DATA604 - Data Representation and Modeling
Final Project

Nischal Chandur

May 15, 2024

UID: 120116650

1 Original Data

The Fruit-360 Dataset is a comprehensive collection of images encompassing fruits, vegetables, and nuts.

- **Primary Link**: Kaggle - Fruit-360 Dataset
- **Secondary Link**: GitHub - Fruit-360 Dataset

Dataset Properties:

- **Total Images**: 90,483
- **Training Set Size**: 67,692 images
- **Test Set Size**: 22,688 images
- **Total Classes**: 131
- **Image Size**: 100×100 pixels

2 Pre-processing

For this project, I refined the Fruit-360 dataset to a more manageable and balanced subset.

1. **Class Reduction:** Manually trimmed the dataset from 131 to 16 visually distinct fruits.

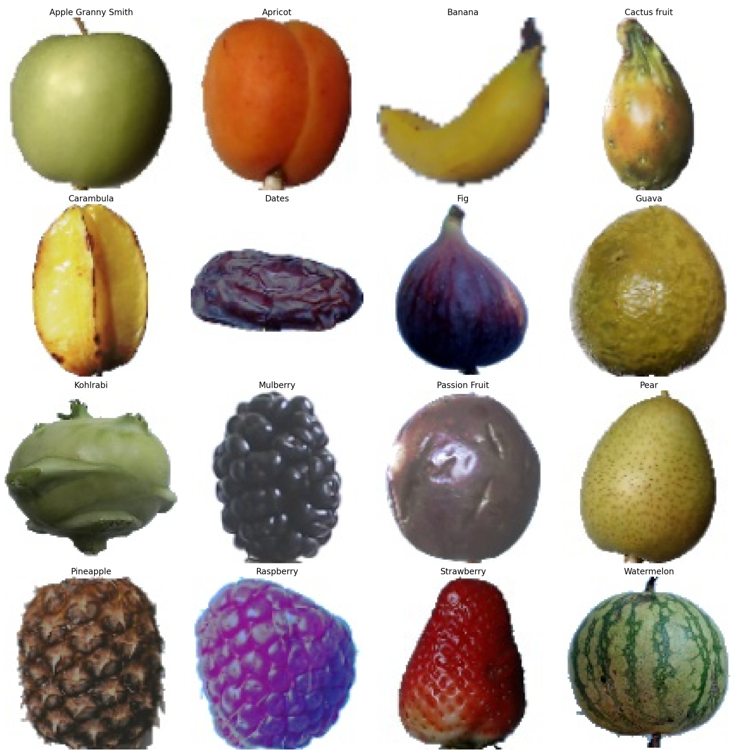

Figure 1: Fruit classes selected for this project

2. **Sample Reduction:** Decreased the samples per class from over 900 to 625.

3. **Color Conversion:** Converted images from RGB to grayscale.

4. **Image Resizing:** Resized images from $100 \times 100 \rightarrow 32 \times 32$ pixels.

5. **Data Organization:** Structured the dataset into a three-dimensional matrix format resembling the USPS Handwritten dataset: $16 \times 625 \times 1024$.

   - **First Dimension:** Represents fruit classes.
   - **Second Dimension:** Indicates the number of samples per class.
   - **Third Dimension:** Denotes the pixel values of individual samples.

Note: Codes for the pre-processing steps can be found in Section 7.1.

In the future, *Original Dataset* refers to this pre-processed dataset stored in matrix form.

3 Dimension Reduction Methods

In this section, we'll dive into different methods of Dimension Reduction to see how well they capture the essence of our dataset in fewer dimensions and to speculate on how a classification algorithm might perform with the transformed data. Four distinct methods have been selected for this study: PCA, kPCA with Gaussian and polynomial kernels, t-SNE, and Autoencoder embedding. To perform these transformations, I utilized the Matlab Toolbox for Dimensionality Reduction, developed by Laurens van der Maaten.

3.1 Original Dataset

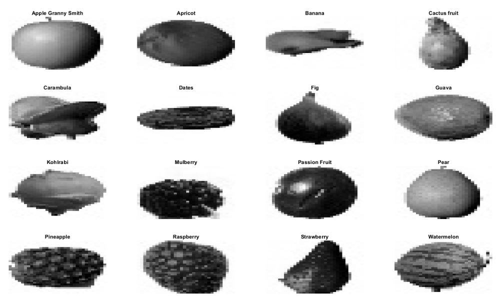

Figure 2: Samples of Original Dataset

3.2 Principal Component Analysis (PCA)

I applied PCA to the dataset, retaining all principal components sorted by the decreasing values of the eigenvalues. Here's how the samples from the previous image are represented after undergoing PCA transformation.

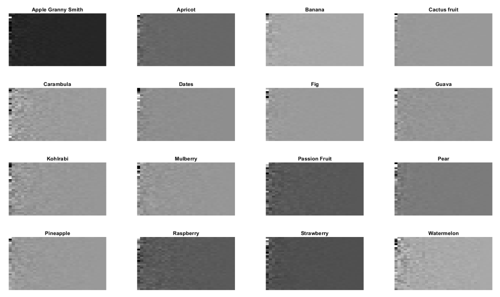

Figure 3: Data after PCA Transformation

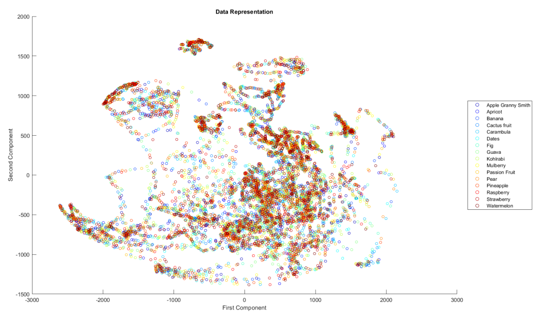

Figure 4: 2D Representation of transformed data

The 2D representation of the data following PCA reveals a well-spread distribution of samples across the space, spanning a range from approximately -3000 to 3000 on the x-axis and -1500 to 2000 on the y-axis. This suggests that in the lower dimension (2D), a significant amount of information from the higher dimension is preserved. Consequently, the classifier is expected to perform effectively on the data after PCA mapping.

3.3 Kernel Principal Component Analysis (kPCA)

I utilized kPCA on the dataset, preserving all principal components. This section explores the application of both Gaussian and polynomial kernels in the kPCA transformation process.

3.3.1 Gaussian kPCA

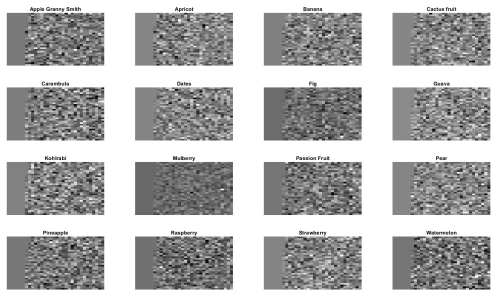

Figure 5: Data after Gaussian kPCA Transformation

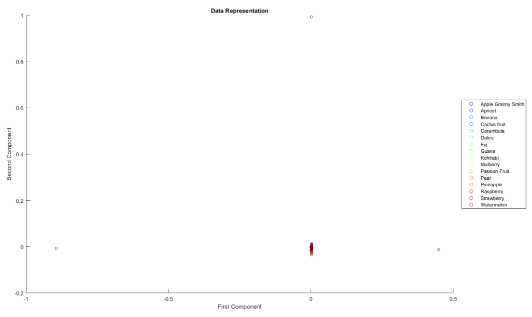

Figure 6: 2D Representation of transformed data

The 2D representation of the data post kPCA transformation with a Gaussian kernel indicates that most samples are clustered around 0, with minimal variation along the x-axis and slight dispersion along the y-axis, except for some outliers. This suggests that the lower dimensions may not adequately represent the data, and the information from the higher dimensions might not be effectively captured. Consequently, classification algorithms could encounter challenges with this representation. Additionally, it's worth noting that kPCA requires considerably more computational time compared to PCA, as highlighted in the table 2.

