Department of Informatics

M.Sc. in Data Science

Numerical optimization and Large Scale Linear Algebra

Kernel Principal Component Analysis

Instructors: Paris Vassalos

Authors: Antonios Mavridis (p3352215)

25/07/2023

Table of Contents

1 Introduction - Experiment Overview

This technical report delves into the application of Principal Component Analysis (PCA) and Gaussian kernel PCA in the field of image processing and classification. Our study is inspired by and seeks to simulate the experiment conducted by Quan Wang in his seminal paper "Kernel Principal Component Analysis and its Applications in Face Recognition and Active Shape Models". In our study, we utilize images with pixel intensities of 168×192, resulting in an original feature vector that is 32256-dimensional. Our primary objective is to extract the nine most significant features from the training data using both standard PCA and Gaussian kernel PCA, and subsequently record the eigenvectors.

In the case of standard PCA, the extraction process only requires the eigenvectors. However, Gaussian kernel PCA necessitates both the eigenvectors and the training data. A significant part of our study involves the use of a Gaussian kernel with $\sigma = 22546$ in kernel PCA. For the classification process, we employ the simplest linear classifier.

Our code includes functions for loading the dataset, performing PCA and kPCA, computing a sparse distance matrix, and computing a sparse kernel matrix. We also implemented a function to compute the value of sigma used in the Gaussian kernel based on the mean distance to the nearest neighbors.

Finally, the report will present a comparative analysis of the training error rates and testing error rates for standard PCA and Gaussian kernel PCA. By replicating and expanding upon Quan Wang's experiment, this report aims to contribute to the ongoing discourse on image processing and classification, and the role of PCA methodologies within it.

2 Code Analysis

Our code implements the experiment described above. The code is organized into several functions, each performing a specific task. Here is a detailed breakdown of the code:

1. **loadmat_v73(file_name)**: This function loads a MATLAB v7.3 format file and returns the data as a dictionary.
2. **PCA_sparse(X, d):** This function performs Sparse Principal Component Analysis (PCA) on a data matrix X and reduces its dimensionality to d. It returns the reduced data matrix, the eigenvectors, and the eigenvalues.
3. **sparse_distance_matrix(X, num_neighbors=5)**: This function computes a sparse pairwise distance matrix based on the data matrix X using nearest neighbors.
4. **kernel(X, kernel_type, para)**: This function computes a sparse kernel matrix based on the data matrix X and the specified kernel type. The kernel can be 'simple', 'poly', or 'gaussian'.
5. **sparse_distance_matrix_new_data(X, Y=None, num_neighbors=5)**: This function computes a sparse pairwise distance matrix between X and Y using nearest neighbors.
6. **kernel_new_data(Y, X, kernel_type, para)**: This function computes a sparse kernel matrix between new data Y and training data X based on the specified kernel type.
7. **compute_sigma(X, num_neighbors=5)**: This function computes the value of sigma used in the Gaussian kernel based on the mean distance to the nearest neighbors.
8. **kpca(X, d, kernel_type, para)**: This function performs Kernel Principal Component Analysis (KPCA) on a data matrix X.

Below are represented the function **PCA_sparse(X, d)**:

```
def PCA_sparse(X, d):
    """
    Performs dimensionality reduction using Sparse Principal Component Analysis (PCA)
    on a data matrix.

    Parameters:
    - X: Data matrix where each row represents an observation
    and each column represents a feature.
    - d: The desired reduced dimension.

    Returns:
    - Y: The dimensionality-reduced data matrix.
    - eigVector: The eigenvectors corresponding to the principal components.
    - eigValue: The eigenvalues associated with the principal components.
    """

    X = sparse.csr_matrix(X) # convert X to sparse matrix if it is not already


    # eigenvalue analysis
    X_mean = X.mean(axis=0)  # calculate mean
    X = X - X_mean  # centering

    def matvec(v):
        v = v.reshape(-1, 1)  # reshape the input vector
        return (X.T @ (X @ v)).ravel()  # ravel to make sure the output is 1D
```

```
    # Since our operator is symmetric (i.e., X.T @ X equals its transpose),
    # rmatvec is the same as matvec
    rmatvec = matvec

    Sx_op = LinearOperator((X.shape[1], X.shape[1]), matvec=matvec, rmatvec=rmatvec)

    # Adjust this line to receive the singular values and the left and right singular vectors
    U, eigValue, Vt = svds(Sx_op, k=d)  # calculate eigenvalues and eigenvectors

    IX = np.argsort(eigValue)[::-1]  # sort eigenvalues in descending order and get indices
    eigValue = eigValue[IX]
    eigVector = Vt.T[:, IX]  # Use the right singular vectors (transpose of Vt) as eigenvectors

    # normalization
    norm_eigVector = np.sqrt(np.sum(eigVector**2, axis=0))
    eigVector = eigVector / norm_eigVector

    Y = X @ eigVector  # sparse matrix multiplication

    return Y, eigVector, eigValue
```

