ECE 66100 Computer Vision
Homework 9
11/27/2022

Name: *Nikita Ravi (ravi30)*

1 Face Recognition using PCA and LDA

Classifying images is a vital subset of computer vision. But an image these days can contain so much information that the processing time can end up becoming exponential. To combat this issue, techniques like PCA and LDA are utilized to reduce the dimension of the images and still make reasonable classifications.

1.1 PCA

The idea behind Principal Component Analysis or PCA consists of calculating the eigenvectors of the covariance matrix C and retaining eigenvectors corresponding to the P largest eigenvalues. This constitutes the orthogonal PCA feature set. The results of the accuracy using PCA for face detection for P eigenvectors is illustrated in figure 1

1.1.1 The Methodology

1. Vectorize each image such that the dimension of the image is converted from $m \times n$ to $mn \times 1$
2. Normalize the vectorized image x_i

$$\hat{x_i} = \frac{x_i}{\|x_i\|} \tag{1}$$

3. Compute the global mean vector $\vec{m}$ for all N images in the train dataset

$$\vec{m} = \frac{1}{N}\sum_{i=1}^{N}\hat{x_i} \tag{2}$$

4. Subtract the global mean $\vec{m}$ from each image from the training dataset

$$X = \text{X-train} - \vec{m} \tag{3}$$

5. Since finding the eigendecomposition of XX^T is computationally complex so instead we find the eigendecomposition of X^TX to get the eigenvectors $\vec{u}$ first

6. We then get the eigenvectors $\vec{w}$ of XX^T by multiplying X with $\vec{u}$ and the normalizing it
7. Since the rank of XX^T is N, at most N eigenvectors can be picked. So when choosing P eigenvectors, $P < N$
8. We then project X onto the eigenspace by multiplying X with w
9. We utilize K-Nearest Neighbors classifier, with $K = 1$ in the lower dimensional space to classify faces

1.2 LDA

The goal of Linear Discriminant Analysis or LDA is to find the directions in the underlying vector space that are maximally discriminating between the classes. A vector direction is maximally discriminating between the classes if it simultaneously maximizes the between-class scatter and minimizes the within-class scatter along that direction. The results of the accuracy using LDA for face detection for P eigenvectors is illustrated in figure 1

1.2.1 The Methodology

1. Vectorize each image such that the dimension of the image is converted from $m \times n$ to $mn \times 1$
2. Normalize the vectorized image x_i

$$\hat{x_i} = \frac{x_i}{\|x_i\|} \tag{4}$$

3. Compute the global mean vector $\vec{m_G}$ for all N images in the train dataset

$$\vec{m_G} = \frac{1}{N}\sum_{i=1}^{N}\hat{x_i} \tag{5}$$

4. Compute the class mean vector for class using the images in that particular class

$$\vec{m_C} = \frac{1}{C}\sum_{i=1}^{N}\hat{x} \tag{6}$$

5. The eigenvectors $\vec{w}$ are computed such that it maximizes the ratio, called the Fisher Discriminant, of between-class scatter S_B to within-class scatter S_W will be the largest. This is accomplished by conducting a eigendecomposition of $S_W^{-1}S_B$ to get $\vec{w}$. Since S_W can be singular sometimes, its inverse may not exist. As a result the Yu and Yang algorithm is used to combat this.

6. We preserve P eigenvectors from $\vec{w}$

7. We then project the training samples onto the the eigenspace by

$$\text{trainFeature} = \vec{w}(\text{X-train} - \vec{m_G}) \tag{7}$$

8. We utilize K-Nearest Neighbors classifier, with $K = 1$ in the lower dimensional space to classify faces

2 Face Recognition using Autoencoders

An autoencoder is a type of neural network that is usually used for dimensionality reduction. It is comprised of two neural networks: an encoder and a decoder. The encoder takes in an high-dimensional sample and outputs a P-dimensional vector just like the PCA and LDA techniques. The decoder then takes the P-dimensional vector and outputs a sample in the original high-dimensional sample space. During training, the autoencoder refines itself by attempting to recreate the input sample from the corresponding P-dimensional vector representation.

The results and comparisions of the accuracies for facial recognition detected by PCA, LDA, and the autoencoder are illustrated below in figure 1