3.3.2 Polynomial kPCA

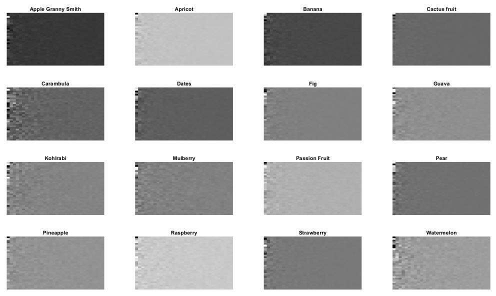

Figure 7: Data after Polynomial kPCA Transformation

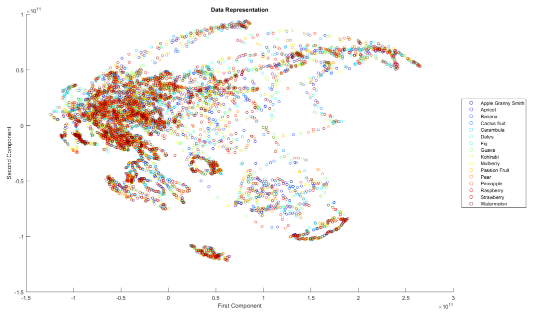

Figure 8: 2D Representation of transformed data

In contrast to the Gaussian kernel, the polynomial kernel presents the data with a well-distributed spread of samples across the 2D space. This indicates that the data is effectively represented in the lower dimension, with the information being preserved adequately. Consequently, the classifier is expected to perform well when kPCA is conducted using the polynomial kernel.

3.4 t-distributed Stochastic Neighbor (t-SNE) Embedding

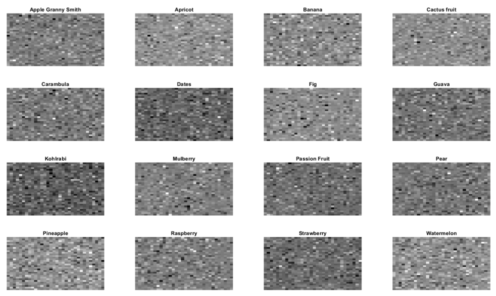

Figure 9: Data after t-SNE embedding

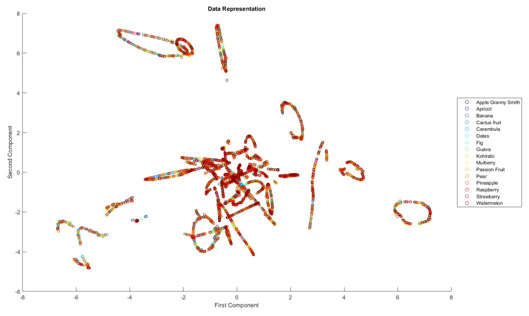

Figure 10: 2D Representation of transformed data

The 2D t-SNE representation displays a degree of dispersion among the samples, with various pockets and clusters scattered across the 2D space. This suggests that some information from the higher dimensions is reasonably well-captured in the lower dimensions, as evidenced by the decent spread of samples. Consequently, the classifier is anticipated to perform well on the data in this representation.

3.5 Autoencoder Embedding

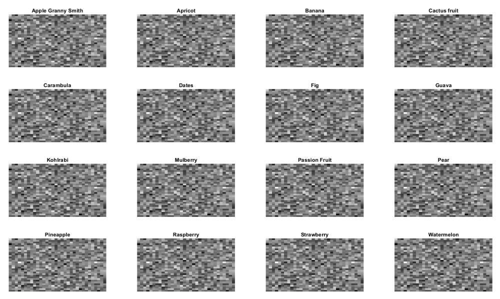

Figure 11: Data after Autoencoder embedding

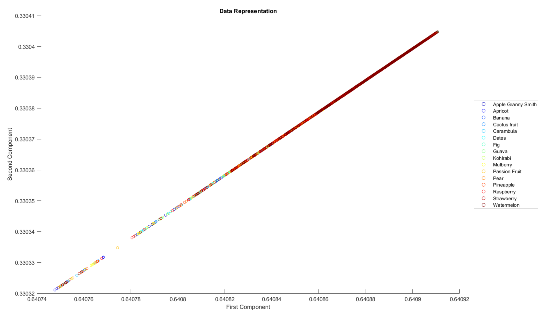

Figure 12: 2D Representation of transformed data

In the 2D autoencoder representation, the samples are depicted along an approximately straight line. However, this linear distribution indicates that the 2D space is not fully utilized. Moreover, unlike the spread observed in Polynomial kPCA and t-SNE embedding, there is a lack of dispersion across the 2D space, indicating that the information from higher dimensions may not be effectively captured in the lower dimensions. Consequently, the classifier may encounter difficulties in performing well on the dataset represented using autoencoder embedding.

Note:

- Codes to perform transformations can be found in Section 7.2.

- Codes to view samples of transformed data can be found in Section 7.3.
- Codes to view 2D representations of transformed data can be found in Section 7.4.

4 Classification Algorithms

In this section, we'll explore into two classification algorithms: KNN Classification and kSVM classification. These models will be compared based on three metrics: test accuracy, validation accuracy, and classification time. Then, we'll apply the best-performing algorithm from this comparison to our transformed datasets to evaluate their performance. We'll determine the number of principal components required to achieve high accuracy, providing insights into which method from the previous section most efficiently represents our dataset.

To compare the KNN Classifier and kSVM Classifier, I'll follow a standardized procedure using the original dataset. Initially, I'll shuffle samples within each class, a method proven to enhance classifier robustness, as learned from project 1. Subsequently, I'll partition the dataset with 400 samples per class for training, 200 samples per class for testing, and reserving 25 samples per class for the validation dataset.

Algorithm	**Test Accuracy**	**Validation Accuracy**	**Time Taken (s)**
KNN Classification	0.981875	0.990000	0.577916
kSVM Classification	0.999687	1.000000	354.926349

Table 1: Comparison of performance of KNN and kSVM Classifiers

From the table, it's evident that the Kernel SVM classifier marginally outperforms the KNN classifier, achieving higher test and validation accuracy. However, the Kernel SVM classifier also exhibits a substantially longer learning time compared to the KNN classifier. Moreover, the 100% validation accuracy of the Kernel SVM classifier raises concerns about potential overfitting. Therefore, based on these observations, I prefer proceeding with the KNN classifier for further exercises. It offers a high accuracy while requiring significantly less time for training compared to the Kernel SVM classifier.

Note: Codes for this comparison of the algorithms and the implementations of these algorithms can be found in Section 7.5.

In the next phase of this exercise, we'll investigate how varying the number of retained principal components in the transformed data impacts testing accuracy. This analysis aims to identify the optimal number of principal components that yield the highest testing accuracy.

