CSE 276C Homework 4

Adrian L. Pavlak
U09830551

28. November 2023

1 Question 1

In robotics it is typical to have to recognize objects in the environment. We will here use the German Traffic Sign dataset for recognition of traffic signs. https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-trafficsign to compute subspaces for the PCA and LDA methods. Provide illustration of the respective 1st and 2nd eigenvectors Compute the recognition rates for the test-set. Report

1. Correct classification
2. Incorrect classification

Provide at least one suggestion for how you might improve performance of the system

Answer: This task involves utilizing PCA and LDA to reduce dimensionality and classify German traffic signs. The dataset consists of three components: Meta, Train, and Test. The Meta data includes images of each of the 43 signs, while the Train and Test data consist of 39,209 and 12,630 RGB images, respectively. Initial preprocessing steps are undertaken to standardize the images.

1.1 Preprocessing

The following steps are taken for preprocessing:

1. Extracting images and labels, converting them into NumPy arrays.
2. Grayscaling RGB images using specified coefficients.
3. Resizing images to the smallest size (25x25).
4. Normalizing grayscale values to a range of 0-1.
5. Vectorizing all images and arranging them in a matrix format.

The resulting matrix, X, has dimensions 39209x625.

1.2 Classifier: Random Forest Regressor

The chosen classifier is the Random Forest Regressor due to its suitability for this problem. This method involves creating decision trees based on input features (PCA and LDA components) and aggregating their predictions for the final result.

1.3 Principal Component Analysis (PCA)

The sklearn PCA function is employed for unsupervised dimensionality reduction. The output includes eigenvectors, with the first and second illustrated below:

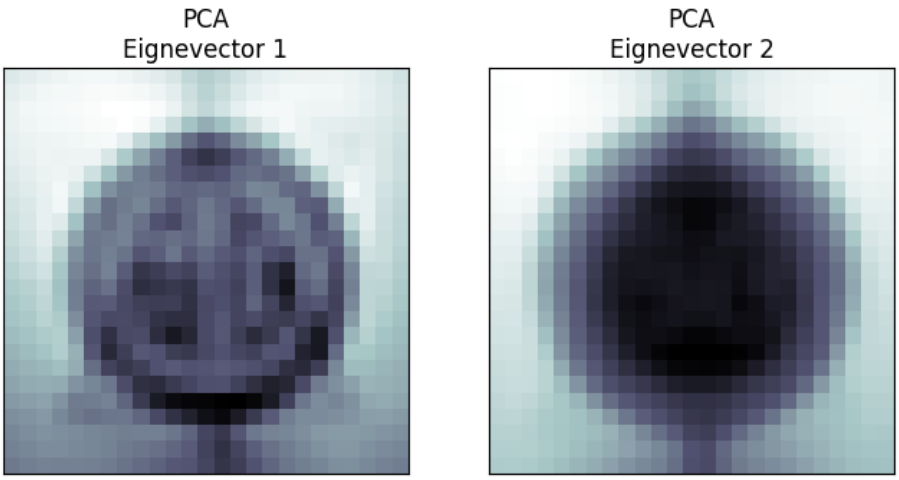


Figure 1: Eigenvectors 1 and 2 for PCA

The cumulative sum of explained variance demonstrates that around 200 dimensions effectively capture the variance.

After implementing the Random Forest Regressor following PCA, the error rate for the 12,630 test images is calculated. Unfortunately, the recognition rate is only 57%, indicating suboptimal performance.

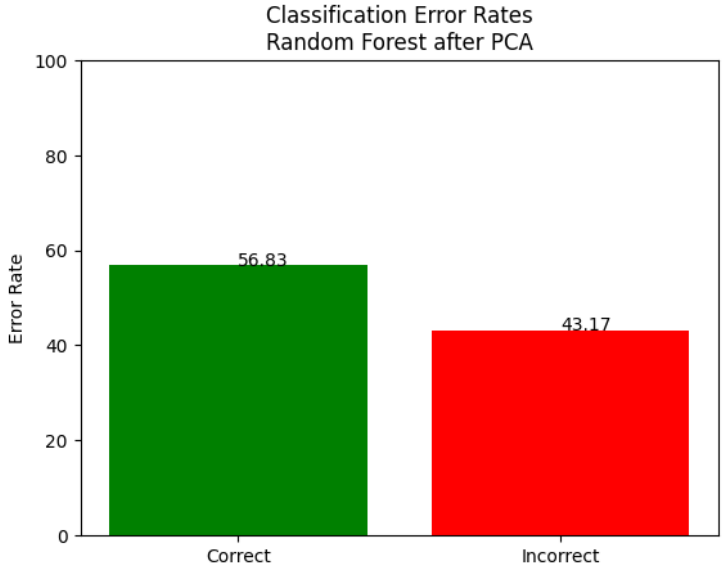


Figure 2: Correct and Incorrect Classification Rates for PCA and Random Forest

1.4 Linear Discriminant Analysis (LDA)

LDA is applied as a supervised algorithm to identify discriminative features. The resulting eigenvectors are as follows:

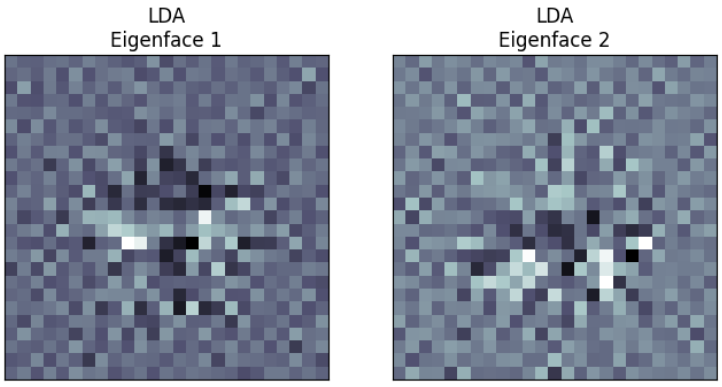


Figure 3: Eigenvectors 1 and 2 for LDA

Post LDA and Random Forest Regressor classification yields a significantly improved recognition rate of 82% for the 12,630 test images.

