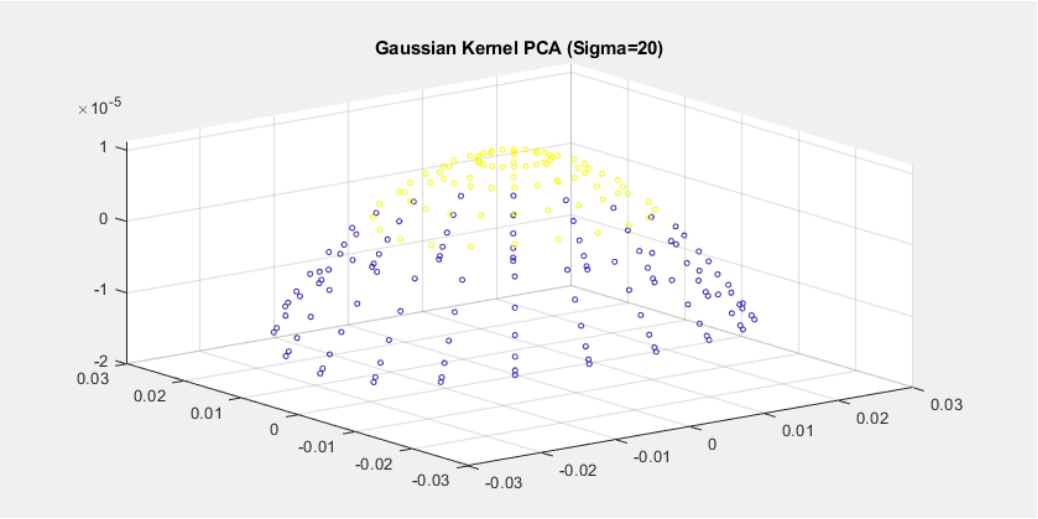
By adjusting these parameters and rerunning the script, you can visualize how different kernel parameters affect the shape and structure of the transformed data. You can add more parameter sets and plot them similarly to explore a wide range of transformations.

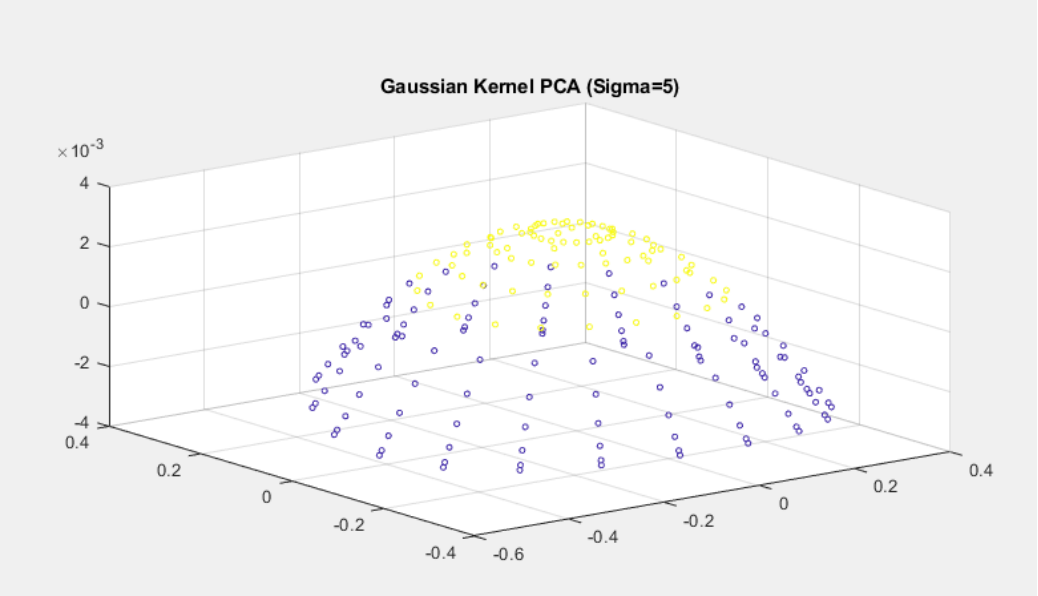
```
% Clear the workspace and close all figures
clear
close all

% Generate data and plot the original 3D data
[featureX, featureY, featureZ, samples, labels] =
    generateData();
figure;
subplot(2, 2, 1)
scatter3(featureX, featureY, featureZ, 20, labels)
title('Original Space');
hold on

% Parameters for Polynomial Kernel PCA
polynomialParams1 = [1, 2]; % Example 1: Lower
    offset and degree
polynomialParams2 = [5, 3]; % Example 2: Higher
    offset and degree

% Perform Polynomial Kernel PCA and plot the
    transformed data (Example 1)
data = samples';
percentVarianceRetained = 100;
[transformedDataPoly1, eigenVectorsPoly1,
    eigenValuesPoly1] = KernelPCA(data,
    percentVarianceRetained, 'polynomial',
    polynomialParams1);
subplot(2, 2, 2)
scatter3(transformedDataPoly1(1, :),
    transformedDataPoly1(2, :),
    transformedDataPoly1(3, :), 10, labels, 'O')
title('Polynomial Kernel PCA (Offset=1, Degree=2)');

% Perform Polynomial Kernel PCA and plot the
    transformed data (Example 2)
[transformedDataPoly2, eigenVectorsPoly2,
    eigenValuesPoly2] = KernelPCA(data,
    percentVarianceRetained, 'polynomial',
    polynomialParams2);
subplot(2, 2, 3)
```