4.1 Principal Component Analysis (PCA)

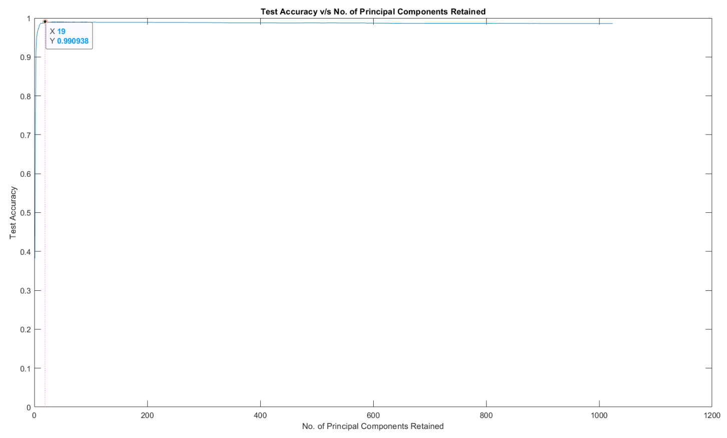

Figure 13: Test Accuracy v/s Number of Principal Components Used for Training

- Maximum accuracy: 0.9909, achieved using 19 principal components.
- Lower dimensions retain high-dimensional information effectively.
- The classifier is expected to perform well given the retained information.

4.2 Kernel Principal Component Analysis (kPCA)

4.2.1 Gaussian kPCA

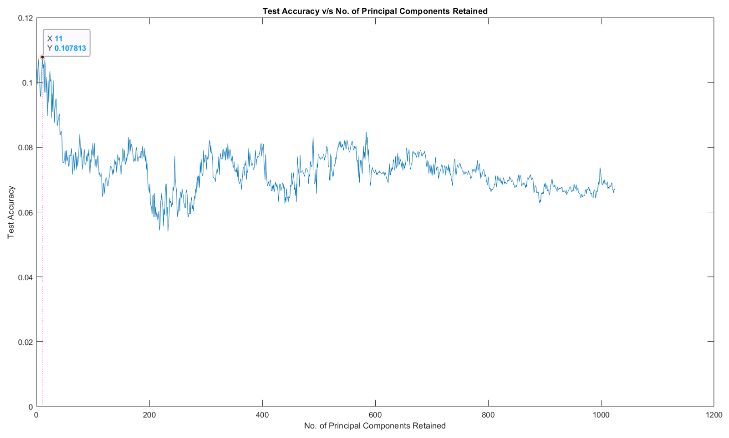

Figure 14: Test Accuracy v/s Number of Principal Components Used for Training

- Maximum accuracy: 0.1078, achieved using 11 principal components.
- Lower dimensions does not retain high-dimensional information effectively.
- The classifier is not expected to perform well given the retained information.

4.2.2 Polynomial kPCA

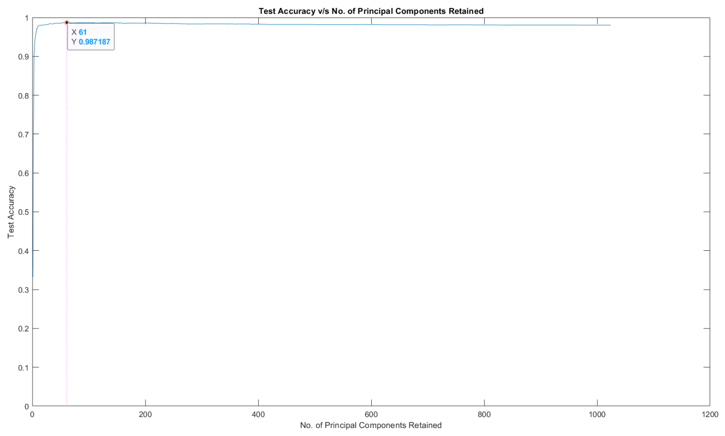

Figure 15: Test Accuracy v/s Number of Principal Components Used for Training

- Maximum accuracy: 0.9872, achieved using 61 principal components.
- Lower dimensions retain high-dimensional information effectively.
- The classifier is expected to perform well given the retained information.

4.3 t-distributed Stochastic Neighbor (t-SNE) Embedding

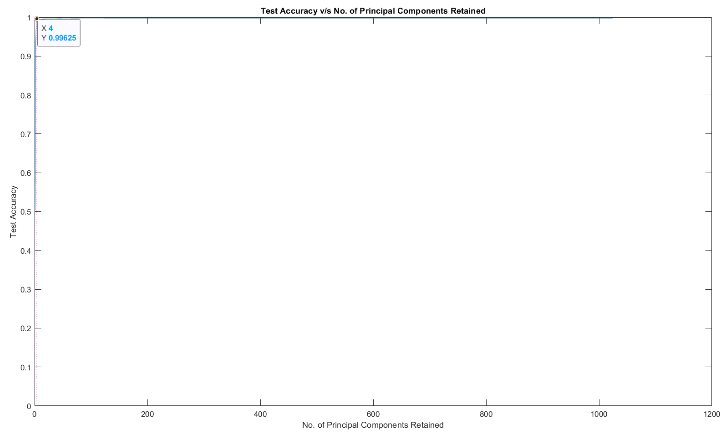

Figure 16: Test Accuracy v/s Number of Principal Components Used for Training

- Maximum accuracy: 0.99625, achieved using 4 principal components.
- Lower dimensions retain high-dimensional information effectively.
- The classifier is expected to perform well given the retained information.

4.4 Autoencoder Embedding

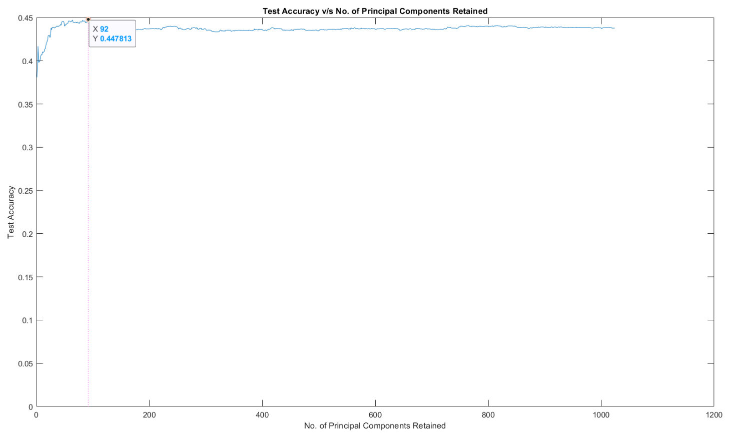

Figure 17: Test Accuracy v/s Number of Principal Components Used for Training

- Maximum accuracy: 0.4478, achieved using 92 principal components.
- Lower dimensions does not retain high-dimensional information effectively.

- The classifier is not expected to perform well given the retained information.

Note: Codes to display the graphs for test accuracy v/s no. of principal components retained can be found in Section 7.6.

5 Results and Conclusions

5.1 Time Taken for performing Dimension Reduction

In this section, we will examine the time required to map the original dataset into the various transformations discussed in the previous sections. This analysis aims to provide insights into the computational complexity of each of these algorithms.

DR Method	Time Taken for DR (s)
PCA	0.78
kPCA (Gaussian)	352.45
kPCA (Polynomial)	344.81
t-SNE	892.39
Autoencoder	24660.55

Table 2: Time Taken for Dimensionality Reduction (DR) Methods

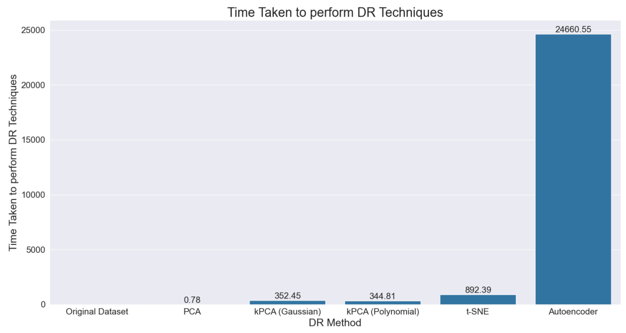

Figure 18: Time Taken for performing DR techniques

1. **Principal Component Analysis (PCA)**: Among the techniques considered, PCA is the fastest method, completing the transformation in 0.78 seconds. Its efficiency stems from directly computing the principal components of the data matrix, resulting in a low computational burden. Its complexity is generally $O(n^2 \cdot d + d \cdot n^2)$, where n is the number of samples and d is the number of features.