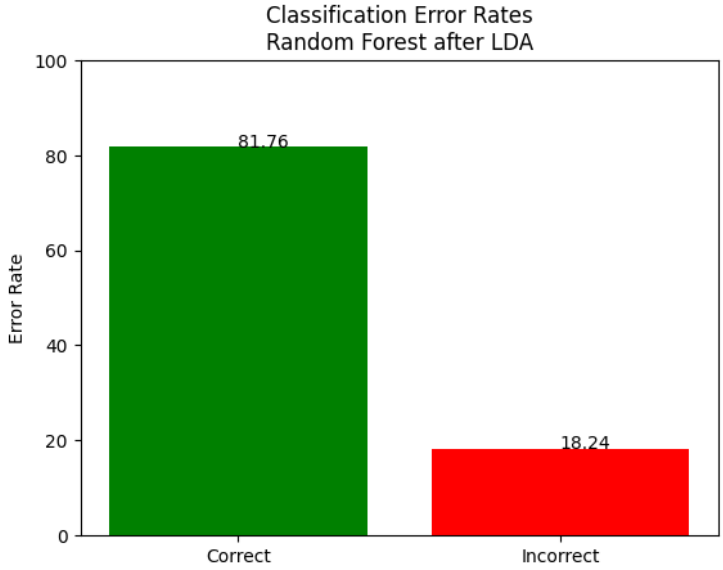


Figure 4: Correct and Incorrect Classification Rates for LDA and Random Forest

1.5 Possible Improvements

Given the discrepancy in performance between PCA and LDA, several improvements are proposed:

- Increase the volume of training data to enhance PCA's ability to represent the features effectively.
- Consider alternative data splits for the test and train sets, potentially incorporating some test images into the training set.
- Explore other classifiers specifically designed for high-dimensional feature classification.
- Fine-tune parameters for both PCA and LDA.

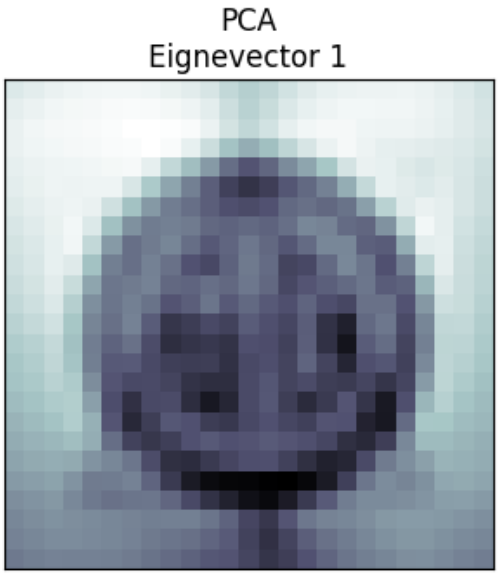
In [ ]:
```
import numpy as np
import csv
import matplotlib.pyplot as plt
from skimage import transform as sk_transform
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

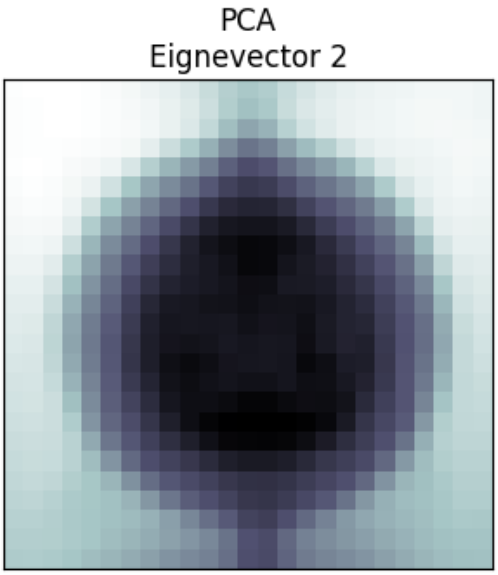
In [ ]:
```
# Load the CSV files and store into numpy arrays
train_data = np.array(read_CSV('Train.csv'))
test_data = np.array(read_CSV('Test.csv'))
meta_data = np.array(read_CSV('Meta.csv'))
```

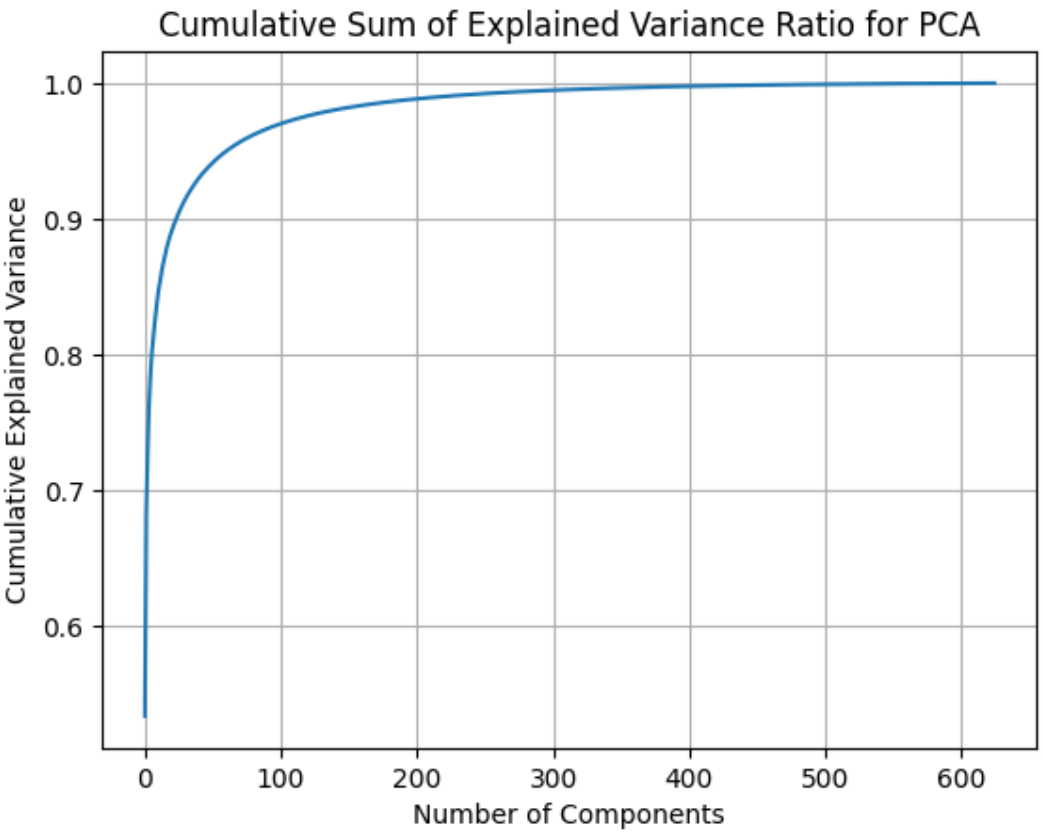
In [ ]:
```
# Get the images and labels
train_img, train_labels = get_images_and_labels(train_data)
test_img, test_labels = get_images_and_labels(test_data)