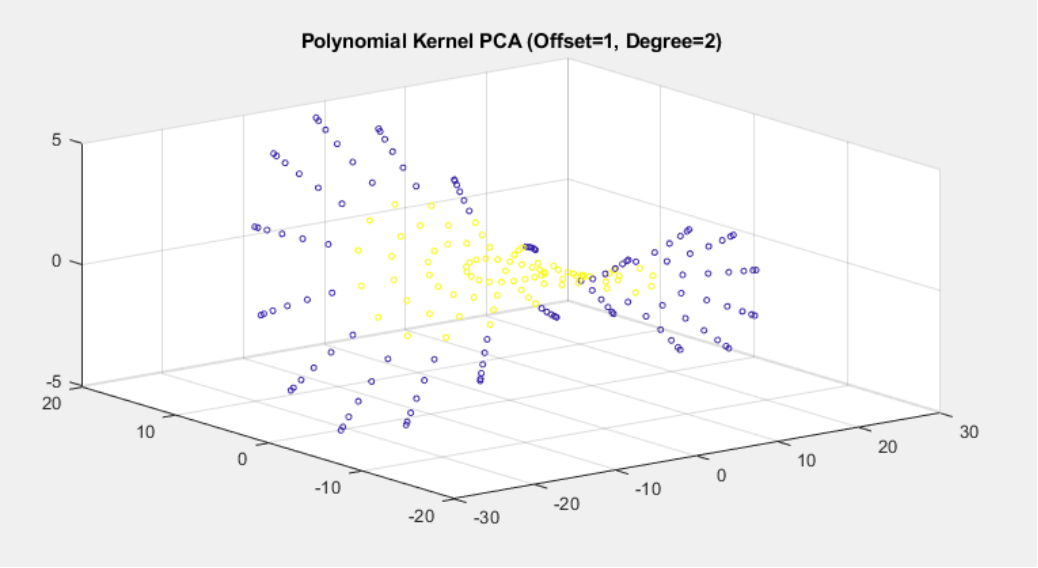
```
scatter3(transformedDataPoly2(1, :),
    transformedDataPoly2(2, :),
    transformedDataPoly2(3, :), 10, labels, 'O')
title('Polynomial Kernel PCA (Offset=5, Degree=3)');

% Parameters for Gaussian Kernel PCA
gaussianParams1 = [5];  % Example 1: Lower sigma
gaussianParams2 = [20]; % Example 2: Higher sigma

% Perform Gaussian Kernel PCA and plot the
    transformed data (Example 1)
[transformedDataGauss1, eigenVectorsGauss1,
    eigenValuesGauss1] = KernelPCA(data,
    percentVarianceRetained, 'gaussian',
    gaussianParams1);
subplot(2, 2, 4)
scatter3(transformedDataGauss1(1, :),
    transformedDataGauss1(2, :),
    transformedDataGauss1(3, :), 10, labels, 'O')
title('Gaussian Kernel PCA (Sigma=5)');

% Perform Gaussian Kernel PCA and plot the
    transformed data (Example 2)
figure;
[transformedDataGauss2, eigenVectorsGauss2,
    eigenValuesGauss2] = KernelPCA(data,
    percentVarianceRetained, 'gaussian',
    gaussianParams2);
subplot(2, 2, 1)
scatter3(transformedDataGauss2(1, :),
    transformedDataGauss2(2, :),
    transformedDataGauss2(3, :), 10, labels, 'O')
title('Gaussian Kernel PCA (Sigma=20)');
```