The function takes two inputs:

- **X**: This is the data matrix where each row represents an observation and each column represents a feature. This data matrix is transformed into a sparse matrix using the sparse.csr_matrix(X) function. Sparse matrices are used to store data that contain mostly zeros, which helps to save memory and computational resources.
- **d**: This is the desired reduced dimension, which is the number of principal components that the function will return.

Here's what the function does in detail:

1. **Centering the Data**: It first calculates the mean of X along the rows (i.e., X_mean = X.mean(axis=0)) and then subtracts this mean from X to center the data.
2. **Eigendecomposition**: It then creates a LinearOperator object, which applies matrix multiplication and transposition efficiently. The function matvec(v) is defined to return the product of X.T @ X @ v, where X.T is the transpose of X, @ is the matrix multiplication operator, and v is a vector. The rmatvec is set to be the same as matvec because X.T @ X is symmetric (i.e., equals its own transpose).
3. **Singular Value Decomposition (SVD)**: Using the svds function, it calculates the left singular vectors (U), singular values (eigValue), and right singular vectors (Vt). Here, Sx_op is the LinearOperator and k=d is the number of singular values/vectors to compute.
4. **Sorting**: The eigenvalues are then sorted in descending order, and the corresponding order of eigenvectors (Vt.T) is also rearranged.
5. **Normalization**: The eigenvectors are normalized to have a unit norm. This is done by dividing each eigenvector by its length (calculated using the Euclidean norm).
6. **Dimensionality Reduction**: Finally, it multiplies the centered data matrix X by the normalized eigenvectors to get the dimensionality-reduced data matrix Y.

The function returns three outputs:

1. **Y**: The dimensionality-reduced data matrix.

2. **eigVector**: The eigenvectors corresponding to the principal components.
3. **eigValue**: The eigenvalues associated with the principal components.

Sparse Principal Component Analysis (PCA) using singular value decomposition (SVD) and sparse matrices, as implemented in this function, is specifically designed to address memory management issues.

In traditional PCA, the computation of covariance matrix and its eigendecomposition can become memory intensive and computationally prohibitive when dealing with high-dimensional datasets. This happens because the size of the covariance matrix is proportional to the square of the number of features, and thus can quickly consume a lot of memory for datasets with many features.

In this case, the function is using SVD and the concept of sparse matrices to alleviate this issue. The svds function is an iterative algorithm for SVD, which is more efficient than standard SVD implementations for large matrices, as it doesn't need to compute all the singular values and vectors, but only the top k ones.

Sparse matrices are used to represent the data matrix X. A sparse matrix is a matrix that is comprised of mostly zero values. Sparse matrices are used to efficiently represent large matrices with lots of zero values in a way that requires significantly less memory.

The sparse representation of X and the iterative svds function allows the function to perform PCA on high-dimensional data in a memory-efficient way, making it a suitable choice for big data applications where memory usage can be a limiting factor.

