Object Recognition Using German Traffic Sign Dataset

Problem Description

In robotics, recognizing objects in the environment is crucial. Here, we use the German Traffic Sign dataset to compute subspaces for PCA and LDA methods. The tasks include:

- Illustration of the 1st and 2nd eigenvectors for PCA and LDA.
- Computing recognition rates for the test set, including:
  - Correct classifications
  - Incorrect classifications
- Suggesting at least one improvement for system performance.

1 Principal Component Analysis (PCA)

PCA Overview:

- PCA is a dimensionality reduction technique that identifies directions of maximum variance in the data (principal components).
- Eigenvectors represent these directions, and eigenvalues quantify the variance explained by each eigenvector.

2 Linear Discriminant Analysis (LDA)

LDA Overview:

- LDA is a classification and dimensionality reduction technique that maximizes class separability by finding directions that maximize the distance between class means while minimizing intra-class variance.

3 Methodology

3.1 1. Data Loading and Preprocessing

Steps:

- Load image paths and labels from CSV files.
- Crop images based on the Region of Interest (ROI) and resize them to 30×30 pixels.
- Convert images to grayscale and store them in NumPy arrays.

3.2 2. Principal Component Analysis (PCA)

Steps:

- Apply PCA to retain 95% of the variance.
- Extract PCA components (eigenvectors) and reduce dimensions for training and testing datasets.

3.3 3. Linear Discriminant Analysis (LDA)

Steps:

- Apply LDA, setting the number of components to the number of classes minus one.
- Extract the top two eigenvectors for visualization.

3.4 4. Recognition Rate Computation

Steps:

- Train a Random Forest classifier with the reduced data.
- Predict labels for the test set and compute accuracy and error rates.

4 Results

4.1 Recognition Rates

- **PCA:**
  - Accuracy: 0.7462
  - Error Rate: 0.2538
- **LDA:**
  - Accuracy: 0.8784
  - Error Rate: 0.1216