2. **Kernel Principal Component Analysis (kPCA)**:

   - **Gaussian kPCA**: This method demands considerably more time, taking 352.45 seconds for transformation. The computation of pairwise distances necessary for the Gaussian kernel contributes to its higher computational complexity, which is $O(n^2 \cdot d + n^3)$, where n is the number of samples and d is the number of features.

   - **Polynomial kPCA**: Similar to Gaussian kPCA, the polynomial kernel completes transformation in 344.81 seconds. Its computational complexity is $O(n^2 \cdot d + d^3)$, where n is the number of samples and d is the number of features.

3. **t-distributed Stochastic Neighbor Embedding (t-SNE)**: t-SNE consumes 892.39 seconds for transformation. Its computational intensity primarily arises from the computation of probabilities of pairwise similarities, leading to a complexity of $O(n^2)$, where n is the number of samples.

4. **Autoencoder Embedding**: Autoencoder embedding is the slowest technique of transformation taking 24660.55 seconds (approximately 6.85 hours). The training of a neural network for autoencoder inherently demands more time compared to linear or kernel-based methods due to its complex architecture and iterative optimization process. Its complexity depends on the architecture and complexity of the neural network, typically $O(n \cdot e)$, where n is the number of samples and e is the number of epochs.

5.2 Optimal Number of Principal Components

In this section, we will study the outcomes of the exercise conducted in 4.

DR Method	Principal Components Used
Original Dataset	N/A
PCA	19
kPCA (Gaussian)	11
kPCA (Polynomial)	61
t-SNE	4
Autoencoder	92

Table 3: Principal Components Used by DR Methods

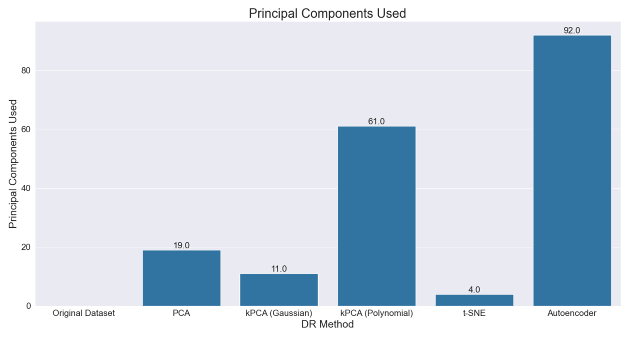

Figure 19: Principal Components used by DR Methods

From the table and graph, it's evident that t-SNE requires the fewest number of components to effectively capture the information from higher dimensions in lower dimensions.

5.3 Analyzing Accuracy and Time Taken for Classification

After determining the optimal number of principal components required for each of the different embedding methods, I conducted KNN classification by retaining these optimal components and compared the classifier's performance. To mitigate the affect of the initial position of centers, I executed the classifier 10 times and averaged the metrics. The graph below illustrates the testing and validation accuracy of the KNN classifier after retaining the optimal number of principal components for each DR method. For all DR methods, the parameters of the KNN classifier is as follows:

- Training Samples: 400 per class
- Testing Samples: 200 per class
- Validation Samples: 25 per class
- $K = 16$
- Distance metric: Euclidean

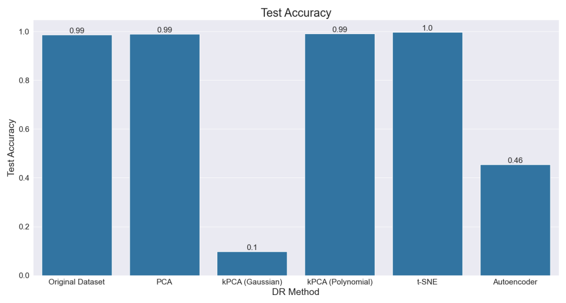

Figure 20: Test Accuracy of DR Methods

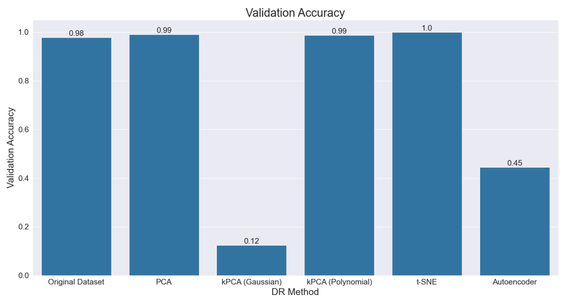

Figure 21: Validation Accuracy of DR Methods

Figure 22: Accuracy of DR Methods after performing KNN Classification

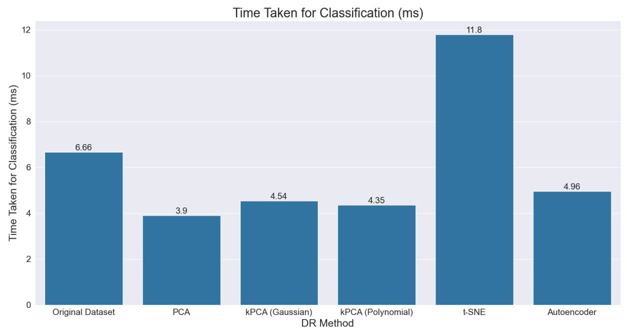

Figure 23: Time Taken for Classification (milliseconds)

Based on the graphs in the Figure 22, the Gaussian kernel Principal Component Analysis (kPCA) and Autoencoder techniques yielded sub-optimal results, with average testing accuracies of 0.1 and 0.46 respectively across 10 simulations. Consequently, these methods are deemed unsuitable for further consideration due to their inadequate performance metrics.

The original dataset, linear PCA, polynomial kPCA, and t-SNE transformations exhibit impressive average testing accuracy, ranging between 0.98 and 1.00. However, to identify the most optimal DR technique, we refer to the Figure 23. Analysis of classification time highlights the superior performance of linear PCA and polynomial kPCA, with classification times of 3.9 ms and 4.35 ms respectively, outperforming other transformations.

Further refinement involves discarding the original representation and t-SNE transformation due to their prolonged classification times. Consequently, the focus narrows down to the choice between linear PCA and polynomial kPCA, which exhibit comparable accuracies and classification times.

Referring to the analysis conducted in section 5.1, it is observed that linear PCA offers significantly faster data transformation, requiring 0.78 seconds compared to polynomial kPCA's notably longer duration of 344.81 seconds (5.75 minutes). This substantial discrepancy in transformation time renders polynomial kPCA impractical for the dataset at hand.

Thus, considering both accuracy and computational efficiency, linear PCA emerges as the preferred DR technique for transforming the dataset. Its superior performance in terms of both accuracy and computational speed makes linear PCA as the optimal choice.

Note: Code to perform KNN classification on transformed data can be found in Section 7.7

6 Conclusion

Throughout this project, I explored various data representation techniques, namely linear PCA, kernel PCA, t-SNE, and Autoencoder. Focusing on the KNN classifier due to its superior accuracy and efficiency, I determined the optimal number of principal components required for each transformation, aiming for efficient representation without compromising accuracy.