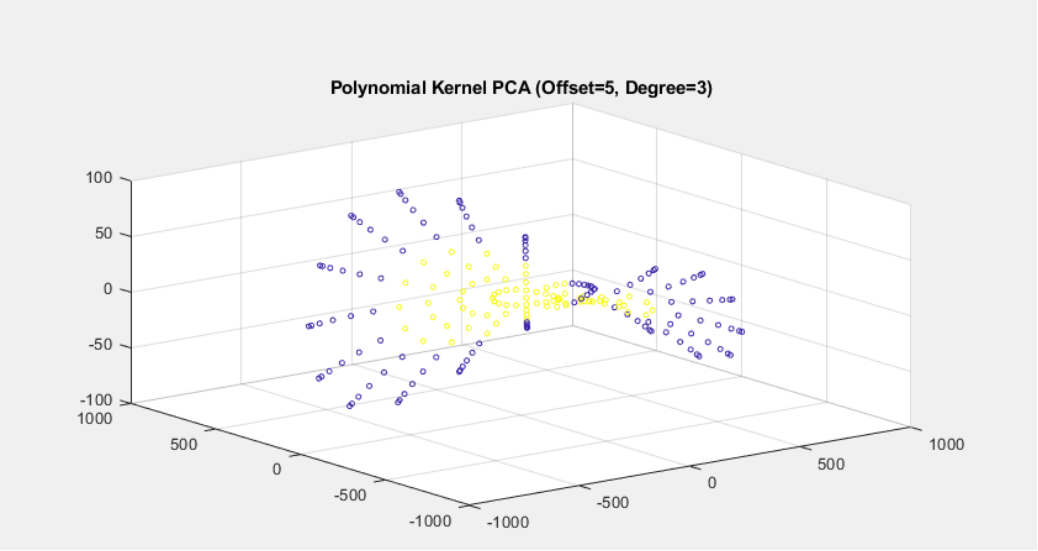
2 Results


Original Space
1
0.5
0
-0.5
-1
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1


Gaussian Kernel PCA (Sigma=20)
×10^-5
1
0
-1
-2
0.03
0.02
0.01
0
-0.01
-0.02
-0.03
-0.03
-0.02
-0.01
0
0.01
0.02
0.03


Gaussian Kernel PCA (Sigma=5)
×10^-3
4
2
0
-2
-4
0.4
0.2
0
-0.2
-0.4
-0.6
-0.4
-0.2
0
0.2
0.4


Polynomial Kernel PCA (Offset=1, Degree=2)
5
0
-5
20
10
0
-10
-20
-30
-20
-10
0
10
20
30


Polynomial Kernel PCA (Offset=5, Degree=3)
100
50
0
-50
-100
1000
500
0
-500
-1000
-1000
-500
0
500
1000