4.2 Eigenvectors

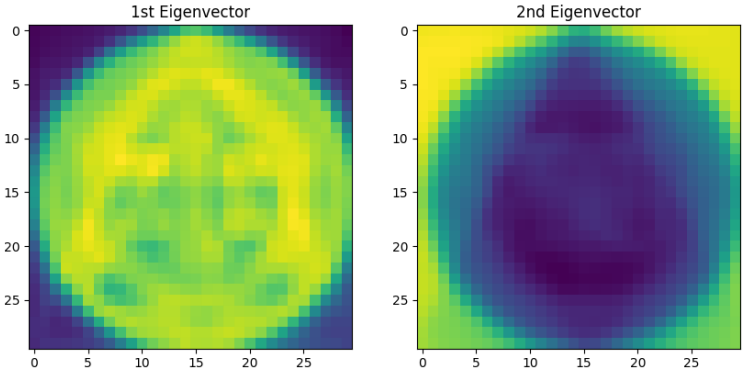

Figure 1: 1st and 2nd Eigenvectors from PCA

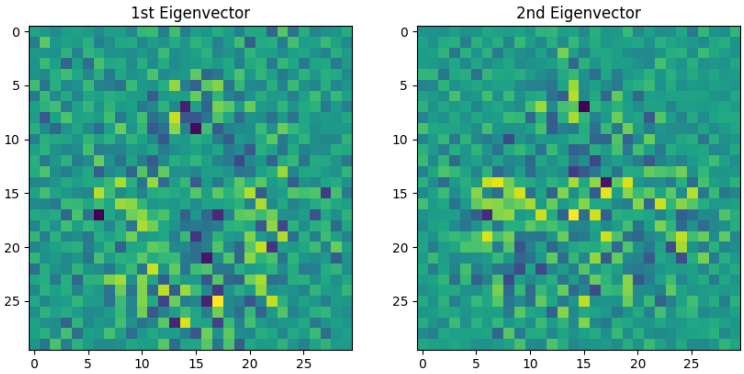

Figure 2: 1st and 2nd Eigenvectors from LDA

5 Code Implementation

```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import matplotlib.pyplot  as plt
import cv2


def load_train_data(train_csv,image_size=(30,30)):

    #Load CSV files
    train_labels = pd.read_csv(train_csv)
    x_train=[]
    y_train=[]

    #Read the train image
    for _,row in train_labels.iterrows():
        img_path="/Users/ivanlin328/Desktop/CSE 276C/HW4/archive/" + row['Path']
        class_id=row['ClassId']

        #Get the ROI(Region of interest)coordinate and crop the image
        img=cv2.imread(img_path)
        x1,y1,x2,y2 = int(row['Roi.X1']),int(row['Roi.Y1']),int(row['Roi.X2']),int(row['Roi
        roi_img=img[y1:y2,x1:x2]

        #resize the cropped image and convert to grayscale
        roi_img= cv2.resize(roi_img,image_size)
        roi_img= cv2.cvtColor(roi_img,cv2.COLOR_BGR2GRAY)

        #Append the cropped image data and label(class_id)to the test lists
        x_train.append(roi_img)
        y_train.append(class_id)

    x_train=np.array(x_train).reshape(len(x_train),-1)
    y_train=np.array(y_train)

    return x_train,y_train



def load_test_data(test_csv,image_size=(30,30)):

    #Load CSV files
    test_labels = pd.read_csv(test_csv)
    x_test, y_test = [], []

        #Read the test image
    for _,row in test_labels.iterrows():
```

```
        img_path="/Users/ivanlin328/Desktop/CSE 276C/HW4/archive/" + row['Path']
        class_id=row['ClassId']

        #Get the ROI(Region of interest)coordinate and crop the image
        img=cv2.imread(img_path)
        x1,y1,x2,y2 = int(row['Roi.X1']),int(row['Roi.Y1']),int(row['Roi.X2']),int(row['Roi
        roi_img=img[y1:y2,x1:x2]

        #resize the cropped image and convert to grayscale
        roi_img= cv2.resize(roi_img,image_size)
        roi_img= cv2.cvtColor(roi_img,cv2.COLOR_BGR2GRAY)

        #Append the cropped image data and label(class_id)to the test lists
        x_test.append(roi_img)
        y_test.append(class_id)

    x_test=np.array(x_test).reshape(len(x_test),-1)
    y_test=np.array(y_test)

    return x_test, y_test

def apply_pca(x_train,x_test):
    pca=PCA(n_components=0.95)
    x_train_pca= pca.fit_transform(x_train)
    x_test_pca=pca.transform(x_test)
    components=pca.components_

    return x_train_pca, x_test_pca,components

def plot_eigenvectors_pca(components_):
     plt.figure(figsize=(10, 5))
     plt.subplot(121)
     plt.imshow(components_[0].reshape(30, 30))
     plt.title('1st Eigenvector')
     plt.subplot(122)
     plt.imshow(components_[1].reshape(30, 30))
     plt.title('2nd Eigenvector')
     plt.show()

def apply_lda(x_train,x_test,y_train):
    lda= LDA(n_components=len(np.unique(y_train))-1)
    x_train_lda=lda.fit_transform(x_train,y_train)
    x_test_lda=lda.transform(x_test)
    eigen=lda.scalings_[:, :2]

    return x_test_lda,x_train_lda,eigen
```

```
def plot_eigenvectors_lda(eigen):
     plt.figure(figsize=(10, 5))
     plt.subplot(121)
     plt.imshow(eigen[:,0].reshape(30, 30))
     plt.title('1st Eigenvector')
     plt.subplot(122)
     plt.imshow(eigen[:,1].reshape(30, 30))
     plt.title('2nd Eigenvector')
     plt.show()

def compute_accuracy(x_train, x_test, y_train, y_test):
    rf = RandomForestClassifier(n_estimators=100, random_state=42)
    rf.fit(x_train, y_train)
    y_pred = rf.predict(x_test)
    correct = accuracy_score(y_test, y_pred)
    incorrect = 1 - correct
    print(f"Accuracy: {correct:.4f}")
    print(f"Error Rate: {incorrect:.4f}")
    return correct, incorrect

# Load and preprocess data
x_train, y_train = load_train_data("/Users/ivanlin328/Desktop/CSE 276C/HW4/archive/Train.csv
x_test, y_test = load_test_data("/Users/ivanlin328/Desktop/CSE 276C/HW4/archive/Test.csv", i

scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test= scaler.transform(x_test)

# Apply PCA and LDA
x_train_pca, x_test_pca, components = apply_pca(x_train, x_test)
x_test_lda, x_train_lda,eigen = apply_lda(x_train, x_test, y_train)

plot_eigen_pca= plot_eigenvectors_pca(components)
plot_eigen_lda=plot_eigenvectors_lda(eigen)

# Calculate recognition rates
print("PCA Recognition Rate:")
PCA_Recognition_Rate = compute_accuracy(x_train_pca, x_test_pca, y_train, y_test)

print("\nLDA Recognition Rate:")
LDA_Recognition_Rate = compute_accuracy(x_train_lda, x_test_lda, y_train, y_test)
```

6 Suggestions for Improvement

- Enhance preprocessing by applying data augmentation (e.g., rotation, scaling) to increase training set diversity.
- Experiment with other classifiers like Support Vector Machines (SVM) or Convolutional Neural Networks (CNN) for better performance.