While preserving the identified optimal principal components, I compared the performances of the KNN classifier across various transformations. This evaluation encompassed testing accuracy, validation accuracy, classification time, and transformation time, prioritized in that sequence.

From my analysis, I conclude that the linear PCA is the most viable data transformation technique. This conclusion is substantiated by several reasons:

1. Exceptionally high testing and validation accuracy, both at 0.99.
2. Minimal time required for classification, with a processing time of 3.9 milliseconds.
3. Transformation of data completed in 0.78 seconds.

The outcomes of this project closely align with the findings layed out in the article by L.J.P. van der Maaten, E.O. Postma, and H.J. van den Herik, titled "Dimensionality Reduction: A Comparative Review," Tilburg University Technical Report, TiCC-TR 2009-005, 2009. In that article, the authors found that despite their large variance, nonlinear techniques for dimensionality reduction often fail to outperform traditional linear techniques such as PCA.

7 Code

Note: Please ensure that all codes files, `*.mat` files and the DR Toolbox are in the same directory before proceeding with testing.

7.1 Preprocessing Codes

trim.py

```python
import os
import shutil
import random
from tqdm import tqdm

# function to reduce the number of samples in each class
def copyFiles(sourceFolder, destinationFolder):
    if not os.path.exists(destinationFolder):
        os.makedirs(destinationFolder)

    # randomly picks 625 samples from each subdirectory and copies it to destination folder
    files = os.listdir(sourceFolder)
    filesToCopy = random.sample(files, 625)

    for fileName in filesToCopy:
        sourcePath = os.path.join(sourceFolder, fileName)
        destinationPath = os.path.join(destinationFolder, fileName)
        shutil.copyfile(sourcePath, destinationPath)

# path to the folders
sourceFolder = "bigData"
destinationFolder = "data"

if not os.path.exists(destinationFolder):
    os.mkdir(destinationFolder)

classes = os.listdir(sourceFolder)

for c in tqdm(classes):
    sourceClass = os.path.join(sourceFolder, c)
    destinationClass = os.path.join(destinationFolder, c)
    copyFiles(sourceClass, destinationClass)
```

saveGrayToMat.m

```matlab
function saveGrayToMat(folder_path, dim)
    % Recursively iterates through a folder containing RGB images organized
    % in subfolders corresponding to their respective labels. Converts the RGB
    % images into grayscale, resizes them from 100x100 to dim x dim, and stores
    % them in a matrix of shape 16x625x(dim*dim). The resulting matrix is saved as
    % 'gray.mat' in the current working directory.
    %
    % Inputs:
    %   - folder_path: A string specifying the path to the root folder
    %                  containing the subfolders of RGB images.
    %   - dim: Dimension to resize the images to.
    %
    % Example:
    %   saveGrayToMat('path/to/root/folder', 32);
    %
    % Note:
    %   - This function assumes that the subfolders in the provided directory
    %     represent the labels, and each contains RGB images to be processed.
    %   - Ensure that the root folder contains only subfolders representing
    %     labels and no other files or folders, as the function processes all
    %     images found in these subfolders.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024
    fprintf("Detecting labels...\n")
    subfolders = dir(folder_path);

    subfolders = subfolders(arrayfun(@(x) x.name(1), subfolders) ~= '.');

    data = zeros(length(subfolders), 625, dim*dim);
    labels = cell(1, length(subfolders));

    fprintf("Iterating through labels...\n")
    for i = 1:length(subfolders)
        subfolder_name = subfolders(i).name;
        image_files = dir(fullfile(folder_path, subfolder_name, '*.jpg'));
        selected_indices = randperm(length(image_files), 625);
        for j = 1:625
            img = imread(fullfile(folder_path, subfolder_name, image_files(selected_indices(j)).name));
            img = imresize(img, [dim, dim]);
            if size(img, 3) == 3
                img = rgb2gray(img);
            end

            img_vector = img(:)';

            data(i, j, :) = img_vector;
        end

        labels{i} = subfolder_name;
    end

    fprintf("Saving data to .mat file...\n");
    save("gray.mat", 'data', 'labels');
end
```

7.2 Transformation Codes

7.2.1 transformation.m

```
clear;
clc;
addpath("drtoolbox")
addpath("drtoolbox/techniques")
load('gray.mat')

% Perform PCA and save transformation as matrix
PCAandSave(data, labels, 32);

% Perform Polynomial kPCA and save transformation as matrix
polyPCAandSave(data, labels, 32);

% Perform Gaussian kPCA and save transformation as matrix
gaussPCAandSave(data, labels, 32);

% Perform t-SNE and save transformation as matrix
TSNEandSave(data, labels, 32);

% Perform autoencoder embedding and save transformation as matrix
AutoEncoderandSave(data, labels, 32);
```

7.2.2 PCAandSave.m

```
function PCAandSave(data, labels, dim)
    % Performs Principal Component Analysis (PCA) on the input data matrix
    % and saves the result along with the corresponding labels. The PCA
    % dimensionality reduction is applied to reduce the data to a specified
    % dimension (dim x dim).
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of samples, N is the number of features per sample, and P
    %           is the number of dimensions per feature.
    %   - labels: Cell array containing labels corresponding to each sample
    %             in the data matrix.
    %   - dim: Dimensionality to which the data will be reduced using PCA.
    %
    % Example:
    %   PCAandSave(data, labels, 32);
    %
    % Note:
    %   - This function performs PCA on the input data matrix using the
    %     'PCA' method provided by the 'compute_mapping' function.
    %   - The resulting PCA-transformed data matrix is saved as 'pcaData.mat'
    %     along with the corresponding labels.
    %   - The input data matrix 'data' must be double type for accurate PCA
    %     computation.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    data = double(data);
    tic;
    data_reshaped = reshapeForPCA(data);
    pcaData = compute_mapping(data_reshaped, 'PCA', dim*dim);
    elapsed = toc;
    pcaData = reshape(pcaData, size(data));
    pcaData = real(pcaData);
    save("pcaData.mat", "pcaData", "labels");
    fprintf("Time Taken for DR: %f seconds\n", elapsed);
end
```

7.2.3 gaussPCAandSave.m

```
function gaussPCAandSave(data, labels, dim)
    % Performs kernel Principal Component Analysis (kPCA) on the input
    % data matrix and saves the result along with the corresponding labels.
    % The kPCA dimensionality reduction is applied to reduce the data to a
    % specified dimension (dim x dim) using a Gaussian kernel.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of samples, N is the number of features per sample, and P
    %           is the number of dimensions per feature.
    %   - labels: Cell array containing labels corresponding to each sample
    %             in the data matrix.
    %   - dim: Dimensionality to which the data will be reduced using kPCA.
    %
    % Example:
    %   gaussPCAandSave(data, labels, 32);
    %
    % Note:
    %   - This function performs kPCA on the input data matrix using the
    %     Gaussian kernel.
    %   - The resulting kPCA-transformed data matrix is saved as 'gauss.mat'
    %     along with the corresponding labels.
    %   - The input data matrix 'data' must be preprocessed and reshaped
    %     appropriately for kPCA.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    tic;
    data_reshaped = reshapeForPCA(data);
    kpcaData = kernel_pca(data_reshaped, dim*dim, 'gauss');
    elapsed = toc;
    kpcaData = reshape(kpcaData, size(data));
    kpcaData = real(kpcaData);
    save("gauss.mat", "kpcaData", "labels");
    fprintf("Time Taken for DR: %f seconds\n", elapsed);
end
```