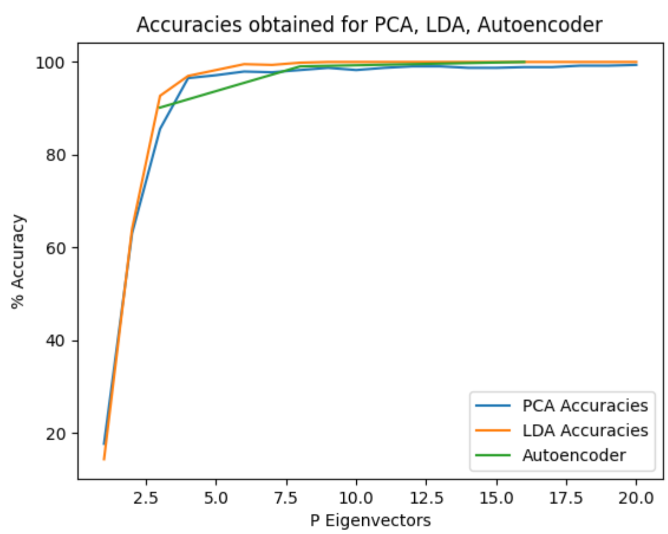


Figure 1: Accuracies for P Eigenvectors using PCA, LDA, and Autoencoder

2.1 Observations

From figure 1 we can see that all three techniques have come close or have achieved a full 100% accuracy. The PCA performed better for lower dimensions, then the LDA, and finally the autoencoder. At higher dimensions but less than N, all three techniques have achieved full 100% accuracy.

3 Car Detection using AdaBoost

AdaBoost stands for Adaptive Boosting because it arranges a set of weak classifiers in a sequence in which each weak classifier is the best choice for a classifier at that point for rectifying the errors made by the previous classifier. A weak classifier is defined as having the ability to at least detect 50% of the images correctly.

3.1 The Methodology

1. Get the integral representation of each image is taken (taking the cumulative sum of each pixel in the image)
2. A feature matrix is created to convert the training dataset to create training feature vectors
3. For each cascade stage t out of a total T stages, a new AdaBoost classifier is added where

    I. the weights are initialized first. Let p be the number of positive classes and n be the number of negative classes

$$w_i = \begin{cases} \frac{1}{2p} \text{ if } x = 1 \\ \frac{1}{2n} \text{ if } x = 0 \end{cases} \tag{8}$$

    II. for each n weak classifier in a total of N weak classifiers:

    i) The weights are normalized $w_i = \frac{w_{ji}}{\sum_j w_{ji}}$
    ii) Iterate over each feature of the images and sort them and their corresponding labels and weights based on the feature element values
    iii) Compute the classification error of the feature elements that would be used as a threshold for both polarities

$$\text{Error if polarity 1: } e_1 = S^+ + T^- - S^- \tag{9}$$

$$\text{Error if polarity -1: } e_{-1} = S^- + T^+ - S^+ \tag{10}$$

Where S^+ is the sum of weights of images belonging to the positive class whose feature value is less than the threshold, S^- is the sum of weights of images belonging to the negative class whose feature value is less than the threshold, T^+ is the sum of weights of images belonging to the positive class, and T is the sum of weights of images belonging to the negative class.
The classification error is calculated by taking the min(e_1, e_{-1}).

iv) The best weak classifier is the weak classifier with the lowest classification error

v) Update the weights with the error ϵ of the best weak classifier as shown below

$$\beta = \frac{\epsilon}{1-\epsilon} \tag{11}$$

$$\alpha = \ln(\frac{1}{\beta}) \tag{12}$$

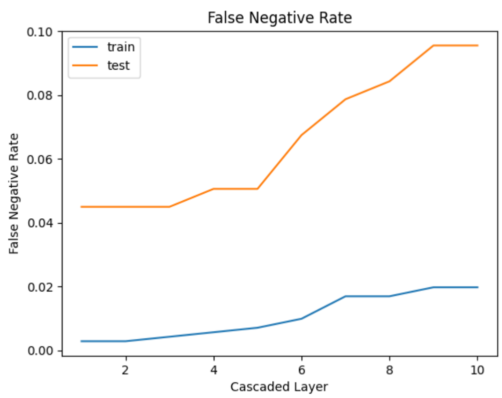
where β represents the confidence and the α is the trust factor

$$w_i = w_i\beta^{1-\epsilon_i} \tag{13}$$

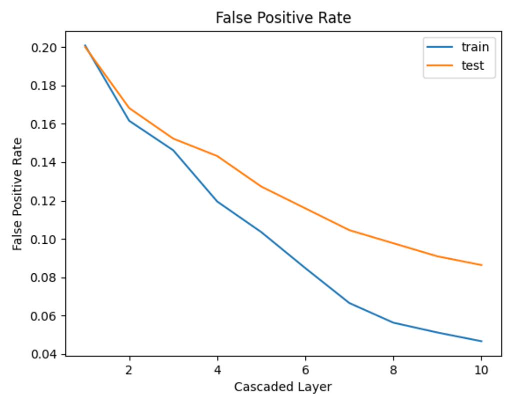
III. We combine all the weak classifiers to form one strong classifier

The AdaBoost implementation was inspired by Fangda Li's implementation.