# Preprocess the training and testing data
train_img_processed = preprocess_images(train_img)
test_img_processed = preprocess_images(test_img)
```

```
C:\Users\adria\AppData\Local\Temp\ipykernel_15976\2647168268.py:6: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  return np.array(gray_images)
C:\Users\adria\AppData\Local\Temp\ipykernel_15976\2647168268.py:13: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  return np.array(norm_images)
```

In [ ]:
```
# Perform PCA on the training images
pca = PCA()
pca.fit(train_img_processed)

# Show the first 2 eigenfaces for PCA
fig = plt.figure(figsize=(8, 6))
for i in range(2):
        ax = fig.add_subplot(1, 2, i + 1, xticks=[], yticks=[])
        ax.imshow(pca.components_[i].reshape((25, 25)), cmap=plt.cm.bone)
        ax.set_title('{}\nEignevector {}'.format('PCA', i + 1))
plt.show()

# Plot cumulative sum of explained variance ratio
plot_cummulative_PCA(pca)
```





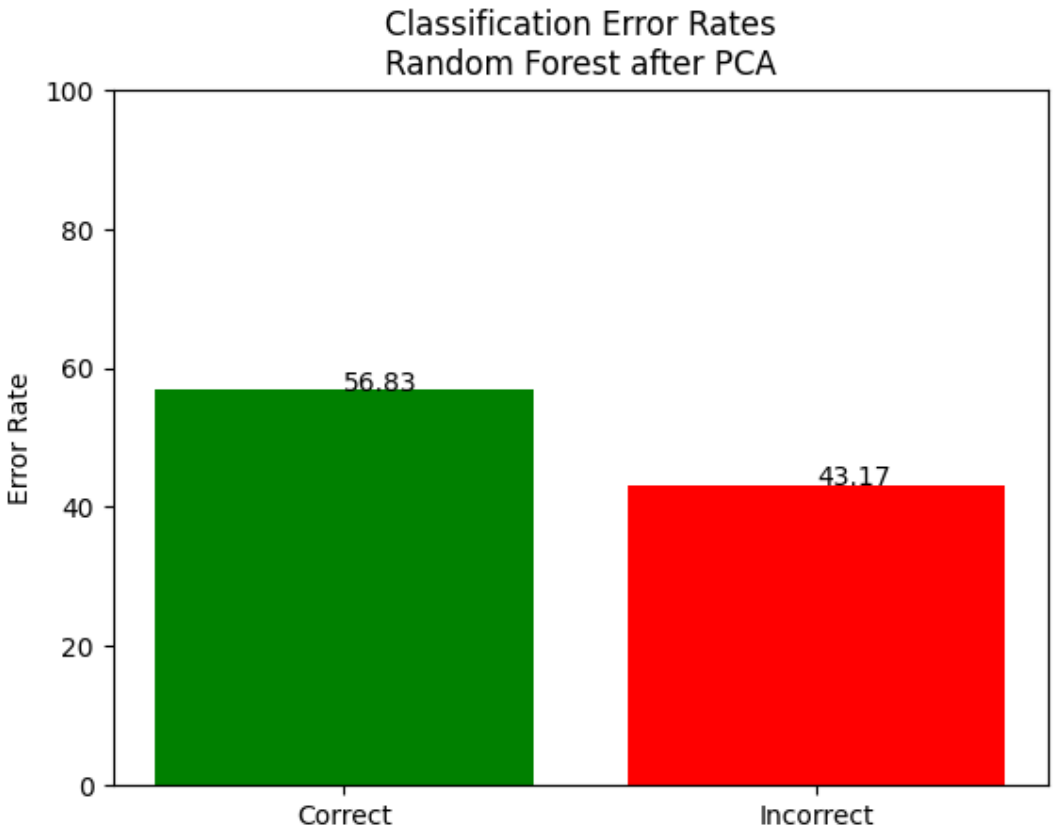


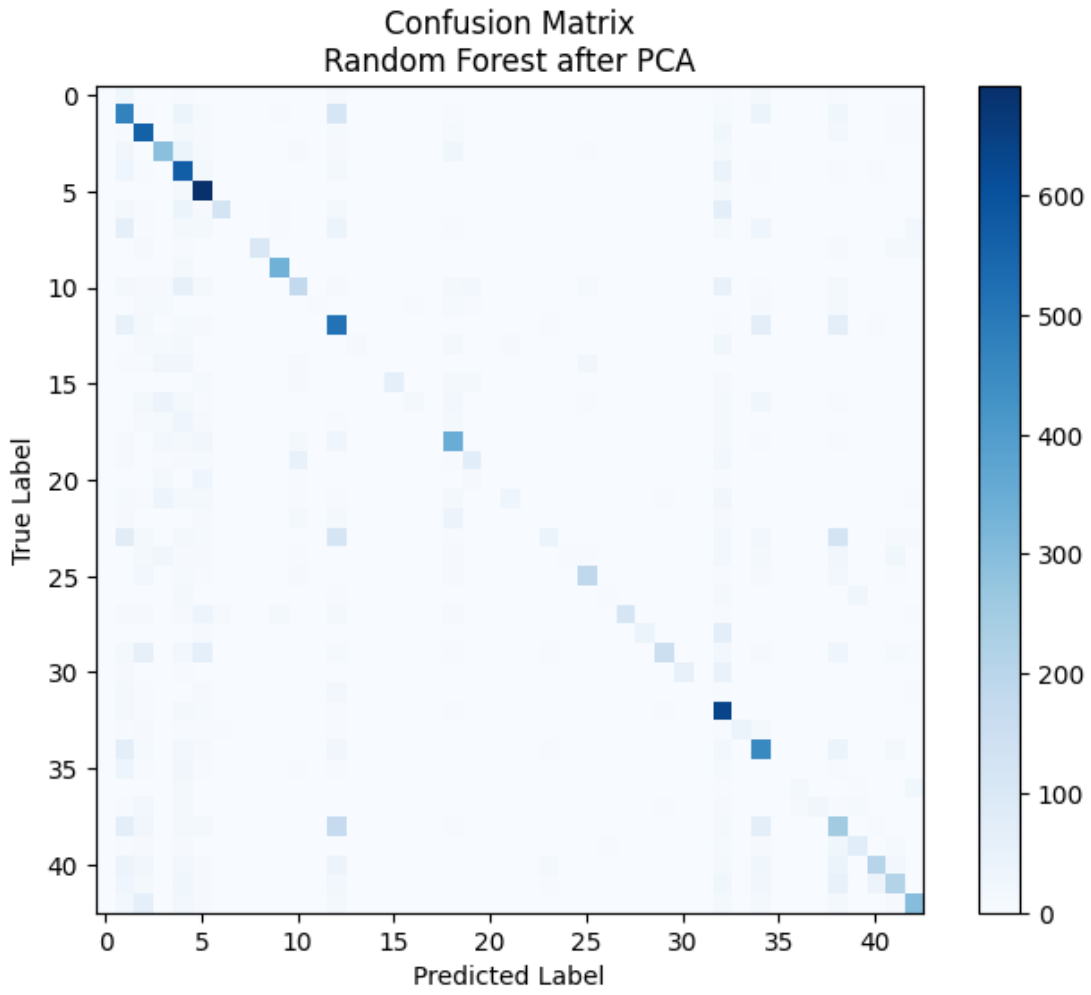
In [ ]:
```python
# Perform Classification using Random Forest after PCA
pca = PCA(n_components=400)
pca.fit(train_img_processed)

# Transform the train and test images using PCA
train_img_pca = pca.transform(train_img_processed)
test_img_pca = pca.transform(test_img_processed)

# Train the Random Forest Classifier
rf = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=0)
rf.fit(train_img_pca, train_labels)
```

```
# Predict the labels for the test images
test_labels_predicted = rf.predict(test_img_pca)
```

In [ ]:
```
# Evaluate the classifier for Random Forest after PCA
evaluate_classifier(test_labels, test_labels_predicted, 'Random Forest after PCA
plot_10_predicted_images(test_img, test_labels, test_labels_predicted, meta_data
```