Below are represented the function **sparse_distance_matrix**:

```
def sparse_distance_matrix(X, num_neighbors=5):
    """
    Computes a sparse pair-wise distance matrix based on the data matrix using nearest neighbors.

    Parameters:
    - X: Data matrix where each row represents an observation
    and each column represents a feature.
    - num_neighbors: Number of nearest neighbors to consider, not including self.

    Returns:
    - D: Sparse pair-wise distance matrix.
    """
    # Ensure X is a sparse matrix
    X = sparse.csr_matrix(X)

    # Use sklearn's NearestNeighbors to find nearest neighbors
    knn = NearestNeighbors(n_neighbors=num_neighbors+1, metric='manhattan')
    knn.fit(X)

    # Get the distances and indices of the nearest neighbors
    distances, indices = knn.kneighbors(X)

    # Build the sparse distance matrix
    m, _ = X.shape
    D = sparse.lil_matrix((m, m))

    for i in range(m):
        D[i, indices[i]] = distances[i]

    return D.tocsr()
```

This function **sparse_distance_matrix** computes a sparse pair-wise distance matrix for a given

data matrix using the concept of nearest neighbors. This sparse pair-wise distance matrix, as the name suggests, represents the distances between each pair of points (rows) in the data matrix, but it only stores distances to the nearest neighbors of each point to save memory.

Here are the inputs:

- **X**: This is the data matrix where each row represents an observation and each column represents a feature.
- **num_neighbors**: This parameter specifies the number of nearest neighbors to consider for each point. Note that this doesn't include the point itself.

This is what the function does in detail:

1. **Convert to Sparse Matrix**: It first converts the input data matrix X to a sparse matrix if it's not already one, using the sparse.csr_matrix(X) function.
2. **Compute Nearest Neighbors**: It then utilizes the NearestNeighbors class from the sklearn.neighbors module to find the num_neighbors+1 nearest neighbors for each observation in X (we add 1 because the point itself is also included in the neighbors). The metric used to calculate the distance between observations is the 'manhattan' distance, but this could be changed to other distance metrics like 'euclidean' depending on the application.
3. **Get Distances and Indices**: After fitting the NearestNeighbors model to X, it calculates the distances to and indices of the nearest neighbors for each observation in X using the kneighbors(X) method.
4. **Build Sparse Distance Matrix**: Next, it initializes an empty sparse matrix D of size m x m where m is the number of observations in X. It then iterates over each observation and sets the distances to its nearest neighbors in the corresponding row of D. This is done using the indices and distances obtained earlier.

Finally, it returns the sparse pair-wise distance matrix D. By only storing the distances to the nearest neighbors and using a sparse representation, this function can efficiently compute and store the pair-wise distance matrix for large, high-dimensional datasets.

Below are represented the function **kernel(X, kernel_type, para)**:

```
def kernel(X, kernel_type, para):
    """
    Computes a sparse kernel matrix based on the data matrix and specified kernel type.

    Parameters:
    - X: Data matrix where each row represents an observation
    and each column represents a feature.
    - kernel_type: Type of kernel, can be 'simple', 'poly', or 'gaussian'.
    - para: Parameter for computing the 'poly' kernel. For 'simple' and 'gaussian',
    it will be ignored.

    Returns:
    - K: Sparse kernel matrix.
    """
    # Ensure X is a sparse matrix
    X = csr_matrix(X)

    if kernel_type == 'simple':
        K = X.dot(X.transpose())
    elif kernel_type == 'poly':
```

```
        K = X.dot(X.transpose()) + 1
        K = K.power(para)
    elif kernel_type == 'gaussian':
        D = sparse_distance_matrix(X)
        D.data **= 2  # square the distances in-place
        K = sparse.csr_matrix(np.exp(-D.toarray() / (2 * para**2)))
    else:
        raise ValueError(f"Unknown kernel_type: {kernel_type}")

    return K
```

This function **kernel** computes a sparse kernel matrix for a given data matrix, a specified kernel type, and a parameter for the 'poly' kernel. A kernel matrix, or Gram matrix, is a matrix whose elements are the result of applying a kernel function to pairs of data points. The kernel function effectively calculates the similarity or "inner product" between pairs of points in a (possibly infinite-dimensional) feature space.

Here are the inputs:

- **X**: This is the data matrix where each row represents an observation and each column represents a feature.
- **kernel_type**: This parameter specifies the type of kernel function to use. It can be 'simple', 'poly', or 'gaussian'.
- **para**: This is a parameter for computing the 'poly' kernel. For 'simple' and 'gaussian', it will be ignored.