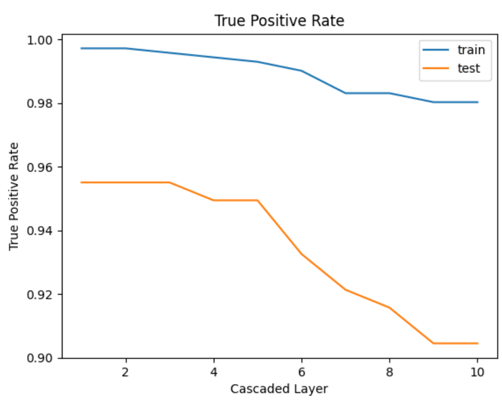
3.2 Results



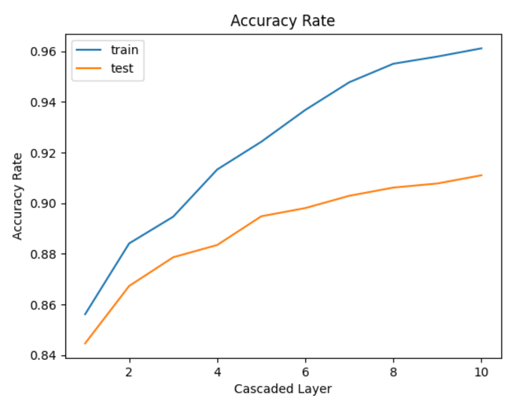
(a) False Negative



(b) False Positive



(c) True Positive



(d) Accuracy

Figure 2: AdaBoost Results for 10 Cascade Layers and 20 Weak Classifiers

3.3 Observations

- As the cascade layer level increased, the false positive rate and true positive rate decreased
- The false negative rate on the other hand increased as the number of cascade layers increased
- The accuracy of detecting the cars increased with every cascade layer and this demonstrates how each weak classifier adds on to and rectifies the previous layers errors to make the detection even more accurate

