# Purdue University

# ECE 661 Computer Vision

# Homework 11

**Submission:** Arjun Kramadhati Gopi

**Email:** akramadh@purdue.edu

## Tasks for this homework

There are two broad tasks for this homework:

1. Face recognition with PCA and LDA for dimensionality reduction and the nearest-neighbor rule for classification and object detection with a cascaded AdaBoost classifier.
2. Object Detection using cascaded AdaBoost implementation

Let us analyse each of the tasks in more detail.

## Face Recognition with PCA and LDA

### Broad view of the methodology

- Convert each image into gray scale and make sure they are 128X128 sized.
- Calculate the eigen vector of the covariance matrix.
- Use either PCA or LDa to project and create a p lower dimensional representation for the face images.
- We project the images onto these subspaces and create the feature vector representations.
- Using these, we classify the classes of the images by selecting the one nearest neighbor.
- We analyse the accuracy of both the PCA and LDA methods by plotting the variation in accuracya with respect to the variation in p values from 1 to 25.

**Principle Component Analysis (PCA)**
We use PCA for dimensionality reduction. We have about 630 images each in the test and training folders. The images are close up pictures of different faces in different angles. The pictures belong to 30 different people - each with 21 different pictures in the folder. We first convert the images into gray scale and make sure they are of 128X128 size. We then vectorize them in to an array of size 16384X1. After normalizing these vectorized images, we calculate the global mean:

$$m = \frac{1}{N} \sum_{i=1}^{i=630} x_i \tag{1}$$

We then can calculate the matrix X:

$$X = [x_1 - m | x_2 - m | ... | x_{630} - m]_{16384X630} \tag{2}$$

The covariance matrix is calculated as follows:

$$C = \frac{1}{N} X X^T \tag{3}$$

We need the eigen vectors $t_i$ of the covariance matrix. Instead of directly using equation 3 we take the eigen vectors $u_i$ for the matrix $XX^T$ thus obtaining the eigen vectors we need by doing:

$$t_i = X u_i \tag{4}$$

We obtained the normalised values:

$$\omega_i = \frac{t_i}{||t_i||} \tag{5}$$

Next we arrange these in descending order with respect to the corresponding eigen values. The vectors corresponding to the p largest eigen values of the $XX^T$ matrix are considered to create the W matrix:

$$W_p = [\omega_1|\omega_2|\omega_3|...|\omega_{p-1}|\omega_p]_{16384X630} \tag{6}$$

The idea is that for each image we create a mapping onto a lower p dimensional subspace to create the final feature vector representation unique to each image. This is done by:

$$y_i = W_p^T(x_i - m) \tag{7}$$

We use these projected vector representations to classify the classes of each image by using the nearest neighbor algorithm.

**Linear Discriminant Analysis (LDA)**

We know that the Fischer Discriminant is given by the function:

$$J(\omega_i) = \frac{\omega_j^T S_B \omega_j}{\omega_j^T S_W \omega_j} \tag{8}$$

Where:

- $S_B$ is the between class scatter value
- $S_W$ is the within class scatter value

Since in most cases the $S_W$ is singular we first convert the image into gray scale and then make sure it is 128X128 sized. Then we vectorize them into vectors of 16384X1 size. We calculate the global mean by ways of the equation described in equation 1. The class mean is then calculated by doing:

$$m_k = \frac{1}{||C_k||} \sum_{i=1}^{i=||C_k||} x_i \tag{9}$$

Using these values, we calculate the mean matrix as follows:

$$M = [m_1 - m|m_2 - m|m_3 - m|...|m_{C-1} - m|m_C - m]_{16384XC} \tag{10}$$

We need the eigen vector $t_i$ from the matrix:

$$S_B = \frac{1}{N} M M^T \tag{11}$$

Alternatively, we calculate the $u_i$ eigen vector of the matrix $MM^T$. Finally we calculate the final eigen vector by doing:

$$t_i = X u_i \tag{12}$$

After normalizing we form the matrix Y:

$$Y = [V_1|V_2|V_3|...|V_{C-1}|V_C]_{16384X630} \tag{13}$$

Next we estimate the Z vector:

$$Z = Y D_B^{\frac{-1}{2}} \tag{14}$$

Where $D_B$ is the eigen values of the matrix $S_B$. Our next task is to find the eigen vector of:

$$Z^T S_W Z = (Z^T X)(Z^T X)^T \tag{15}$$

Where X is

$$X = [x_1 1 - m_1 | x_1 2 - m_1 | ... | x_1 k - m_1 | ... | x_{C1} - m_C | ... | x_{Ck} - m_c] \tag{16}$$

We then arrange the eigen vectors in ascending order and we then select the vectors which have the smallest p eigen values. We use these normalised values to create the final projection vector representation of the images:

$$W_p = Z U_p \tag{17}$$

Therefore, finally for each image we create the feature representation vector:

$$y_i = W_p^T (x_i - m) \tag{18}$$

We use these representations to classify the classes of the image by getting closest match from the nearest neighbor classifier.

## Face Recognition Result Analysis

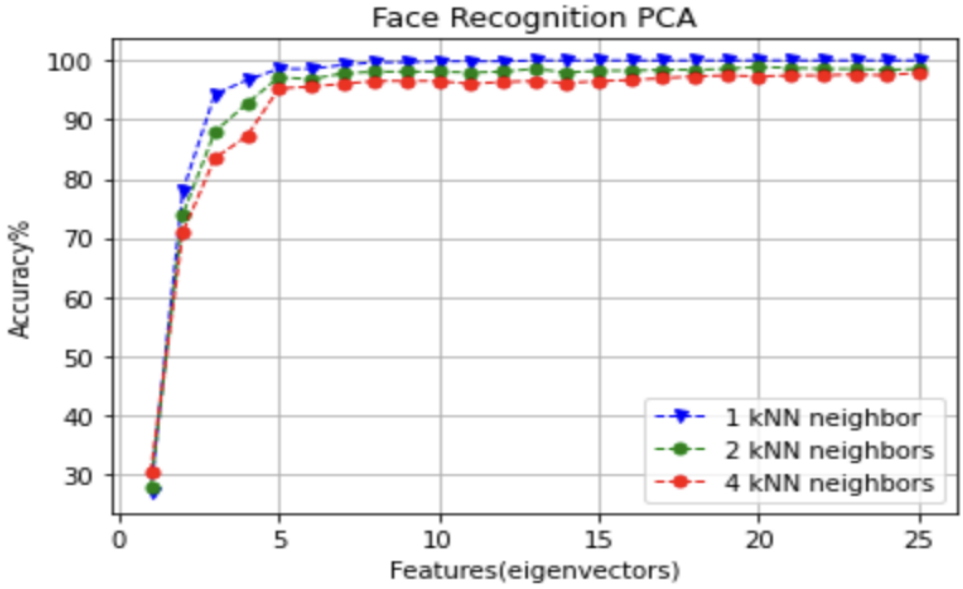