```
Correct classification rate: 0.5683293745051464
Incorrect classification rate: 0.4316706254948536
```

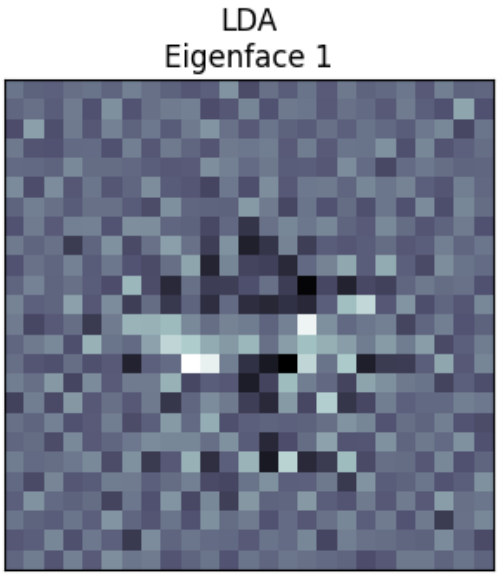


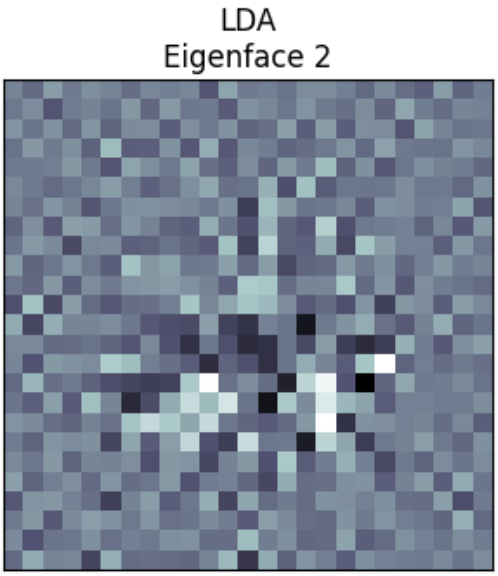



```
C:\Users\adria\AppData\Local\Temp\ipykernel_15976\1045172290.py:131: VisibleDepre
cationWarning: Creating an ndarray from ragged nested sequences (which is a list-
or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is depr
ecated. If you meant to do this, you must specify 'dtype=object' when creating th
e ndarray.
  meta_img = np.array(meta_img)
(43,)
```

In [ ]:
```
# LDA Classification with PCA
# Perform LDA on the PCA-transformed images
lda = sk.discriminant_analysis.LinearDiscriminantAnalysis()
lda.fit(train_img_processed, train_label)

# Plot the first 2 eigenfaces
fig = plt.figure(figsize=(8, 6))
for i in range(2):
    ax = fig.add_subplot(1, 2, i + 1, xticks=[], yticks=[])
    ax.imshow(lda.scalings_[:, i].reshape((25, 25)), cmap=plt.cm.bone)
```