4 Source Code

```
# Name: Nikita Ravi
# Class: ECE 66100
# Homework #10
# Deadline: 12/09/2022

# Import Modules
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt
from autoencoder import evaluateautoencoder

def getimages(path):
  filenames = [image for image in os.listdir(path) if os.path.isfile(os.path.join(path, image))]
  images = [cv2.imread(os.path.join(path, image)) for image in os.listdir(path) if os.path.isfile(os.path.join(path, image))]

  return images, filenames

def displayimage(image, points=False):
  def clickevent(event, x, y, flags, params):
    # This function was inspired by https://www.geeksforgeeks.org/displaying-the-coordinates-of-the-points-clicked-on-the-image-using-python-opencv/
    if(event == cv2.EVENTLBUTTONDOWN):
      print(x, y)
      font = cv2.FONTHERSHEYSIMPLEX
      cv2.circle(image, (x, y), 1, (0,0,255), thickness=-1)
      cv2.putText(image, str(x) + ’,’ +
            str(y), (x,y), font,
            1, (0, 0, 255), 2)
      cv2.imshow(’window’, image)

  cv2.imshow(”window”, image)
  if(points):
    cv2.setMouseCallback(’window’, clickevent)
  cv2.waitKey(0)
  cv2.destroyAllWindows()
  quit()

def calculatedistance(trainFeatureVectors, testFeatueVector):
  dist = np.linalg.norm(trainFeatureVectors - testFeatueVector, axis=0)
  return dist

class KNearestNeighbors:
  def  i n i t (self, k):
    self.k = k

  def fit(self, Xtrain, ytrain):
    self.Xtrain = Xtrain
    self.ytrain = ytrain
```

```python

  def predict(self, Xtest):
    Xtest = Xtest.T
    predictedclasses = np.zeros(len(Xtest))
    for idx, X in enumerate(Xtest):
      dist = calculatedistance(self.Xtrain, X.reshape(-1, 1))
      sorteddistidx = np.argsort(dist)
      predictedclass = self.ytrain[sorteddistidx[:self.k]]

      frequency = np.bincount(predictedclass)
      label = np.argmax(frequency)
      predictedclasses[idx] = label

    return predictedclasses

class PCA:
  def __init__(self, Xtrain, ytrain, Xtest, ytest, P, k):
    self.k = k
    self.Xtrain = Xtrain
    self.ytrain = ytrain
    self.Xtest = Xtest
    self.ytest = ytest
    self.P = P

  def train(self):
    self.globalmean = np.mean(self.Xtrain, axis=1).reshape(-1, 1)
    X = self.Xtrain - self.globalmean

    _, _, u = np.linalg.svd(X.T @ X)
    w = X @ u # True eigenvectors of covariance
    w /= np.linalg.norm(w, axis=0)

    self.wp = w[:, :self.P] # Preserving P eigenvectors
    trainFeature = self.wp.T @ X # Project onto eigenspace

    self.knn = KNearestNeighbors(self.k)
    self.knn.fit(trainFeature, self.ytrain)

  def test(self):
    X = self.Xtest - self.globalmean
    testFeature = self.wp.T @ X # Project onto eigenspace
    predictedclasses = self.knn.predict(testFeature)

    acc = np.sum((predictedclasses - self.ytest) == 0) / np.float(self.ytest.size) * 100
    return acc

class LDA:
  def __init__(self, Xtrain, ytrain, Xtest, ytest, P, NUMCLASSES, k):
    self.Xtrain = Xtrain
    self.ytrain = ytrain
    self.Xtest = Xtest
    self.ytest = ytest
    self.P = P
```

```
    self.C = NUMCLASSES
    self.k = k

  def train(self):
    self.P = self.C - 1 if self.P ¿ self.C - 1 else self.P
    self.globalmean = np.mean(self.Xtrain, axis=1)

    self.classmeans = np.zeros((self.Xtrain.shape[0], self.C))
    for idx in range(self.C):
      self.classmeans[:, idx] = np.mean(self.Xtrain[:, self.ytrain == idx + 1], axis=1)

    ”””
    It can be shown that when Sw is isotropic (all classes have identical variances in all of the same principal directions), the
    LDA eigenvectors are the eigenvectors of the Sb matrix. These correspond to the space spanned by the —C— - 1 mean difference
    m i - m.
    ”””
    w = self.classmeans - self.globalmean.reshape(-1, 1) # Equation 26
    self.wp = w[:, :self.P] # Preserve P eigenvectors

    trainFeature = self.wp.T @ (self.Xtrain - self.globalmean.reshape(-1, 1)) # Project onto eigenspace
    self.knn = KNearestNeighbors(self.k)
    self.knn.fit(trainFeature, self.ytrain)

  def test(self):
    X = self.Xtest - self.globalmean.reshape(-1, 1)
    testFeature = self.wp.T @ X

    predictedclasses = self.knn.predict(testFeature)
    acc = np.sum((predictedclasses - self.ytest) == 0) / np.float(self.ytest.size) * 100
    return acc

def vectorizefaceimages(X):
  Xmean = np.mean(X)
  Xstd = np.std(X)
  Xnormalized = (X - Xmean) / Xstd

  return Xnormalized

def createfacedatasets(IMAGESIZE, N, images, filenames):
  X = np.zeros((IMAGESIZE, N), dtype=np.float32)
  y = np.zeros(N, dtype=np.int64)

  for idx, image in enumerate(images):
    filename = filenames[idx]
    gray = cv2.cvtColor(image, cv2.COLORBGR2GRAY) if len(image.shape) == 3 else image

    X[:, idx] = gray.flatten()
    y[idx] = int(filename.split(” ”)[0])
```

```
  X = vectorizefaceimages(X) # Normalize the images so the mean is at zero and standard deviation at 1