Figure 1: **PCA Results using kNN 1,2,4 neighbors**

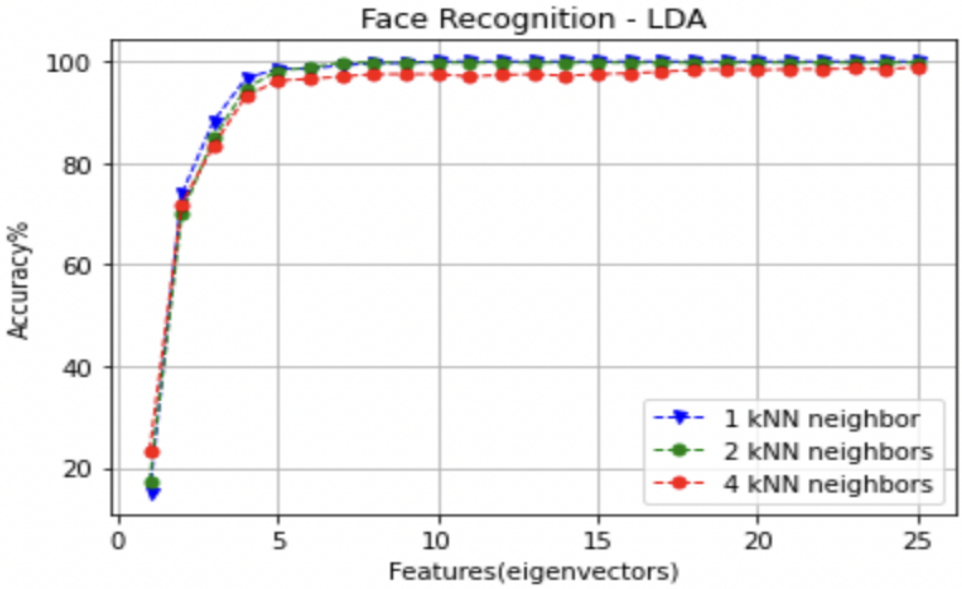


Figure 2: **LDA Results using kNN 1,2,4 neighbors**

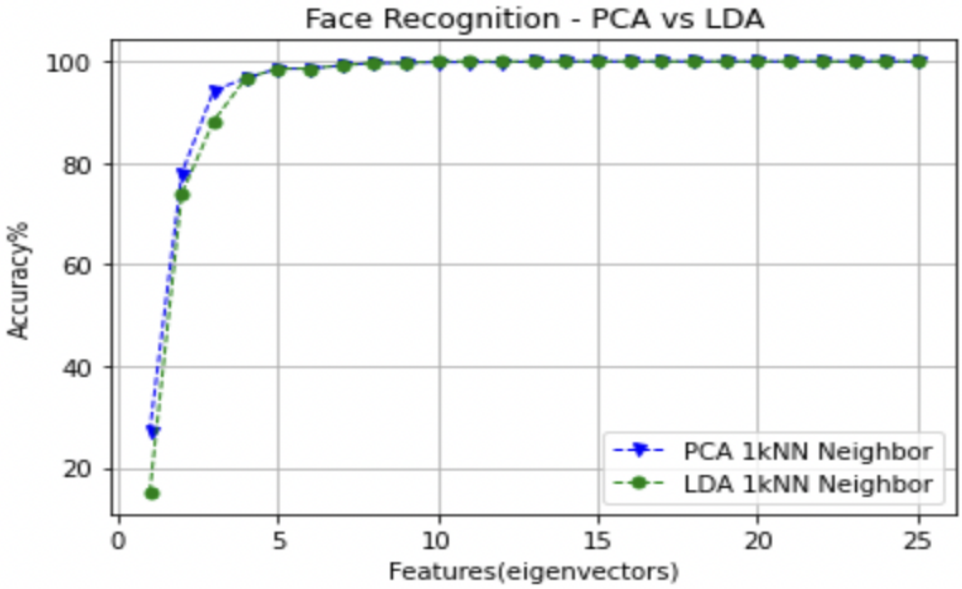


Figure 3: **PCA vs LDA for 1 kNN neighbor**

We see that for higher neighbor values, the accuracy tends to not converge to 1 as fast as the lower kNN niehgbor values. Therefore, having 1 kNN neighbor is the best for this application.
We can see how LDA tends to perform poorly compared with PCA at lower eigen vector numbers. But, LDA outperforms PCA when it comes to the pace at which it converges to an accuracy of 1 or 100 %. This means that LDA reaches full accuracy at lower dimensions when compared to PCA which reaches full accuracy at higher dimensions.

## Object Detection with AdaBoost cascaded classifier

The idea behind the AdaBoost classifier is to have multiple (cascaded) weak classifiers all boosting the final classification accuracy.