```
    ax.set_title('LDA\nEigenface {}'.format(i + 1))
plt.show()
```

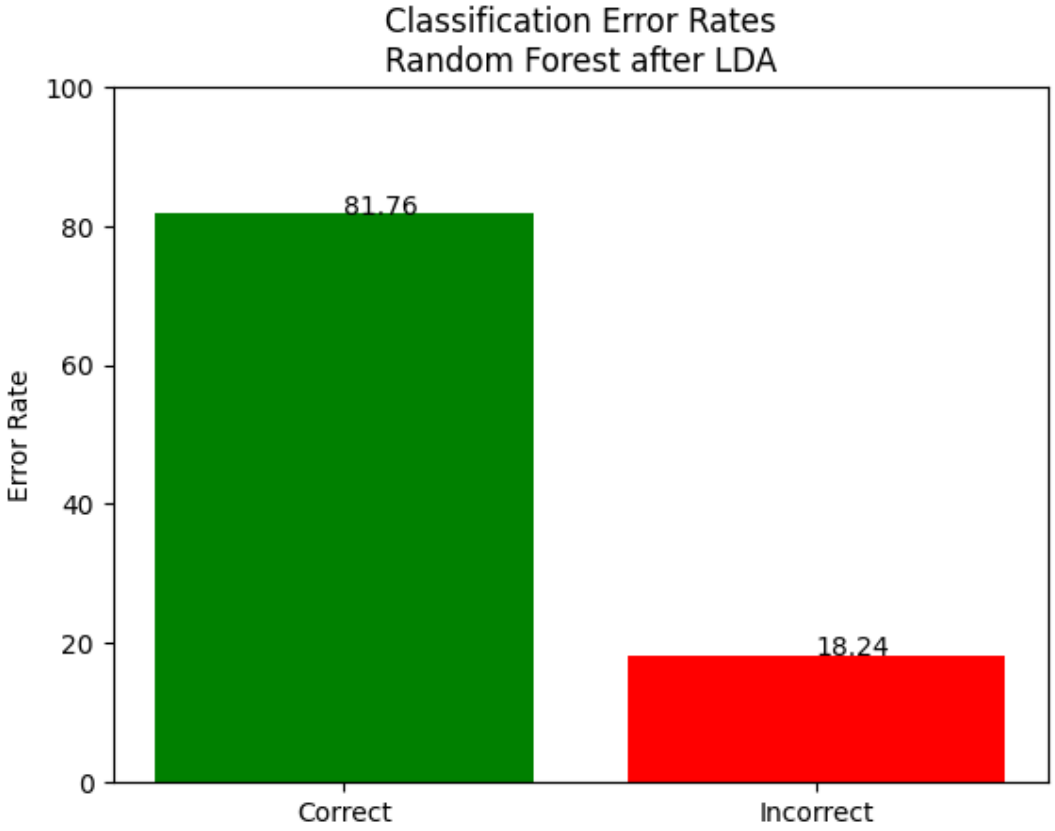


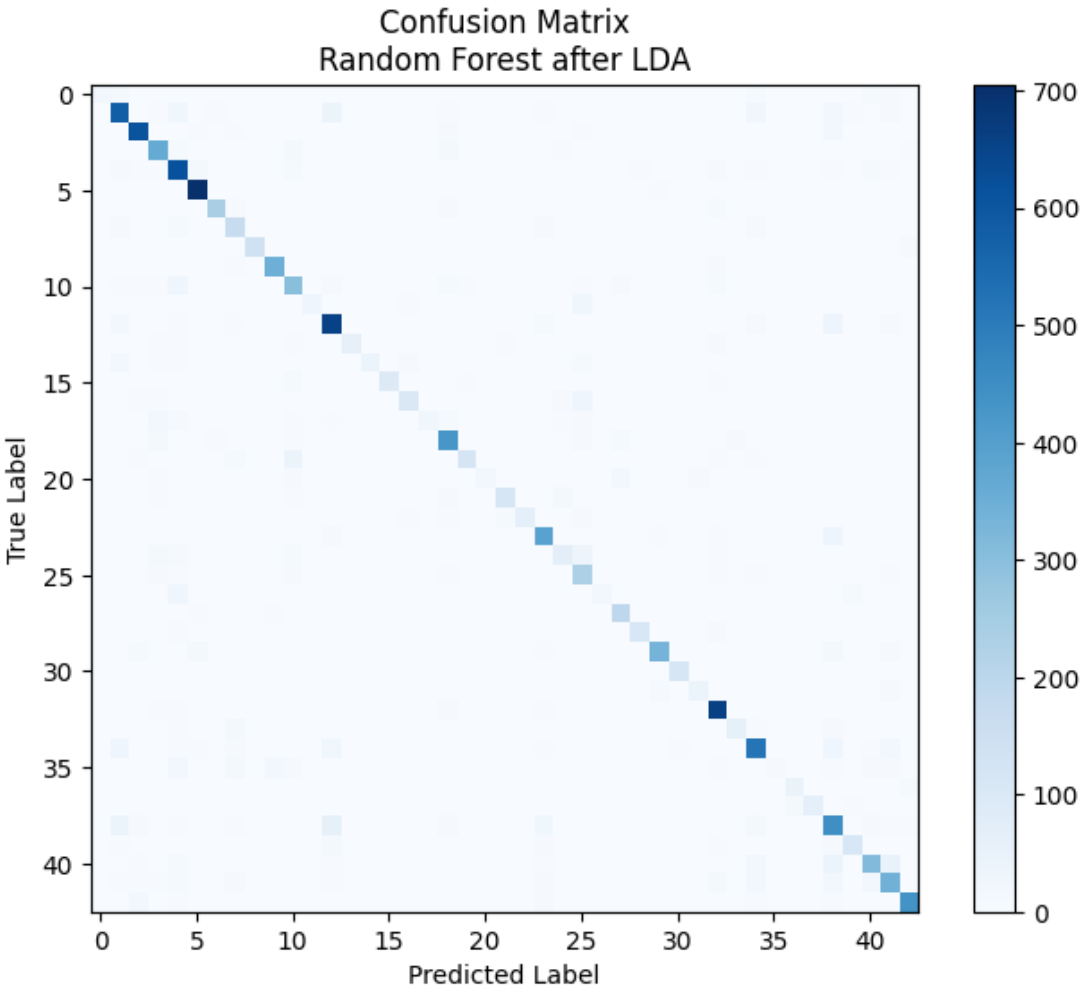


In [ ]:
```
# Transform the train and test images using LDA
train_img_lda = lda.transform(train_img_processed)
test_img_lda = lda.transform(test_img_processed)

# Train the Random Forest Classifier
rf = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=0)
rf.fit(train_img_lda, train_label)

# Predict the labels for the test images
test_labels_predicted = rf.predict(test_img_lda)
```

In [ ]:
```
# Evaluate the classifier for Random Forest after LDA
evaluate_classifier(test_labels, test_labels_predicted, 'Random Forest after LDA
plot_10_predicted_images(test_img, test_labels, test_labels_predicted, meta_data
```

```
Correct classification rate: 0.8175771971496437
Incorrect classification rate: 0.18242280285035628
```


Classification Error Rates
Random Forest after LDA
Error Rate
100
80
60
40
20
0
81.76
18.24
Correct
Incorrect


Confusion Matrix
Random Forest after LDA
True Label
Predicted Label
0
5
10
15
20
25
30
35
40
700
600
500
400
300
200
100
0

```
C:\Users\adria\AppData\Local\Temp\ipykernel_15976\1045172290.py:131: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.
  meta_img = np.array(meta_img)
(43,)
```

In [ ]:

```python
def convert_to_gray(images):
    gray_images = []
    for i in range(len(images)):
        gray_image = np.dot(images[i][...,:3], [0.299, 0.587, 0.114])
        gray_images.append(gray_image)
    return np.array(gray_images)

def normalize_images(images):
    norm_images = []
    for i in range(len(images)):
        norm_image = images[i] / 255
        norm_images.append(norm_image)
    return np.array(norm_images)

def resize_images(images):
    min_height = min(img.shape[0] for img in images)
    min_width = min(img.shape[1] for img in images)

    resized_images = []
    for i in range(len(images)):
        resized_image = sk_transform.resize(images[i], (min_height, min_width))
        resized_images.append(resized_image)
    return np.array(resized_images)

def flatten_images(images):
    vectorized_images = []
    for i in range(len(images)):
        vectorized_image = images[i].flatten()
        vectorized_images.append(vectorized_image)
    return np.array(vectorized_images)

def read_csv(file_path):
    with open(file_path) as csvfile:
        readCSV = csv.reader(csvfile, delimiter=',')
        data = []
        for row in readCSV:
            data.append(row)
    return np.array(data)

def get_images_and_labels(data):
    images = []
    labels = []
    for i in range(1, len(data)):
```