  return X, y

def plotaccuracies(PCALDAaccuracies, autoencoderaccuracies):
  pca, lda = PCALDAaccuracies

  plt.plot(range(1, 20 + 1), pca, label=”PCA Accuracies”)
  plt.plot(range(1, 20 + 1), lda, label=”LDA Accuracies”)
  plt.plot([3, 8, 16], autoencoderaccuracies, label=”Autoencoder”) # From autoencoder script results

  plt.title(”Accuracies obtained for PCA, LDA, Autoencoder”)
  plt.xlabel(”P Eigenvectors”)
  plt.ylabel(”% Accuracy”)

  plt.legend()
  plt.show()

def task1():
  # Constants
  TRAINDIR = ”/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE 66100/hw10/FaceRecognition/train”
  TESTDIR = ”/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE 66100/hw10/FaceRecognition/test”
  N = 630 # Number of images in both training and testing dataset
  IMAGESIZE = 128 * 128 # 128 x 128 resolution
  NUMCLASSES = 30 # Number of classes
  MAXP = 20 # Maximum number of eigenvalues we want
  NUMNEIGHBORS = 1 # 1-NN Classification

  trainImages, trainFileNames = getimages(TRAINDIR)
  testImages, testFileNames = getimages(TESTDIR)

  Xtrain, ytrain = createfacedatasets(IMAGESIZE, N, trainImages, trainFileNames) # Xtrain shape: (128x128, 630), ytrain shape: (630, 1)
  Xtest, ytest = createfacedatasets(IMAGESIZE, N, testImages, testFileNames) # Xtest shape: (128x128, 630), ytest shape: (630, 1)
  accuraciespcalda = []

  for mode in [”pca”, ”lda”]:
    print(”===========”+mode.upper()+”=================”)
    accuraciesforeachP = []
    for P in range(1, MAXP+1):
      accuracy = None
      if(mode == ”pca”):
        pca = PCA(Xtrain, ytrain, Xtest, ytest, P, k=NUMNEIGHBORS)
        pca.train()
        accuracy = pca.test()

      elif(mode == ”lda”):
        lda = LDA(Xtrain, ytrain, Xtest, ytest, P, NUMCLASSES, k=NUMNEIGHBORS)
```

```
        lda.train()
        accuracy = lda.test()

      accuraciesforeachP.append(accuracy)
      print(f"Accuracy at -P  eigenvectors: -accuracy ")
    accuraciespcalda.append(accuraciesforeachP)

  return accuraciespcalda

def task2():
  accuraciesforp = []
  for p in [3, 8, 16]:
    Xtrain, ytrain, Xtest, ytest = evaluateautoencoder(p, training=False)
    knn = KNearestNeighbors(k=1)
    knn.fit(Xtrain.T, ytrain)
    predictedclasses = knn.predict(Xtest.T)

    accuracy = np.sum((predictedclasses - ytest) == 0) / np.float(ytest.size) * 100
    print(f"Accuracy at -p  eigenvectors: -accuracy ")
    accuraciesforp.append(accuracy)

  return accuraciesforp

def createfeaturematrix(IMAGESIZE, height, width):
  # Haar Kernel Feature Extraction inspired by Fangda Li's code
  numfeatures = 47232 # Initially tried with 60000 but index ended with 47232
  featurematrix = np.zeros((numfeatures, IMAGESIZE), dtype=int)
  index = 0 # Keep track of the index of featurematric when appending to count number of features
  offset = 2 # offset of pixels from image borders

  # Horizontal haar filter
  widthstep, heightstep = 2, 1
  for i in range(1, height, heightstep): # row multiplier
    for j in range(1, width, widthstep): # height multiplier
      for y in range(offset, height - heightstep * i + 1 - offset): # Go through the image
        for x in range(offset, width - widthstep * j + 1 - offset):
          featurematrix[index, y * height + x] = 1.0
          featurematrix[index, y * height + x + widthstep * j//2] = -2.0
          featurematrix[index, y * height + x + widthstep * j] = 1.0
          featurematrix[index, (y + heightstep * i) * height + x] = -1.0
          featurematrix[index, (y + heightstep * i) * height + x + widthstep * j//2] = 2.0
          featurematrix[index, (y + heightstep * i) * height + x + widthstep * j] = -1.0
          index += 1

  # Vertical haar filter
  widthstep, heightstep = 1, 2
  for i in range(1, height, heightstep): # row multiplier
    for j in range(1, width, widthstep): # height multiplier
      for y in range(offset, height - heightstep * i + 1 - offset): # Go
```

```
  through the image
        for x in range(offset, width - widthstep * j + 1 - offset):
          featurematrix[index, y * height + x] = -1.0
          featurematrix[index, y * height + x + widthstep * j] = 1.0
          featurematrix[index, (y + heightstep * i//2) * height + x] = 2.0
          featurematrix[index, (y + heightstep * i//2) * height + x + widthstep * j] = -2.0
          featurematrix[index, (y + heightstep * i) * height + x//2] = -1.0
          featurematrix[index, (y + heightstep * i) * height + x + widthstep * j] = 1.0
          index += 1

  print(f"The number of features are: -index ")
  return featurematrix

class CascadedAdaBoost:
  # This class was inspired by Fangda Li's code
  def  init (self, NUMCASCADES, NUMWEAKCLASSIFIERS, IMAGESIZE, height, width):
    self.NUMCASCADES = NUMCASCADES
    self.NUMWEAKCLASSIFIERS = NUMWEAKCLASSIFIERS
    self.IMAGESIZE = IMAGESIZE