**Data Set**
The data consists of positive and negative folders. That is the positive folder has pictures of cars and the negative folder has pictures of non-car images. Each image is the size of 20X40.
**Feature Extraction**
In AdaBoost, we use the cascading power of multiple weak classifiers. The weak classifiers are classifiers which classify minute features. Of this we need a lot of minute features. To get these, we use the Haar filters to extract many features by modifying the size and orientations of the filter masks.
For the vertical directions we use 2X2,4X2,...,20X2 masks. For the horizontal directions we use 1X2,1X4,...,1X40 masks. We get a total of 11900 feature extractions.
**Training Procedure**

- Constructing the strong classifier:

  1. Create the weights of the samples with equally spaced distribution. If j and k are the number of negative and positive samples then the weights we create are:
  $$\frac{1}{2j}$$
  and
  $$\frac{1}{2k}$$
  We then normalise these weights.
  2. We construct the ordered list of the training samples arranged using the corresponding features. If $T^+$, $T^-$ are the total sum of positive and negative weights and if $S^+$ and $S^-$ are the sum of positive and negative sample weights below current samples respectively we calculate the error term using:
  $$\epsilon_t = min(S^+ + (T^- - S^-), S^- + (T^+ - S^+)) \tag{19}$$
  3. Once we have the minimum error term, the weak classifier is constructed as:
  $$h_t(x) = h(x, f_t, p_t, \theta_t) \tag{20}$$
  Where for the t iteration, $f_t$ is the feature used, $p_t$ is the polarity of the classifier, $\theta_t$ for the given minimum error term.
  4. The weight of the next term is then given by:
  $$\omega_{t+1,i} = \omega_{t,i}\beta_t^{1-e_i} \tag{21}$$
  Where $e_i$ is zero or one depending on whether the classification was correct or incorrect.
  5. We then check the classifier's effectiveness. That is it should achieve a true detection rate of at least one and the false positive rate of less than 0.5.
  6. We get the final strong classifier as:
  $$C(x) = \begin{cases} 1, & \sum_{t=1}^{T} \alpha_x h_t(x) \geq threshold \\ 0, & otherwise \end{cases} \tag{22}$$

- Cascading the strong classifiers
  1. We build the strong classifier as described in the previous section , dictated by equation 22.
  2. Ensure **FP** = 0
- Testing process flow
  1. We define one single resulting accumulated strong classifier as:
  2. Go through all the pictures to pick the best strong classifier described in step one.
- Performance Evaluation:
  1. False positive rate:

$$FP = \frac{Numberofmisclassifiednegativetestimages}{Numberofnegativetestimages} \tag{23}$$

  2. False negative rate:

$$FN = \frac{Numberofmisclassifiedpositivetestimages}{Numberofpositivetestimages} \tag{24}$$