Here's what the function does in detail:

1. **Convert to Sparse Matrix**: It first converts the input data matrix X to a sparse matrix if it's not already one, using the csr_matrix(X) function.
2. **Compute Kernel Matrix**: Depending on the specified kernel_type, it computes the sparse kernel matrix K in one of three ways:
   (a) If kernel_type is 'simple', it simply takes the dot product of X with its transpose, which is equivalent to calculating the inner product of each pair of observations in X.
   (b) If kernel_type is 'poly', it calculates a polynomial kernel. This involves first computing the dot product of X with its transpose, adding 1 to each element, and then raising the result to the power of para.
   (c) If kernel_type is 'gaussian', it computes a Gaussian (RBF) kernel. This involves first calculating the pair-wise distance matrix D of X using the sparse_distance_matrix(X) function, squaring each distance, and then applying the Gaussian kernel function exp(-D / (2 * para**2)).
   (d) If kernel_type is not any of the above, it raises a ValueError with a descriptive message.

Finally, it returns the sparse kernel matrix K. This function is an efficient way to compute the kernel matrix for large, high-dimensional datasets because it stores the result as a sparse matrix, which saves memory when most of the elements are zero. It's also flexible because it can compute different types of kernel matrices depending on the specified kernel_type.

Below are represented the function **sparse_distance_matrix_new_data(X, Y=None, num_neighbors=5)**:

```
def sparse_distance_matrix_new_data(X, Y=None, num_neighbors=5):
    """
    Computes a sparse pair-wise distance matrix
```

```
    between X and Y using nearest neighbors.

    Parameters:
    - X: Data matrix where each row represents an
    observation and each column represents a feature.
    - Y: New data matrix.
    - num_neighbors: Number of nearest neighbors
    to consider, not including self.

    Returns:
    - D: Sparse pair-wise distance matrix.
    """
    # Ensure X and Y are sparse matrices
    X = sparse.csr_matrix(X)
    Y = sparse.csr_matrix(Y)

    # Use sklearn's NearestNeighbors to find nearest neighbors
    knn = NearestNeighbors(n_neighbors=num_neighbors+1, metric='manhattan')
    knn.fit(X)

    # Get the distances and indices of the nearest neighbors
    distances, indices = knn.kneighbors(Y)

    # Build the sparse distance matrix
    m, _ = Y.shape
    n, _ = X.shape
    D = sparse.lil_matrix((m, n))

    for i in range(m):
        D[i, indices[i]] = distances[i]

    return D.tocsr()
```

This Python function sparse_distance_matrix_new_data computes a sparse pairwise distance matrix between two data matrices, X and Y, using the concept of nearest neighbors. Here, X is the original data matrix and Y is a new data matrix for which we want to compute distances to the nearest points in X. This can be particularly useful in scenarios like predicting outcomes for new data points based on their similarity to known data points.

Here are the inputs:

- **X**: This is the original data matrix where each row represents an observation and each column represents a feature.
- **Y**: This is the new data matrix for which we want to compute distances to the points in X.
- **num_neighbors**: This parameter specifies the number of nearest neighbors in X to consider for each point in Y. Note that this doesn't include the point itself.

This is what the function does in detail:

1. **Convert to Sparse Matrix**: It first converts both X and Y to sparse matrices if they're not already, using the sparse.csr_matrix() function.
2. **Compute Nearest Neighbors**: It then utilizes the NearestNeighbors class from the sklearn.neighbors module to find the num_neighbors+1 nearest neighbors in X for each observation in Y. The metric used to calculate the distance between observations is the 'manhattan' distance.

3. **Get Distances and Indices**: After fitting the NearestNeighbors model to X, it calculates the distances to and indices of the nearest neighbors in X for each observation in Y using the kneighbors(Y) method.

4. **Build Sparse Distance Matrix**: Next, it initializes an empty sparse matrix D of size m x n where m is the number of observations in Y and n is the number of observations in X. It then iterates over each observation in Y and sets the distances to its nearest neighbors in X in the corresponding row of D. This is done using the indices and distances obtained earlier.

Finally, it returns the sparse pairwise distance matrix D as a csr (Compressed Sparse Row) matrix. This function is an efficient way to compute and store the pairwise distance matrix for large, high-dimensional datasets, because it only stores distances to the nearest neighbors and uses a sparse representation.