    self.height = height
    self.width = width
    self.cascadedadaboost = []
    self.featurematrix = createfeaturematrix(self.IMAGESIZE, self.height, self.width)
    self.numtrainpositives = None
    self.numtrainnegatives = None
    self.Xtrain = None
    self.ytrain = None

  def fit(self, Xtrain, ytrain):
    self.Xtrain = Xtrain
    self.ytrain = ytrain
    self.numtrain = self.ytrain.size

  def train(self):
    self.trainFeatureVectors = self.featurematrix @ self.Xtrain
    self.numtrainpositives = int(np.sum(self.ytrain))
    self.numtrainnegatives = len(self.ytrain) - self.numtrainpositives

    self.positivetrainfeaturevector = self.trainFeatureVectors[:, self.ytrain == 1]
    self.negativetrainfeaturevector = self.trainFeatureVectors[:, self.ytrain == 0]

    currentpositivesfeaturevectors = self.positivetrainfeaturevector
    currentnegativesfeaturevectors = self.negativetrainfeaturevector

    falsepositivetrain = []
    falsenegativetrain = []
    truepositivetrain = []
    accuracytrain = []
```

```

        falsepositivetest = []
        falsenegativetest = []
        truepositivetest = []
        accuracytest = []

        for idx in range(self.NUMCASCADES):
            print(f”Training AdaBoost with Cascade Number –idx+1 ”)
            currentadaboostclassifier = self.addnewadaboostclassifier()
            currentadaboostclassifier.settrainingfeaturevectors(currentpositivesfeaturevectors, currentnegativesfeaturevectors)

            for weak in range(self.NUMWEAKCLASSIFIERS):
                print(f”Adding weak classifier number: –weak + 1 ”)
                currentadaboostclassifier.addweakclassifier()

            self.FalsePositiveIdx, FalsePositive, FalseNegative, TruePositive, Accuracy = self.classifytrainingdata()
            falsepositivetrain.append(FalsePositive)
            falsenegativetrain.append(FalseNegative)
            truepositivetrain.append(TruePositive)
            accuracytrain.append(Accuracy)

            currentnegativesfeaturevectors = self.negativetrainfeaturevector[:, self.FalsePositiveIdx - self.numtrainpositives]
            FalsePositive, FalseNegative, TruePositive, Accuracy = self.classifytestingdata()
            falsepositivetest.append(FalsePositive)
            falsenegativetest.append(FalseNegative)
            truepositivetest.append(TruePositive)
            accuracytest.append(Accuracy)

        return falsepositivetrain, falsepositivetest, falsenegativetrain, falsenegativetest, truepositivetrain, truepositivetest, accuracytrain, accuracytest

    def addnewadaboostclassifier(self):
        adaboost = Adaboost()
        adaboost.setfeaturematrix(self.featurematrix)
        self.cascadedadaboost.append(adaboost)

        return adaboost

    def classifytrainingdata(self):
        tempfeaturevector = self.trainFeatureVectors
        posidx = np.arange(self.numtrain)

        for classifier in self.cascadedadaboost:
            predicted = classifier.classifyfeaturevectors(tempfeaturevector)
            tempfeaturevector = tempfeaturevector[:, predicted == 1]
            posidx = posidx[predicted == 1]

        # Sort TruePositives, FalsePositives, FalseNegatives
        FalsePositiveIdx = posidx[np.take(self.ytrain, posidx) == 0]
```

```
    numtruepositives = np.sum(np.take(self.ytrain, posidx))
    TruePositive = numtruepositives / self.numtrainpositives
    FalsePositive = (posidx.size - numtruepositives) / self.numtrainnegatives
    FalseNegative =  1 - TruePositive
    w = self.numtrainpositives / (self.numtrainpositives + self.numtrainnegatives)
    Accuracy = TruePositive * w + (1 - FalsePositive) * (1 - w)

    print("Training FP = %.4f, FN = %.4f, TP = %.4f, Acc = %.4f" % (FalsePositive, FalseNegative, TruePositive, Accuracy))
    return FalsePositiveIdx, FalsePositive, FalseNegative, TruePositive, Accuracy

  def settesting(self, Xtest, ytest):
    self.Xtest = Xtest
    self.ytest = ytest

  def classifytestingdata(self):
    self.testFeatureVectors = self.featurematrix @ self.Xtest
    tempfeaturevector = self.testFeatureVectors

    self.numtestpositives = int(np.sum(self.ytest))
    self.numtestnegatives = len(self.ytest) - self.numtestpositives

    numtest = self.ytest.size
    posidx = np.arange(numtest)

    for classifier in self.cascadedadaboost:
      predicted = classifier.classifyfeaturevectors(tempfeaturevector)
      tempfeaturevector = tempfeaturevector[:, predicted == 1]
      posidx = posidx[predicted == 1]

    # Sort TruePositives, FalsePositives, FalseNegatives
    FalsePositiveIdx = posidx[np.take(self.ytest, posidx) == 0]
    numtruepositives = np.sum(np.take(self.ytest, posidx))
    TruePositive = numtruepositives / self.numtestpositives
    FalsePositive = (posidx.size - numtruepositives) / self.numtestnegatives
    FalseNegative =  1 - TruePositive