7.2.4 polyPCAandSave.m

```
function polyPCAandSave(data, labels, dim)
    % Performs kernel Principal Component Analysis (kPCA) on the input
    % data matrix and saves the result along with the corresponding labels.
    % The kPCA dimensionality reduction is applied to reduce the data to a
    % specified dimension (dim x dim) using a polynomial kernel.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of samples, N is the number of features per sample, and P
    %           is the number of dimensions per feature.
    %   - labels: Cell array containing labels corresponding to each sample
    %             in the data matrix.
    %   - dim: Dimensionality to which the data will be reduced using kPCA.
    %
    % Example:
    %   polyPCAandSave(data, labels, 32);
    %
    % Note:
    %   - This function performs kPCA on the input data matrix using the
    %     polynomial kernel.
    %   - The resulting kPCA-transformed data matrix is saved as 'poly.mat'
    %     along with the corresponding labels.
    %   - The input data matrix 'data' must be preprocessed and reshaped
    %     appropriately for kPCA.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    tic;
    data_reshaped = reshapeForPCA(data);
    kpcaData = kernel_pca(data_reshaped, dim*dim, 'poly');
    elapsed = toc;
    kpcaData = reshape(kpcaData, size(data));
    kpcaData = real(kpcaData);
    save("poly.mat", "kpcaData", "labels");
    fprintf("Time Taken for DR: %f seconds\n", elapsed);
end
```

7.2.5 TSNEandSave.m

```
function TSNEandSave(data, labels, dim)
    % Performs t-Distributed Stochastic Neighbor Embedding (t-SNE) on the
    % input data matrix and saves the result along with the corresponding
    % labels. The t-SNE dimensionality reduction is applied to reduce the
    % data to a specified dimension (dim x dim).
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of samples, N is the number of features per sample, and P
    %           is the number of dimensions per feature.
    %   - labels: Cell array containing labels corresponding to each sample
    %             in the data matrix.
    %   - dim: Dimensionality to which the data will be reduced using t-SNE.
    %
    % Example:
    %   TSNEandSave(data, labels, 32);
    %
    % Note:
    %   - This function performs t-SNE on the input data matrix using the
    %     'tSNE' method provided by the 'compute_mapping' function.
    %   - The resulting t-SNE-transformed data matrix is saved as
    %     'tsneData.mat' along with the corresponding labels.
    %   - The input data matrix 'data' must be double type for accurate t-SNE
    %     computation.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    tic;
    data = double(data);
    data_reshaped = reshapeForPCA(data);
    tsneData = compute_mapping(data_reshaped, "tSNE", dim*dim);
    tsneData = reshape(tsneData, size(data));
    save("tsneData.mat", "tsneData", "labels");
    elapsed = toc;
    fprintf("Time Taken for DR: %f seconds\n", elapsed);
end
```

7.2.6 AutoEncoderandSave.m

```
function AutoEncoderandSave(data, labels, dim)
    % Applies an autoencoder for dimensionality reduction (DR) on the input
    % data matrix and saves the result along with the corresponding labels.
    % The autoencoder dimensionality reduction is applied to reduce the data
    % to a specified dimension (dim x dim).
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of samples, N is the number of features per sample, and P
    %           is the number of dimensions per feature.
    %   - labels: Cell array containing labels corresponding to each sample
    %             in the data matrix.
    %   - dim: Dimensionality to which the data will be reduced using the
    %          autoencoder.
    %
    % Example:
    %   AutoEncoderandSave(data, labels, 32);
    %
    % Note:
    %   - This function applies an autoencoder for dimensionality reduction
    %     on the input data matrix using the 'Autoencoder' method provided
    %     by the 'compute_mapping' function.
    %   - The resulting autoencoder-transformed data matrix is saved as
    %     'autoData.mat' along with the corresponding labels.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    tic;
    data_reshaped = reshapeForPCA(data);
    autoData = compute_mapping(data_reshaped, "Autoencoder", dim*dim);
    elapsed = toc;
    autoData = reshape(autoData, size(data));
    autoData = real(autoData);
    save("autoData.mat", "autoData", "labels");
    fprintf("Time Taken for DR: %f seconds\n", elapsed);
end
```

7.3 View Samples Post Transformation

sampleViewer.m

```
clc;
clear;


load("gray.mat")
plot4x4(data, labels, 1, 32);

load("pcaData.mat")
plot4x4(pcaData, labels, 1, 32);

load("gauss.mat")
plot4x4(kpcaData, labels, 1, 32);

load("poly.mat")
plot4x4(kpcaData, labels, 1, 32);

load("tsneData.mat")
plot4x4(tsneData, labels, 1, 32);

load("autoData.mat")
plot4x4(autoData, labels, 1, 32);



function plot4x4(data, labels, sample, dim)
    % Plots a 4x4 grid of images from the input data matrix along with
    % their corresponding labels. Each row of the grid represents a label
    % and displays the specified sample image resized to the specified
    % dimensions.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of classes/labels, N is the number of samples per class, and
    %           P is the number of dimensions per sample.
    %   - labels: Cell array containing labels corresponding to each class
    %             in the data matrix.
    %   - sample: Index of the sample to display for each class.
    %   - dim: Dimension of the images (assumed to be square).
    %
    % Example:
    %   plot4x4(data, labels, 1, 32);
    %
    % Note:
    %   - This function assumes that the data matrix contains images
    %     organized such that each row corresponds to a class, and each
    %     column corresponds to a sample within that class.
    %   - The specified sample from each class is resized to the specified
    %     dimensions and displayed in a 4x4 grid.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024
    figure;
    for i=1:numel(labels)
        subplot(4, 4, i)
        colormap('gray')
        image = data(i, sample, :);
        image = reshape(image, [dim, dim]);
        imagesc(image);
        axis off;
        title(labels{i});
    end
end
```

7.4 2D Representation of Transformed Data

twoDimRep.m

```
clc;
clear;

load("gray.mat")
twoDimRepresentation(data, labels);

load("pcaData.mat")
twoDimRepresentation(pcaData, labels);

load("gauss.mat")
twoDimRepresentation(kpcaData, labels);

load("poly.mat")
twoDimRepresentation(kpcaData, labels);

load("tsneData.mat")
twoDimRepresentation(tsneData, labels);

load("autoData.mat")
twoDimRepresentation(autoData, labels);


function twoDimRepresentation(data, labels)
    % Plots a two-dimensional representation of the input data using the
    % first two components obtained from dimensionality reduction (DR).
    % Each class is represented by a distinct color in the plot, and the
    % legend indicates the corresponding labels.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of classes/labels, N is the number of samples per class, and
    %           P is the number of dimensions per sample.
    %   - labels: Cell array containing labels corresponding to each class
    %             in the data matrix.
    %
    % Example:
    %   twoDimRepresentation(data, labels);
    %
    % Note:
    %   - This function assumes that the input data has been reduced to two
    %     dimensions using dimensionality reduction techniques such as PCA,
    %     t-SNE, or autoencoder.
    %   - Each class in the data is represented by a distinct color, and
    %     the legend indicates the corresponding labels.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    % Extracting only the first two dimensions of the data
    data = data(:, :, 1:2);

    % Reshape data for plotting
    data = reshapeForPCA(data);

    % Determine the number of unique labels
    num_labels = numel(unique(labels));

    % Generate a colormap with distinct colors for each label
    color_map = jet(num_labels);

    % Plotting
    figure;
    hold on;
    for i = 1:num_labels
        idx = (i - 1) * 625 + 1 : i * 625;
        scatter(data(idx, 1), data(idx, 2), 20, color_map(i, :), 'o');
    end

    % Adding legend
    legend(labels, "Location","eastoutside");
    hold off;

    % Title and axis labels
    title('Data Representation');
    xlabel('First Component');
    ylabel('Second Component');
end
```