Below are represented the function **kernel_new_data(Y, X, kernel_type, para)**:

```
def kernel_new_data(Y, X, kernel_type, para):
    """
    Computes a sparse kernel matrix between new data Y and training
    data X based on the specified kernel type.

    Parameters:
    - Y: New data matrix.
    - X: Training data matrix where each row represents an observation
    and each column represents a feature.
    - kernel_type: Type of kernel, can be 'simple', 'poly', or 'gaussian'.
    - para: Parameter for computing the 'poly' kernel. For 'simple' and 'gaussian',
    it will be ignored.

    Returns:
    - K: Sparse kernel matrix.
    """
    # Ensure X and Y are sparse matrices
    X = sparse.csr_matrix(X)
    Y = sparse.csr_matrix(Y)

    if kernel_type == 'simple':
        K = Y.dot(X.transpose())
    elif kernel_type == 'poly':
        K = Y.dot(X.transpose()) + 1
        K = K.power(para)
    elif kernel_type == 'gaussian':
        D = sparse_distance_matrix_new_data(X, Y)
        D.data **= 2  # square the distances in-place
        D_dense = D.toarray()  # Convert sparse matrix to dense array
        K = np.exp(-D_dense / (2 * para**2))
    else:
        raise ValueError(f"Unknown kernel_type: {kernel_type}")

    return K
```

The **kernel_new_data** function is used to compute a sparse kernel matrix between a new data matrix Y and a training data matrix X, based on a specified kernel type. It's a kernel function that is specifically designed to compute the similarity between a new data set and an existing data set.

Here's a breakdown of the inputs:

- **Y**: New data matrix, each row represents a new observation and each column represents a feature.
- **X**: Training data matrix, each row represents an observation and each column represents a feature.
- **kernel_type**: A string specifying the type of kernel to use, which can be 'simple', 'poly', or 'gaussian'.
- **para**: A parameter for computing the 'poly' kernel. For 'simple' and 'gaussian', it will be ignored.

Here's a detailed step-by-step description of the function:

1. **Convert to Sparse Matrix**: Both X and Y are first converted to sparse matrices, if they are not already, using the sparse.csr_matrix() function.
2. **Compute Kernel Matrix**: Depending on the specified kernel_type, it computes the kernel matrix K:
   (a) If kernel_type is 'simple', it computes the dot product of Y with the transpose of X. The resulting matrix has the shape of (Y.shape[0], X.shape[0]), where each element represents the dot product between a row in Y and a row in X.
   (b) If kernel_type is 'poly', it computes a polynomial kernel. This first involves computing the dot product of Y with the transpose of X, adding 1 to each element, and then raising the result to the power of para. The result is a matrix of the same size as for the 'simple' kernel, but the elements are transformed by the polynomial function.
   (c) If kernel_type is 'gaussian', it computes a Gaussian (or Radial Basis Function) kernel. This involves first calculating the pairwise distance matrix D of X and Y using the sparse_distance_matrix_new_data(X, Y) function, squaring each distance, converting the distance matrix to a dense array, and then applying the Gaussian kernel function exp(-D / (2 * para**2)) to each element of the array.
   (d) If kernel_type is not any of the above, it raises a ValueError with an informative message.

The function finally returns the computed sparse kernel matrix K. It is an efficient way of storing and processing high-dimensional data where most of the elements in the resulting kernel matrix are expected to be zero.

Below are represented the function **compute_sigma(X, num_neighbors=5)**:

```
def compute_sigma(X, num_neighbors=5):
    """
    Computes the value of sigma used in the Gaussian kernel based
    on the mean distance to the nearest neighbors.

    Parameters:
    - X: Data matrix where each row represents an
    observation and each column represents a feature.
    - num_neighbors: Number of nearest neighbors to consider, not including self.

    Returns:
    - sigma: Value of sigma for the Gaussian kernel.
    """
    knn = NearestNeighbors(n_neighbors=num_neighbors+1, metric='euclidean')
    knn.fit(X)
    distances, _ = knn.kneighbors(X)
```

```
    # Compute the mean of the distances to the nearest neighbors
    mean_d_NN = np.mean(distances[:, 1:])

    # Compute  based on the formula
    sigma = 5 * mean_d_NN

    return sigma
```

The function **compute_sigma** calculates the value of sigma that is used in the Gaussian kernel, which is based on the mean distance to a specific number of nearest neighbors in the dataset.