## Object Detection Performance analysis of results

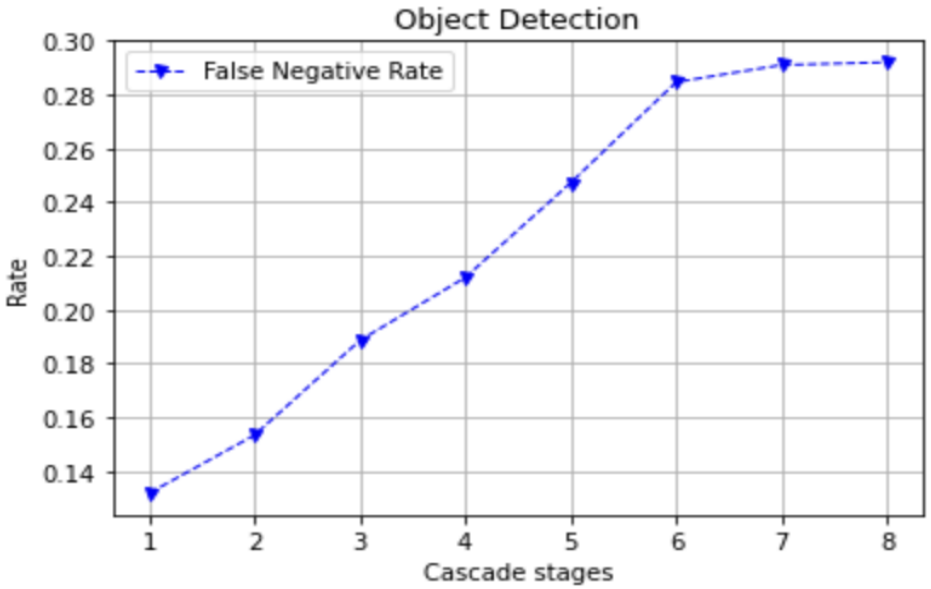


Figure 4: **False Negative Rate**

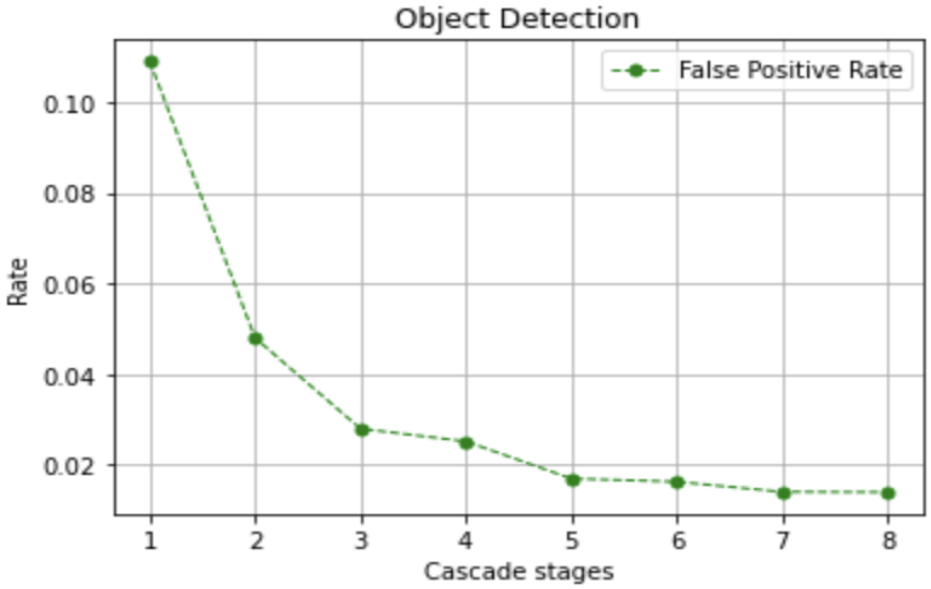


Figure 5: **False Positive Rate**

## Source Code

**These 2 codes are not entirely original. Many of the functions were written by referencing previous year solutions: Link**

```
1
```

```
"""
-------------------------------------
-------------------------------------
Computer Vision - Purdue University - Homework 11
-------------------------------------
Face Recognition & Object Detection
-------------------------------------
Author : Arjun Kramadhati Gopi, MS-Computer & Information Technology, Purdue University.
Date: Dec 2, 2020

Reference : https://engineering.purdue.edu/RVL/ECE661_2018/Homeworks/HW10/2BestSolutions/2.pdf
-------------------------------------
[TO RUN CODE]: python3 FaceRecognition.py
-------------------------------------
-------------------------------------
"""
import os
import pickle
import cv2 as cv
import numpy as np
from tqdm import tqdm
from sklearn.neighbors import KNeighborsClassifier