7.5 Classification Algorithms

7.5.1 Comparing Classification Algorithms

compareAlgos.m

```
clc;
clear;
load("gray.mat")

% Splitting data into train, test and validatation datasets
[train, test] = randomTrainTestSplit(data, 400);
[test, valid] = randomTrainTestSplit(test, 200);

% Performing KNN Classification
[testAcc, validAcc, execTime, ~] = KNNClassification(train, test, valid, 16);
fprintf("KNN Classification Results...\nTest Accuracy: %f\nValidation Accuracy: %f\nTime Taken for Classification: %f seconds\n", testAcc, validAcc, execTime);

% Performing kSVM Classification
[testAcc, validAcc, execTime, ~] = kSVMClassification(train, test, valid);
fprintf("kSVM Classification Results...\nTest Accuracy: %f\nValidation Accuracy: %f\nTime Taken for Classification: %f seconds\n", testAcc, validAcc, execTime);
```

7.5.2 KNN Classification

KNNClassification.m

```
function [testAccuracy, validAccuracy, executionTime, perClassAccuracy] = KNNClassification(trainData, testData, validData, k)
    % KNNClassification(trainData, testData, validData, k)
```

```
%
% Performs k-Nearest Neighbors (KNN) classification on the training
% dataset and evaluates the model on both testing and validation
% datasets. Returns the test accuracy, validation accuracy, execution
% time, and per-class accuracy.
%
% Inputs:
%   - trainData: Training data matrix of size C x M x N, where C is the
%                number of classes, M is the number of samples per class,
%                and N is the number of features per sample.
%   - testData: Testing data matrix of size C x P x Q, where C is the
%               number of classes, P is the number of samples per class,
%               and Q is the number of features per sample.
%   - validData: Validation data matrix of size C x R x S, where C is
%                the number of classes, R is the number of samples per
%                class, and S is the number of features per sample.
%   - k: Number of neighbors to consider in the KNN algorithm.
%
% Outputs:
%   - testAccuracy: Accuracy of the KNN model on the testing dataset.
%   - validAccuracy: Accuracy of the KNN model on the validation dataset.
%   - executionTime: Time taken to train the KNN model in seconds.
%   - perClassAccuracy: Per-class accuracy of the KNN model on the
%                       testing dataset, represented as a vector of size
%                       1 x C, where C is the number of classes.
%
% Example:
%   [testAccuracy, validAccuracy, executionTime, perClassAccuracy] = KNNClassification(trainData, testData, validData, 5);
%
% Note:
%   - This function uses the fitcknn function from MATLAB's Statistics
%     and Machine Learning Toolbox to perform KNN classification.
%
% Author: Nischal Chandur
% Date: 05/15/2024

% Convert input data to double
trainData = double(trainData);
testData = double(testData);
validData = double(validData);

% Get sizes of datasets
trainSamples = size(trainData, 2);
testSamples = size(testData, 2);
validSamples = size(validData, 2);
components = size(trainData, 3);
classes = size(trainData, 1);

% Reshape data matrices
trainFlat = reshape(trainData, [classes*trainSamples, components]);
testFlat = reshape(testData, [classes*testSamples, components]);
validFlat = reshape(validData, [classes*validSamples, components]);

% Extracting labels
trainLabels = repmat((1:classes)', trainSamples, 1);
testLabels = repmat((1:classes)', testSamples, 1);
validLabels = repmat((1:classes)', validSamples, 1);

% Start clock
tic;

% Performing KNN Classification
model = fitcknn(trainFlat, trainLabels, 'NumNeighbors', k, 'Distance','euclidean');

% Stop clock
```

```
    executionTime = toc;

    % Making predictions on the testing and validation datasets
    testPredictions = predict(model, testFlat);
    validPredictions = predict(model, validFlat);

    % Measuring accuracy of model on test data
    testAccuracy = sum(testLabels == testPredictions) / numel(testLabels);

    % Measuring accuracy of model on validation data
    validAccuracy = sum(validLabels == validPredictions) / numel(validLabels);

    % Computing per-class accuracy
    perClassAccuracy = zeros(1, classes);
    for i = 1:classes
        classIndices = testLabels == i;
        classAccuracy = sum(testPredictions(classIndices) == i) / sum(classIndices);
        perClassAccuracy(i) = classAccuracy;
    end
end
```

7.5.3 kSVM Classification

kSVMClassification.m

```
function [testAccuracy, validAccuracy, executionTime, perClassAccuracy] = kSVMClassification(trainData, testData, validData)
    % kSVMClassification(trainData, testData, validData)
    %
    % Performs multi-class Support Vector Machine (SVM) classification on
    % the training dataset and evaluates the model on both testing and
    % validation datasets. Returns the test accuracy, validation accuracy,
    % execution time, and per-class accuracy.
    %
    % Inputs:
    %   - trainData: Training data matrix of size C x M x N, where C is the
    %                number of classes, M is the number of samples per class,
    %                and N is the number of features per sample.
    %   - testData: Testing data matrix of size C x P x Q, where C is the
    %               number of classes, P is the number of samples per class,
    %               and Q is the number of features per sample.
    %   - validData: Validation data matrix of size C x R x S, where C is
    %                the number of classes, R is the number of samples per
    %                class, and S is the number of features per sample.
    %
    % Outputs:
    %   - testAccuracy: Accuracy of the SVM model on the testing dataset.
    %   - validAccuracy: Accuracy of the SVM model on the validation dataset.
    %   - executionTime: Time taken to train the SVM model in seconds.
    %   - perClassAccuracy: Per-class accuracy of the SVM model on the
    %                       testing dataset, represented as a vector of size
    %                       1 x C, where C is the number of classes.
    %
    % Example:
    %   [testAccuracy, validAccuracy, executionTime, perClassAccuracy] = kSVMClassification(trainData, testData, validData);
    %
    % Note:
    %   - This function uses the fitcecoc function from MATLAB's Statistics
    %     and Machine Learning Toolbox to perform multi-class SVM
    %     classification.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024
    % Convert input data to double
    trainData = double(trainData);
    testData = double(testData);
    validData = double(validData);

    % Get sizes of datasets
    trainSamples = size(trainData, 2);
    testSamples = size(testData, 2);
    validSamples = size(validData, 2);
    components = size(trainData, 3);
    classes = size(trainData, 1);

    % Reshape data matrices
    trainFlat = reshape(trainData, [classes*trainSamples, components]);
    testFlat = reshape(testData, [classes*testSamples, components]);
    validFlat = reshape(validData, [classes*validSamples, components]);

    % Extracting labels
    trainLabels = repmat((1:classes)', trainSamples, 1);
    testLabels = repmat((1:classes)', testSamples, 1);
    validLabels = repmat((1:classes)', validSamples, 1);

    % Start clock
    tic;

    % Performing SVM Classification
    model = fitcecoc(trainFlat, trainLabels);

    % Stop clock
    executionTime = toc;

    % Making predictions on the testing and validation datasets
    testPredictions = predict(model, testFlat);
    validPredictions = predict(model, validFlat);

    % Measuring accuracy of model on test data
    testAccuracy = sum(testLabels == testPredictions) / numel(testLabels);

    % Measuring accuracy of model on validation data
    validAccuracy = sum(validLabels == validPredictions) / numel(validLabels);

    % Computing per-class accuracy
    perClassAccuracy = zeros(1, classes);
    for i = 1:classes
        classIndices = testLabels == i;
        classAccuracy = sum(testPredictions(classIndices) == i) / sum(classIndices);
        perClassAccuracy(i) = classAccuracy;
    end
end
```