Here are the inputs:

- **X**: A data matrix where each row represents an observation and each column represents a feature.
- **num_neighbors**: The number of nearest neighbors to consider when calculating the distances, not including the data point itself.

Here is a detailed explanation of what the function does:

1. **Create Nearest Neighbors Model**: It creates a model instance of the NearestNeighbors class from the sklearn.neighbors module with n_neighbors=num_neighbors+1 and metric='euclidean'. Here, the metric used to calculate the distance between observations is the Euclidean distance.
2. **Fit the Model and Compute Distances**: The NearestNeighbors model is fitted to the data matrix X. After fitting the model, it computes the distances and the indices of the num_neighbors+1 nearest neighbors for each observation in X using the kneighbors(X) method.
3. **Compute Mean Distance to Nearest Neighbors**: The distances to the nearest neighbors are stored in the distances matrix. We are interested in the distances to the num_neighbors nearest neighbors, excluding the point itself. Therefore, it slices the distances matrix to exclude the first column (which represents the distance of each point to itself), and calculates the mean of these distances using np.mean(distances[:, 1:]).
4. **Compute Sigma**: Finally, it computes the value of sigma based on a chosen formula: sigma = 5 * mean_d_NN, where mean_d_NN is the mean distance to the nearest neighbors. This sigma value is used in the Gaussian kernel to control the width of the Gaussian function, with larger values resulting in a wider, more spread-out function.

The function returns this computed value of sigma. The computed sigma can then be used to calculate the Gaussian kernel. It's a crucial hyperparameter for any algorithm that relies on the Gaussian kernel, as it has a significant effect on the model's performance.

Below are represented the function **kpca(X, d, kernel_type, para)**:

```
def kpca(X, d, kernel_type, para):
    """
    Performs Kernel Principal Component Analysis (KPCA) on a data matrix X.

    Parameters:
    - X: Data matrix where each row represents an observation
    and each column represents a feature.
    - d: Reduced dimension.
    - kernel_type: Type of kernel, can be 'simple', 'poly', or 'gaussian'.
```

```
    - para: Parameter for computing the 'poly' and 'gaussian' kernel.
    For 'simple', it will be ignored.

    Returns:
    - Y: Dimensionality-reduced data.
    - eigVector: Eigen-vectors, which can be used for pre-image reconstruction.
    - eigValue: Eigenvalues.
    """

    if kernel_type not in ['simple', 'poly', 'gaussian']:
        print(f"\nError: Kernel type {kernel_type} is not supported. \n")
        return None, None, None

    if kernel_type == 'gaussian':
        sigma = compute_sigma(X)
        #sigma =para
        para = sigma

    N = X.shape[0]

    # Compute the kernel matrix
    K0 = kernel(X, kernel_type, para).toarray()  # convert to dense matrix
    oneN = np.ones((N, N)) / N
    K = K0 - oneN.dot(K0) - K0.dot(oneN) + oneN.dot(K0).dot(oneN)


    # Perform eigenvalue analysis
    eigValue, eigVector = eigs(K / N, k=d, which='LM')
    eigValue = eigValue.real
    eigVector = eigVector.real

    # Normalize the eigenvectors
    norm_eigVector = np.sqrt(np.sum(eigVector ** 2, axis=0))
    eigVector = eigVector / norm_eigVector

    # Perform dimensionality reduction
    Y = K0.dot(eigVector)

    return Y, eigVector, eigValue
```

This function **kpca** performs Kernel Principal Component Analysis (KPCA) on a data matrix X.

Here are the inputs:

- **X**: A data matrix where each row represents an observation and each column represents a feature.
- **d**: The reduced dimension. This is the number of dimensions we want to reduce our data to.
- **kernel_type**: The type of kernel to use for the analysis. It can be 'simple', 'poly', or 'gaussian'.
- **para**: The parameter for computing the 'poly' and 'gaussian' kernel. This is ignored if the kernel type is 'simple'.