    w = self.numtestpositives / (self.numtestpositives + self.numtestnegatives)
    Accuracy = TruePositive * w + (1 - FalsePositive) * (1 - w)

    print("Testing FP = %.4f, FN = %.4f, TP = %.4f, Acc = %.4f" % (FalsePositive, FalseNegative, TruePositive, Accuracy))
    return FalsePositive, FalseNegative, TruePositive, Accuracy

class Adaboost:
  # This class was inspired by Fangda Li's code
  def  init (self):
    self.weakclassifierindex = np.array([], dtype=int)
    self.weakclassifierpolarities = np.array([])
    self.weakclassifierthreshold = np.array([])
```

```
    self.weakclassifierweights = np.array([])
    self.weakclassifierresults = np.array([])
    self.weakclassifierweightedresults = None
    self.threshold = 1.0
    self.ytrain = None
    self.trainsortedidx = None
    self.trainfeaturevector = None
    self.numpositive = None
    self.numnegative = None
    self.weightsforsample = None

  def setfeaturematrix(self, featurematrix):
    self.featurematrix = featurematrix

  def settrainingfeaturevectors(self, positivefeaturevector, negativefeaturevector):
    self.NUMFEATURES = positivefeaturevector.shape[0]
    self.numpositive = positivefeaturevector.shape[1] # shape: numfeaturs=47232 x numimages
    self.numnegative = negativefeaturevector.shape[1]
    self.trainfeaturevector = np.hstack((positivefeaturevector, negativefeaturevector))
    self.ytrain = np.hstack((np.ones(self.numpositive), np.zeros(self.numnegative)))
    self.trainsortedidx = np.argsort(self.trainfeaturevector, axis=1)

    print(f”Number of positive training data: -self.numpositive  / negative training data: -self.numnegative ”)

  def addweakclassifier(self):
    # Initialize the weights for the first weak classifier
    if(not self.weakclassifierindex.size):
      self.weightsforsample = np.zeros(self.ytrain.size, dtype=float)
      self.weightsforsample[self.ytrain == 1] = 1 / (2 * self.numpositive)
      self.weightsforsample[self.ytrain == 0] = 1 / (2 * self.numnegative)

    # Normalize the weights
    else:
      self.weightsforsample /= np.sum(self.weightsforsample)

    # Get the best weak classifier that minimizes the error with the current weights
    bestfeatureindex, bestfeaturepolarity, bestfeaturethreshold, bestfeatureerror, bestfeatureresults = self.getbestweakclassifier()

    self.weakclassifierindex = np.append(self.weakclassifierindex, bestfeatureindex)
    self.weakclassifierpolarities = np.append(self.weakclassifierpolarities, bestfeaturepolarity)
    self.weakclassifierthreshold = np.append(self.weakclassifierthreshold, bestfeaturethreshold)

    # Confidence
```

```
    beta = bestfeatureerror / (1 - bestfeatureerror)

    # Trust Factor
    alpha = np.log(1 / np.abs(beta))
    self.weakclassifierweights = np.append(self.weakclassifierweights, alpha)
    e = np.abs(bestfeatureresults - self.ytrain)

    # Update the weights
    self.weightsforsample = self.weightsforsample * beta ** (1 - e)

    # Adjust the threshold
    if(not len(self.weakclassifierresults)):
      self.weakclassifierresults = bestfeatureresults.reshape(-1,1)
    else:
      self.weakclassifierresults = np.hstack((self.weakclassifierresults, bestfeatureresults.reshape(-1,1)))

    self.weakclassifierweightedresults = np.dot(self.weakclassifierresults, self.weakclassifierweights)
    self.threshold = min(self.weakclassifierweightedresults[self.ytrain == 1])

  def getbestweakclassifier(self):
    featureerrors = np.zeros(self.NUMFEATURES)
    featurethreshold = np.zeros(self.NUMFEATURES)
    featurepolarity = np.zeros(self.NUMFEATURES)
    featuresortedidx = np.zeros(self.NUMFEATURES, dtype=int)

    Tpos = np.sum(self.weightsforsample[self.ytrain == 1])
    Tneg = np.sum(self.weightsforsample[self.ytrain == 0])
    for f in range(self.NUMFEATURES):
      sortedweights = self.weightsforsample[self.trainsortedidx[f, :]]
      sortedlabels = self.ytrain[self.trainsortedidx[f, :]]

      Spos = np.cumsum(sortedlabels * sortedweights)
      Sneg = np.cumsum(sortedweights) - Spos

      Epos = Spos + Tneg - Sneg
      Eneg = Sneg + Tpos - Spos

      polarities = np.zeros(self.numpositive + self.numnegative)
      polarities[Epos ¿ Eneg] = -1
      polarities[Epos ¡= Eneg] = 1
      # print(Epos ¿ Eneg)

      errors = np.minimum(Epos, Eneg)
      sortedidx = np.argmin(errors)

      minerrorsampleidx = self.trainsortedidx[f, sortedidx]
      minerror = np.min(errors)

      threshold = self.trainfeaturevector[f, minerrorsampleidx]
      polarities = polarities[sortedidx]
```

```
      featureerrors[f] = minerror
      featurethreshold[f] = threshold
      featurepolarity[f] = polarities
      featuresortedidx[f] = sortedidx

    bestfeatureindex = np.argmin(featureerrors)
    bestfeaturethreshold = featurethreshold[bestfeatureindex]