7.6 Optimum Principal Components

retainPC.m

```
clc;
clear;

disp("Analyzing PCA Dataset...");
load("pcaData.mat")
PCvsAcc(pcaData);

disp("Analyzing Gaussian kPCA Dataset...");
```

```matlab
load("gauss.mat")
PCvsAcc(kpcaData);

disp("Analyzing Polynomial kPCA Dataset...");
load("poly.mat")
PCvsAcc(kpcaData);

disp("Analyzing t-SNE Dataset...");
load("tsneData.mat")
PCvsAcc(tsneData)

disp("Analyzing AutoEncoder Dataset...");
load("autoData.mat")
PCvsAcc(autoData);



function PCvsAcc(data)
    % Computes the test accuracy of a KNN classifier with varying numbers
    % of principal components retained from the input data. Plots the test
    % accuracy against the number of principal components retained.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of classes, N is the number of samples per class, and P is
    %           the number of features per sample.
    %
    % Example:
    %   PCvsAcc(data);
    %
    % Note:
    %   - This function uses the KNNClassification function to compute the
    %     test accuracy of a KNN classifier with varying numbers of
    %     principal components retained.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    % Initialize arrays to store accuracies and indices
    accs = zeros(1, size(data, 3));
    indices = zeros(1, size(data, 3));

    % Get the number of components in the input data
    components = size(data, 3);

    % Split the data into training, testing, and validation sets
    [train, test] = randomTrainTestSplit(data, 400);
    [test, valid] = randomTrainTestSplit(test, 200);

    % Loop over different numbers of principal components
    for i = 1:components
        if mod(i, 100) == 0
            fprintf("Iteration %d\n", i);
        end

        % Compute test accuracy using KNN classifier
        acc = KNNClassification(train(:, :, 1:i), test(:, :, 1:i), valid(:, :, 1:i), 16);

        % Store accuracy and index
        indices(i) = i;
        accs(i) = acc;
    end

    % Plot test accuracy vs. number of principal components retained
    figure;
    [acc_max, index] = max(accs);
    idx_max = indices(index);
    plot(indices, accs, idx_max, acc_max, 'ro');
    xlabel("No. of Principal Components Retained");
    ylabel("Test Accuracy")
    title("Test Accuracy v/s No. of Principal Components Retained");
    hold on
    plot([idx_max idx_max],[0 acc_max],':m');
    hold off
end
```

7.7 KNN Classification on Transformed Data

compareTechniques.m

```
clc;
clear;

disp("Performing KNN on Original Dataset...")
load("gray.mat")
KNNnTimes(data, labels, 10);

disp("Performing KNN on PCA Dataset...")
load("pcaData.mat")
KNNnTimes(pcaData(:, :, 1:19), labels, 10);

disp("Performing KNN on Gaussian kPCA Dataset...")
load("gauss.mat")
KNNnTimes(kpcaData(:, :, 1:11), labels, 10);

disp("Performing KNN on Polynomial kPCA Dataset...")
load("poly.mat")
KNNnTimes(kpcaData(:, :, 1:61), labels, 10);

disp("Performing KNN on t-SNE Dataset...")
load("tsneData.mat")
KNNnTimes(tsneData(:, :, 1:4), labels, 10);

disp("Performing KNN on Autoencoder Dataset...")
load("autoData.mat")
KNNnTimes(autoData(:, :, 1:92), labels, 10);
```

KNNnTimes.m

```
function KNNnTimes(data, labels, n)
    % Performs KNN classification multiple times and computes the average
    % test accuracy, validation accuracy, execution time, and per-class
    % test accuracy over the specified number of simulations.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of classes, N is the number of samples per class, and P is
    %           the number of features per sample.
    %   - labels: Cell array containing labels corresponding to each class
    %             in the data matrix.
    %   - n: Number of simulations to run.
    %
    % Example:
    %   KNNnTimes(data, labels, 10);
    %
    % Note:
```

```
    %   - This function uses the KNNClassificationV2 function to perform
    %     KNN classification and computes the average test accuracy,
    %     validation accuracy, execution time, and per-class test accuracy
    %     over multiple simulations.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    % Initialize arrays to store results
    testAcc = zeros(1, n);
    validAcc = zeros(1, n);
    time = zeros(1, n);
    perClassTestAcc = zeros(size(data, 1), n);

    % Split the data into training, testing, and validation sets
    [train, test] = randomTrainTestSplit(data, 400);
    [test, valid] = randomTrainTestSplit(test, 200);

    % Run KNN classification multiple times
    for i = 1:n
        [ta, va, t, perClass] = KNNClassificationV2(train, test, valid, 16);
        testAcc(i) = ta;
        validAcc(i) = va;
        time(i) = t;
        perClassTestAcc(:, i) = perClass;
    end

    % Display results after multiple simulations
    fprintf('Results After %d simulations\n', n);
    fprintf('Test Accuracy: %f\nValid Accuracy: %f\nTime Taken: %f ms\n', mean(testAcc), mean(validAcc), mean(time) * 1000);

    % Compute and display average per-class test accuracy
    averagePerClass = mean(perClassTestAcc, 2);
    for i = 1:size(data, 1)
        fprintf('Class %s: %.2f\n', labels{i}, averagePerClass(i));
    end
end
```

7.8 Utility Codes

7.8.1 Reshape data

reshapeForPCA.m

```
function data_reshaped = reshapeForPCA(data)

    % Reshapes the input data matrix into a format suitable for Principal
    % Component Analysis (PCA) by converting it into a 2D matrix where each
    % row represents a sample and each column represents a feature.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of classes, N is the number of samples per class, and P is
    %           the number of features per sample.
    %
    % Output:
    %   - data_reshaped: Reshaped data matrix of size (M*N) x P, suitable
    %                    for PCA.
    %
    % Example:
    %   data_reshaped = reshapeForPCA(data);
    %
    % Note:
    %   - This function converts the input data matrix into a 2D matrix
    %     where each row represents a sample and each column represents a
    %     feature, making it suitable for PCA.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    % Reshape the input data matrix
    data_reshaped = reshape(data, [size(data, 1) * size(data, 2), size(data, 3
```

7.8.2 Splitting data into training and testing datasets

randomTrainTestSplit.m

```
function [train, test] = randomTrainTestSplit(data, trainSamples)
    % Splits the input data matrix into training and testing sets using a
    % random permutation of sample indices.
    %
    % Inputs:
    %   - data: Input data matrix of size M x N x P, where M is the number
    %           of classes, N is the total number of samples, and P is the
    %           number of features per sample.
    %   - trainSamples: Number of samples to include in the training set.
    %
    % Outputs:
    %   - train: Training data matrix of size M x trainSamples x P.
    %   - test: Testing data matrix of size M x (N - trainSamples) x P.
    %
    % Example:
    %   [train, test] = randomTrainTestSplit(data, 400);
    %
    % Note:
    %   - This function randomly shuffles the sample indices and splits the
    %     input data matrix into training and testing sets based on the
    %     specified number of training samples.
    %
    % Author: Nischal Chandur
    % Date: 05/15/2024

    % Generate a random permutation of sample indices
    indices = randperm(size(data, 2));

    % Select samples for training and testing based on the random permutation
    train = data(:, indices(1:trainSamples), :);
    test = data(:, indices(trainSamples + 1:end), :);
end
```