```

```
24
25 class RecognizeFace:
26
27     class LDA_Recog:
28         """
29         This is the class for Face Recognition using LDA
30         """
31         def __init__(self, dataset_path, params, thresh = 25,
               neighbors = 1):
32             """
33             Initialise the LDA Face Rec object
34             :param dataset_path: Path to two folders- Training,
                   Testing
35             :param params: Parameter list with class numbers and
                   sample numbers
36             :param thresh: Component cutoff
37             :param neighbors: Number of neighbors needed from the
                    KNN classifier
38             """
39             self.classes = params[0]
40             self.samples = params[1]
41             self.path = dataset_path
42             self.params = dict()
43             self.model_predictor = KNeighborsClassifier(
                   n_neighbors=neighbors)
44             self.thresh = thresh
45             self.params['Vectors'] = list()
46             self.params['Means'] = list()
47             for index, element in enumerate(tqdm(self.path, desc
                   ='Data Loading')):
48                 folder = os.listdir(element)
49                 folder.sort()
50                 print(element)
51                 temp = list()
52                 for image_name in folder:
53                     image_file = cv.imread(element+image_name)
54                     image_file = cv.cvtColor(image_file, cv.
                           COLOR_BGR2GRAY)
55                     image_file = image_file.reshape((1,-1))
56                     temp.append(image_file)
57                 mean, vector = self.process_data(vectors=temp,
                       sizeX=len(folder))
58                 self.params['Vectors'].append(vector)
59                 self.params['Means'].append(mean)
60
61         def scheduler(self):
62             """
63             This function handles all the functions in the right
                   sequence to complete
64             the task
65             :return:
66             """
67             self.compute()
```

```
68
69          def get_predictions(self, feature_set, label_set,
                test_set):
70              """
71              Fit the KNN model with the vector and get
                    preditctions on the test set
72              :param feature_set: Training feature vector
73              :param label_set: Labels
74              :param test_set: Test feature vector
75              :return: precitions
76              """
77              feature_set = np.nan_to_num(feature_set)
78              test_set = np.nan_to_num(test_set)
79              self.model_predictor.fit(feature_set.transpose(),
                    label_set)
80              return self.model_predictor.predict(test_set.
                    transpose())
81
82          def get_transpose(self, ss):
83              return ss.transpose()
84
85          def get_acc(self, label_list, prediction):
86              correct = np.zeros((len(label_list), 1))
87              result = list()
88              correct[prediction == label_list] = 1
89              result += [np.sum(correct) / len(label_list)]
90              return result
91
92          def get_results(self, vectorZ, u_value_w, label_list):
93              """
94              This function returns the accuracy or the score of
                    the model's
95              prection capability
96              :param vectorZ: Vector Z
97              :param u_value_w: decomposed u value
98              :param label_list: Labels
99              :return: Accuracy
100             """
101             for value_K in range(self.thresh):
102                 ss = vectorZ.dot(u_value_w[: , :value_K+1])
103                 ss = ss/np.linalg.norm(ss, axis=0)
104                 value = self.get_transpose(ss=ss)
105                 trFeatures = np.dot(value, self.params['Vectors
                        '][0]-self.params['Means'][0][:, None])
106                 tsFeatures = np.dot(value, self.params['Vectors
                        '][1]-self.params['Means'][1][:, None])
107                 prediction = self.get_predictions(feature_set=
                        trFeatures, label_set=label_list, test_set=
                        tsFeatures)
108                 result = self.get_acc(label_list=label_list,
                        prediction=prediction)
109             return result
110
```

```
111         def get_mean_vector(self, vector):
112             """
113             Function to get the mean vectors for LDA.
114             :param vector: Initial image vector
115             :return: Mean vector, difference vector and the
                    vector B
116             """
117             vec_difference = np.zeros(vector.shape)
118             vec_mean = np.zeros((vector.shape[0], self.classes))
119             for component in range(self.classes):
120                 vec_mean[:, component] = np.mean(vector[:,
                        component * self.samples:component * self.
                        samples],
121                                                  axis=1)
122                 vec_difference[:, component*self.samples+1:(
                        component)*self.samples]=vector[:, component*
                        self.samples+1:(component)*self.samples]-
                        vec_mean[:,component, None]
123             valLst = vec_mean-self.params['Means'][0][:, None]