    bestfeatureerror = featureerrors[bestfeatureindex]
    bestfeaturepolarity = featurepolarity[bestfeatureindex]

    bestfeatureresults = np.zeros(self.numpositive + self.numnegative)
    bestsortedindex = featuresortedidx[bestfeatureindex]
    if(bestfeaturepolarity == 1):
      bestfeatureresults[self.trainsortedidx[bestfeatureindex, bestsortedindex:]] = 1
    else:
      bestfeatureresults[self.trainsortedidx[bestfeatureindex, :bestsortedindex]] = 1

    return bestfeatureindex, bestfeaturepolarity, bestfeaturethreshold, bestfeatureerror, bestfeatureresults

  def classifyfeaturevectors(self, trainfeaturevector):
    weakclassifiers = trainfeaturevector[self.weakclassifierindex, :]

    polarityvector = self.weakclassifierpolarities.reshape(-1, 1)
    thresholdvector = self.weakclassifierthreshold.reshape(-1, 1)

    # Predictions made by weak classifier
    weakclassifierpredictions = weakclassifiers * polarityvector ¿ thresholdvector * polarityvector
    weakclassifierpredictions[weakclassifierpredictions == True] = 1
    weakclassifierpredictions[weakclassifierpredictions == False] = 0

    # Apply weak classifier weights
    strongclassifierresult = self.weakclassifierweights @ weakclassifierpredictions

    # Apply strong classifier threshold
    finalpredictions = np.zeros(strongclassifierresult.size)
    finalpredictions[strongclassifierresult ¿= self.threshold] = 1
    return finalpredictions

def getintegralimage(image):
  # Return integral representation of the image
  # https://towardsdatascience.com/understanding-face-detection-with-the-viola-jones-object-detection-framework-c55cc2a9da14
  # Cumulative sum of above and to the left of the current pixel
  return np.cumsum(np.cumsum(image, axis=0), axis=1)

def createcardatasets(IMAGESIZE, positiveimages, negativeimages):
  N = len(positiveimages) + len(negativeimages)
  positiveflatteneddata = np.zeros((IMAGESIZE, len(positiveimages)))
```

```
  negativeflatteneddata = np.zeros((IMAGESIZE, len(negativeimages)))

  for idx, image in enumerate(positiveimages):
    image = cv2.cvtColor(image, cv2.COLORRGBA2GRAY) if len(image.shape) == 3 else image
    image = getintegralimage(image)
    positiveflatteneddata[:, idx] = image.flatten()

  positivelabels = np.ones(len(positiveimages))

  for idx, image in enumerate(negativeimages):
    image = cv2.cvtColor(image, cv2.COLORRGBA2GRAY) if len(image.shape) == 3 else image
    image = getintegralimage(image)
    negativeflatteneddata[:, idx] = image.flatten()

  negativelabels = np.zeros(len(negativeimages))

  X = np.hstack((positiveflatteneddata, negativeflatteneddata))
  y = np.hstack((positivelabels, negativelabels))
  return X, y

def plottask3results(train, test, title):
  plt.plot(range(1, len(train) + 1), train, label="train")
  plt.plot(range(1, len(test) + 1), test, label="test")

  plt.title(title)
  plt.xlabel("Cascaded Layer")
  plt.ylabel(title)

  plt.legend()
  plt.show()

def task3():
  # Constants
  TRAINDIR = "/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE 66100/hw10/CarDetection/train"
  TESTDIR = "/Users/nikitaravi/Documents/Academics/Year 4/Semester 2/ECE 66100/hw10/CarDetection/test"
  IMAGESIZE = 40 * 20 # 40x20 image
  HEIGHT, WIDTH = 20, 40
  NUMCASCADES = 10
  NUMWEAKCLASSIFIERS = 20

  trainpositiveimages, trainpositivefilenames = getimages(os.path.join(TRAINDIR, "positive"))
  trainnegativeimages, trainnegativefilenames = getimages(os.path.join(TRAINDIR, "negative"))
  testpositiveimages, testpositivefilenames = getimages(os.path.join(TESTDIR, "positive"))
  testnegativeimages, testnegativefilenames = getimages(os.path.join(TESTDIR, "negative"))

  Xtrain, ytrain = createcardatasets(IMAGESIZE, trainpositiveimages,
```

```
    trainnegativeimages) # Xtrain: (800, 2468), ytrain: (2468,)
    Xtest, ytest = createcardatasets(IMAGESIZE, testpositiveimages, testnegativeimages) # Xtest: (800, 618), ytest: (618,)

    classifier = CascadedAdaBoost(NUMCASCADES, NUMWEAKCLASSIFIERS, IMAGESIZE, HEIGHT, WIDTH)
    classifier.fit(Xtrain, ytrain)
    classifier.settesting(Xtest, ytest)
    falsepositivetrain, falsepositivetest, falsenegativetrain, falsenegativetest, truepositivetrain, truepositivetest, accuracytrain, accuracytest = classifier.train()

    plottask3results(falsepositivetrain, falsepositivetest, "False Positive Rate")
    plottask3results(falsenegativetrain, falsenegativetest, "False Negative Rate")
    plottask3results(truepositivetrain, truepositivetest, "True Positive Rate")
    plottask3results(accuracytrain, accuracytest, "Accuracy Rate")

if name == " main ":
    print("=====TASK 1=====")
    pcaldaaccuracies = task1()

    print("=====TASK 2=====")
    autoencoderaccuracies = task2()
    plotaccuracies(pcaldaaccuracies, autoencoderaccuracies)

    print("=====TASK 3=====")
    task3()
```

Listing 1: The Source Code