Predator-Prey Dynamics: Lotka-Volterra Model

The Lotka-Volterra equations describe the interaction between predator and prey populations:

$$x' = f(x), \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \text{Prey population} \\ \text{Predator population} \end{pmatrix}$$

$$f(x) = \begin{pmatrix} (b - px_2)x_1 \\ (rx_1 - d)x_2 \end{pmatrix}$$

Parameters and Initial Conditions:

- $b = p = r = d = 1$
- $x_1(0) = 0.3$, $x_2(0) = 0.2$

7 Numerical Solution:

```
import numpy as np
import matplotlib.pyplot as plt

def lotka_volterra(x,b, p, r, d):

    x1,x2=x
    dx1_dt=(b - p*x2) * x1
    dx2_dt=(r * x1 - d) * x2

    return np.array([dx1_dt,dx2_dt])

def fourth_order_Runge_Kutta(fx,x0,h,t_span,params):
    t_start,t_end=t_span
    t = np.arange(t_start, t_end + h, h)
    x = np.zeros((len(t), len(x0)))
    x[0] = x0
    # Implement fourth_order_Runge_Kutta
    for i in range(1, len(t)):
        k1 = h * fx(x[i-1], *params)
        k2 = h * fx(x[i-1] + 0.5*k1, *params)
```

```
        k3 = h * fx(x[i-1] + 0.5*k2, *params)
        k4 = h * fx(x[i-1] + k3, *params)

        x[i] = x[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

    return t, x

x0=([0.3,0.2])
h=0.01
time_span=(0,20)
t,solution=fourth_order_Runge_Kutta(lotka_volterra,x0,h,time_span,(1,1,1,1))
print(solution)
plt.plot(t, solution[:, 0], label='Prey Population')
plt.plot(t, solution[:, 1], label='Predator Population')
plt.title('Lotka-Volterra Predator-Prey Dynamics')
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend()
plt.grid(True)
plt.show()
```

The solution would be

$$x = \begin{bmatrix} 0.3 & 0.2 \\ 0.30241174 & 0.19860728 \\ 0.30484708 & 0.19722904 \\ \vdots & \vdots \\ 0.29612203 & 2.84633847 \\ 0.29073391 & 2.82629755 \\ 0.28550105 & 2.80624872 \end{bmatrix}$$

8 Result

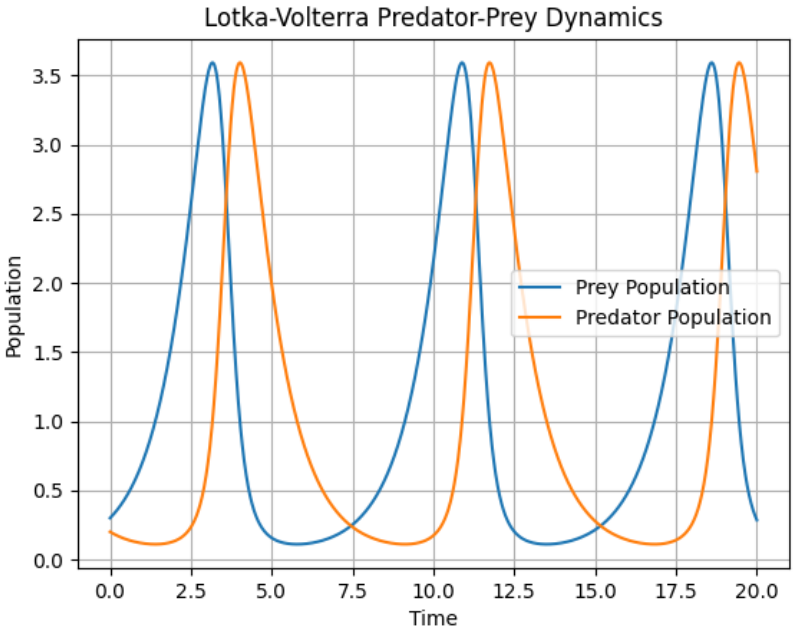

Figure 3: Lotka-Volterra Predator-Prey Dynamics