```
        label = data[i, 6]
        path = data[i, 7]

        image = plt.imread(path)
        images.append(image)
        labels.append(label)
    return images, labels

def preprocess_images(images):
    # Convert to grayscale
    gray_images = convert_to_gray(images)
    # Normalize
    norm_images = normalize_images(gray_images)
    # Resize
    resized_images = resize_images(norm_images)
    # Flatten
    flattened_images = flatten_images(resized_images)
    return flattened_images

def plot_cummulative_PCA(pca):
    explained_variance_ratio = pca.explained_variance_ratio_
    cumulative_explained_variance = np.cumsum(explained_variance_ratio)
    plt.plot(cumulative_explained_variance)
    plt.title('Cumulative Sum of Explained Variance Ratio for PCA')
    plt.xlabel('Number of Components')
    plt.ylabel('Cumulative Explained Variance')
    plt.grid()
    plt.show()

def plot_eigenfaces(components, title_prefix):
    fig = plt.figure(figsize=(8, 6))
    for i in range(2):
        ax = fig.add_subplot(1, 2, i + 1, xticks=[], yticks=[])
        ax.imshow(components[i].reshape((25, 25)), cmap=plt.cm.bone)
        ax.set_title('{}\nEigenface {}'.format(title_prefix, i + 1))
    plt.show()

def evaluate_classifier(test_labels, predicted_labels, title):
    correct_rate = accuracy_score(test_labels, predicted_labels)
    incorrect_rate = 1 - correct_rate

    print('Correct classification rate:', correct_rate)
    print('Incorrect classification rate:', incorrect_rate)

    bar_plot_error_rates(correct_rate, incorrect_rate, title)
    plot_confusion_matrix(test_labels, predicted_labels, title)

    return predicted_labels

def plot_confusion_matrix(true_labels, predicted_labels, title):
    cm = confusion_matrix(true_labels, predicted_labels)
    plt.figure(figsize=(8, 6))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title('Confusion Matrix\n' + title)
    plt.colorbar()
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

def bar_plot_error_rates(correct_rate, incorrect_rate, title):
```

```
    correct_rate *= 100
    incorrect_rate *= 100

    plt.bar(['Correct', 'Incorrect'], [correct_rate, incorrect_rate], color=['gr
    plt.title('Classification Error Rates\n' + title)
    plt.ylabel('Error Rate')

    # Add labels of the correct and incorrect rates to the bars
    plt.text(0, correct_rate + 0.01, str(round(correct_rate, 2)))
    plt.text(1, incorrect_rate + 0.01, str(round(incorrect_rate, 2)))
    plt.ylim([0,100])

    plt.show()

def get_meta_images(meta):
    label = []
    for i in range(len(meta)):
        label.append(i)

    meta_img = []
    for el in label:
        for i in range(1,len(meta)):
            if el == int(meta[i][1]):
                path = meta[i][0]
                img = plt.imread(path)
                meta_img.append(img)

    meta_img = np.array(meta_img)
    return meta_img

def plot_10_predicted_images(images, true_labels, predicted_labels, meta_data):
    meta_img = get_meta_images(meta_data)
    print(meta_img.shape)
    # Plot the first 10 images and their predicted labels compared to the meta d
    fig, axes = plt.subplots(2, 10, figsize=(16, 4))
    for i in range(10):
        # Plot predicted image
        axes[0, i].imshow(images[i], cmap=plt.cm.bone)
        axes[0, i].set_title('Predicted: {}'.format(predicted_labels[i]))
        axes[0, i].axis('off')