Here's what the function does in detail:

1. **Check Kernel Type**: It checks if the provided kernel_type is supported. If not, it prints an error message and returns None, None, None.

2. **Compute Sigma if Kernel is Gaussian**: If the kernel type is 'gaussian', it computes the value of sigma using the compute_sigma(X) function and assigns it to para.

3. **Get the Number of Observations**: It assigns the number of observations (the number of rows in X) to the variable N.

4. **Compute the Kernel Matrix**: It calculates the kernel matrix using the kernel(X, kernel_type, para) function and converts the sparse matrix to a dense one. Then it calculates the centered kernel matrix K using the formula for centering a kernel matrix. This is a double-centering process to ensure that the mean of the observations is zero in the feature space.

5. **Perform Eigenvalue Analysis**: It uses the eigs function from the scipy.sparse.linalg module to compute the d largest eigenvalues and their corresponding eigenvectors of the centered kernel matrix. The division by N scales the centered kernel matrix.

6. **Normalize the Eigenvectors**: It normalizes the eigenvectors such that their lengths (L2-norm) are 1. This is a usual step in PCA and KPCA to ensure that the principal components are of unit length.

7. **Perform Dimensionality Reduction**: It projects the original data onto the space spanned by the leading eigenvectors to get the dimensionality-reduced data Y.

The function returns the following:

- **Y**: The dimensionality-reduced data matrix.
- **eigVector**: The normalized eigenvectors corresponding to the principal components.
- **eigValue**: The eigenvalues associated with the principal components.

KPCA is a nonlinear version of PCA that uses a kernel function to map the data into a higher-dimensional feature space, where it can apply PCA. This is especially useful when the data is not linearly separable or does not follow a Gaussian distribution.

3 Experimentation

Here is a step-by-step of our explanation using the above mentioned function:

1. **Loading the dataset**: The dataset 'YaleFaceData.mat' is loaded using a custom function loadmat_v73. This function is assumed to return a dictionary where each key-value pair corresponds to a variable in the .mat file.
2. **Splitting the dataset**: The loaded data is split into training features (train_x), training labels (train_t), testing features (test_x), and testing labels (test_t).
3. **Dimensionality reduction using PCA**: Next, PCA is performed on the training data, reducing its dimensionality to d (which is set to 9 here). The function PCA_sparse returns the PCA-transformed training data, the eigenvectors, and eigenvalues. Only the first d eigenvectors are kept.
4. **Transforming the test data using PCA**: The test data is then projected onto the PCA space by multiplying it with the PCA eigenvectors.
5. **Classification using PCA features**: A Perceptron classifier is trained on the PCA-transformed training data and labels. The zero_one_loss function is then used to compute the error rates on the PCA-transformed training and testing data.
6. **Dimensionality reduction using KPCA**: KPCA is performed on the training data with the Gaussian kernel, reducing its dimensionality to d. The function kpca returns the KPCA-transformed training data, the eigenvectors, and eigenvalues. Only the first d eigenvectors are kept.
7. **Transforming the test data using KPCA**: The test data is then projected onto the KPCA space by multiplying it with the KPCA-transformed training data and the KPCA eigenvectors.
8. **Classification using KPCA features**: A Perceptron classifier is trained on the KPCA-transformed training data and labels. The zero_one_loss function is then used to compute the error rates on the KPCA-transformed training and testing data.

The error rates for both PCA and KPCA are then printed.

	Standard PCA	Kernel PCA
Error Rate on Training Data	0.0980392156862745	0.38235294117647056
Error Rate on Testing Data	0.23076923076923073	0.42307692307692313

Table 1: Classification Results

Standard PCA achieved an error rate of approximately 0.098 (or 9.8%) on the training data and an error rate of about 0.231 (or 23.1%) on the testing data. Kernel PCA, on the other hand, had a significantly higher error rate of approximately 0.382 (or 38.2%) on the training data and about 0.423 (or 42.3%) on the testing data.

Given these results, we can observe that Standard PCA performed significantly better than Kernel PCA on both training and testing data. This implies that for this particular dataset and under these specific conditions (e.g., choice of kernel in Kernel PCA, number of dimensions reduced to, etc.), Standard PCA was more effective in reducing the dimensionality of the data without losing significant information important for classification.