        # Plot actual image
        j = int(test_labels_predicted[i])
        axes[1, i].imshow(meta_img[j], cmap=plt.cm.bone)
        axes[1, i].set_title('Actual: {}'.format(true_labels[i]))
        axes[1, i].axis('off')
    plt.show()
```

2 Question 2

Consider a predator-prey dynamics such as the simple Lotka-Volterra mode:

$$\begin{aligned} \boldsymbol{x'} &= \boldsymbol{f(x)} \\ \boldsymbol{x} &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} Praypopulation \\ Predatorpopulation \end{bmatrix} \\ \boldsymbol{f(x)} &= \begin{bmatrix} (b - px_2)x_1 \\ (rx_1 - d)x_2 \end{bmatrix} \end{aligned}$$

Without predators, the prey population increases (exponentially) without bound, whereas without prey, the predator population diminishes (exponentially) to zero. The nonlinear interaction, with predators eating prey, tends to diminish the prey population and increase the predator population. Use your Runga-Kutta to solve this system, with the values $b = p = r = d = 1$, $x_1(0) = 0.3$, and $x_2(0) = 0.2$.

Answer: In addressing the predator-prey dynamics described by the Lotka-Volterra model, I employed my Runge-Kutta 4 algorithm, as implemented in homework 3. The numerical integration scheme is outlined as follows:

$$\begin{aligned} k_1 &= h \cdot f(x_n, y_n) \\ k_2 &= h \cdot f(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 &= h \cdot f(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 &= h \cdot f(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

The Lotka-Volterra function, denoted as $f(x)$ in the system, encapsulates the dynamics of prey and predator interactions. With the specified parameters $b = p = r = d = 1$, and initial conditions $x_1(0) = 0.3$ and $x_2(0) = 0.2$, I applied a time step of $h = 0.001$ to discretize the system. Below is the figures for predators plotted against pray and predators and prays plotted against time:

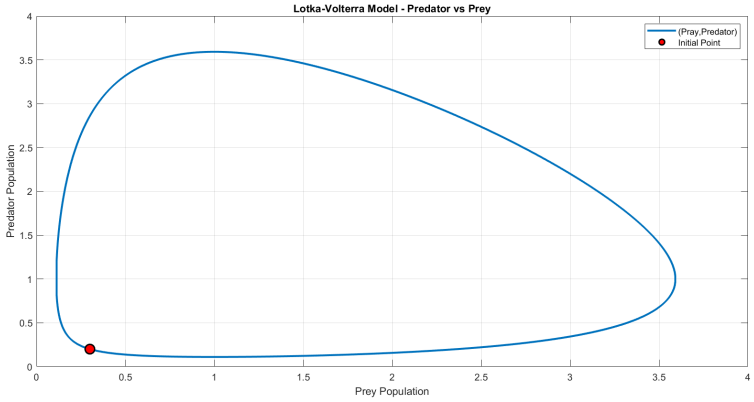


Figure 5: Pray and Predators

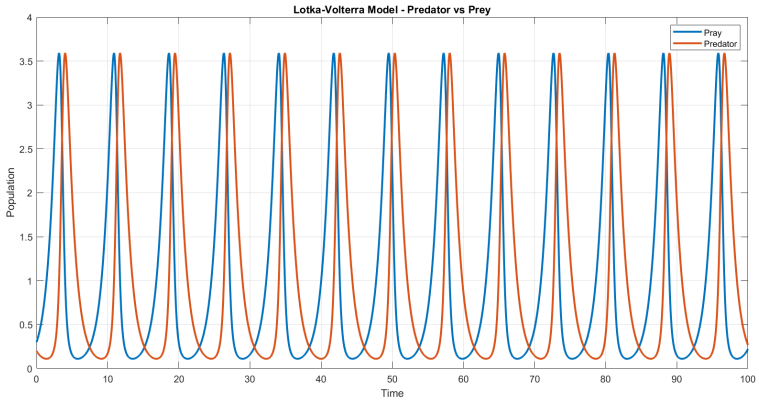


Figure 6: Pray and Predators against time

As illustrated in the plots, the population dynamics exhibit a cyclic behavior. When the predator population is low, the prey population flourishes, leading to an increase in predators. Subsequently, a decline in prey occurs, followed by a decrease in predators. This cyclical pattern persists, illustrating the intricate nonlinear interactions between the predator and prey populations. The Runge-Kutta 4 method effectively captures these dynamics, providing an understanding of the system's behavior.

```
% Problem 2
clear all; clc;

%% Initial conditions
x1_0 = 0.3; % Pray population t0
x2_0 = 0.2; % Predator population t0
x0 = [x1_0; x2_0];

%% Runge-Kutta 4 solver
h = 0.001;
t_end = 100;

[t, x_RK4] = RK4(x0,t_end,h);
x1 = x_RK4(1,:);
x2 = x_RK4(2,:);

% Plotting the result
plot_result_time(t,x1,x2);
plot_result(x1_0,x2_0,x1,x2);

%% Lotka - Volterra model
function y_dot = f(t,x)
    % Growth parameters
    b = 1;
    p = 1;
    r = 1;
    d = 1;

    x1 = x(1);
    x2 = x(2);
    x1_dot = (b-p*x2)*x1;
    x2_dot = (r*x1-d)*x2;

    y_dot = [x1_dot;x2_dot];
end

function [x,y] = RK4(x0,t_end,h)
    t = (0:h:t_end);
    y = zeros(2,length(t));
    y(:,1) = x0;

    for i = 1:(length(t)-1)
        ti = t(i);
        yi = y(:,i);

        k1 = h * f(ti, yi);
        k2 = h * f(ti + h/2, yi + k1/2);
        k3 = h * f(i + h/2, yi + k2/2);
        k4 = h * f(ti + h, yi+ k3);

        y(:,i+1) = yi + (1/6)*(k1+2*k2+2*k3+k4);
    end
   x = t;
end
```

```
function plot_result_time(t, x1,x2)
    figure;
    plot(t, x1, 'LineWidth', 2, 'DisplayName', 'Pray');
    hold on;
    plot(t, x2, 'LineWidth', 2, 'DisplayName', 'Predator');
    title('Lotka-Volterra Model - Predator vs Prey');
    xlabel('Time');
    ylabel('Population');
    legend('show');
    grid on;
    hold off;
end

function plot_result(x1_0, x2_0,x1,x2)
    figure;
    plot(x1, x2, 'LineWidth', 2, 'DisplayName', '(Pray,Predator)');
    hold on;
    scatter(x1_0, x2_0, 100, 'filled', 'MarkerFaceColor', 'r', 'MarkerEdgeColor',↙
'k', 'LineWidth', 1.5, 'DisplayName', 'Initial Point');
    title('Lotka-Volterra Model - Predator vs Prey');
    xlabel('Prey Population');
    ylabel('Predator Population');
    legend('show');
    grid on;
    hold off;
end
```