124             return vec_mean, vec_difference, valLst
125
126         def get_label_vector(self):
127             """
128             Get label list
129             :return: List of the labels
130             """
131             label_list = list()
132             for class_number in range(self.classes):
133                 label_list.extend(np.ones((self.samples,1))
                        [:,0]*(class_number+1))
134             return np.asarray(label_list)
135
136         def process_data(self, vectors, sizeX):
137             """
138             Get initial image vectors and their mean
139             :param vectors: Image vector
140             :param sizeX: Number of images in the directory
141             :return: Mean, image vector
142             """
143             vectors = (np.asarray(vectors).reshape((sizeX,-1)))
144             vectors = vectors.transpose()
145             vectors = vectors/np.linalg.norm(vectors, axis=0)
146             value_mean = np.mean(vectors, axis=1)
147             return value_mean, vectors
148
149         def get_sorted_du(self, d_value, u_value):
150             index = np.argsort(-1 * d_value)
151             d_value = d_value[index]
152             u_value = u_value[:, index]
153             return d_value, u_value
154
155         def get_entry_value(self, d_value):
156             return np.eye(self.classes)*(d_value**(-0.5))
```

```
157
158         def compute(self):
159             """
160             This function computes and executes the face
                    recognition
161             using LDA.
162             :return:
163             """
164             label_list = self.get_label_vector()
165             mean, difference, mean_two = self.get_mean_vector(
                    vector=self.params['Vectors'][0])
166             d_value, u_value = np.linalg.eig(mean_two.transpose()
                    .dot(mean_two))
167             d_value, u_value = self.get_sorted_du(d_value=d_value
                    ,u_value=u_value)
168             eigenvector = mean_two.dot(u_value)
169             entry_value = self.get_entry_value(d_value=d_value)
170             vectorZ = eigenvector.dot(entry_value)
171             vectorX = np.dot(vectorZ.transpose(),difference)
172             vectorX = np.nan_to_num(vectorX)
173             d_value_w, u_value_w = np.linalg.eig(vectorX.dot(
                    vectorX.transpose()))
174             _, u_value_w = self.get_sorted_du(d_value=d_value,
                    u_value=u_value_w)
175             result = self.get_results(vectorZ, u_value_w,
                    label_list)
176             file = open('result_lda.obj', 'wb')
177             pickle.dump(result, file)
178             print('LDA Face Recognition complete')
179
180     class PCA_Reccog:
181         """
182         This is the class to perform Face Recognition using PCA
183         """
184         def __init__(self, dataset_path, features, labels,
                neighbors=4):
185             """
186             Initialise the object for PCA Face Recognition
187             :param dataset_path: Path to the
188             :param features: Size of feature vector
189             :param labels: Size of label vector
190             :param neighbors: Number of neighbors needed from the
                     KNN predictor
191             """
192             self.parameter_dict = dict()
193             self.parameter_dict['Labels_Train'] = np.zeros((
                    labels))
194             self.parameter_dict['Labels_Test'] = np.zeros((labels
                    ))
195             self.parameter_dict['Features_Train'] = np.zeros(
                    features)
196             self.parameter_dict['Features_Test'] = np.zeros(
                    features)
```

```
197             self.path = dataset_path
198             self.model_classify = KNeighborsClassifier(
                    n_neighbors=neighbors)
199
200         def scheduler(self):
201             """
202             This function runs all the required functions in the
                    right order
203             :return:
204             """
205             self.prepare_data()
206             self.commence_analysis()
207
208         def add_features(self, idx, index, image_file):
209             """
210             Build the features list for both training and testing
                     data set
211             :param idx: Identifier - 0 for train; 1 for test
212             :param index: Image index
213             :param image_file: Image read by opencv
214             :return: None
215             """
216             if idx == 0:
217                 self.parameter_dict['Features_Train'][index] =
                        image_file
218             elif idx == 1:
219                 self.parameter_dict['Features_Test'][index] =
                        image_file
220
221         def add_labels(self, idx, index, filename):
222             """
223             Build the label list for both training and testing
                    data set
224             :param idx: Identifier - 0 for train; 1 for test
225             :param index: Image index
226             :param image_file: Image read by opencv
227             :return: None
228             """
229             if idx == 0:
230                 self.parameter_dict['Labels_Train'][index] = int(
                        filename[0]+filename[1])
231             elif idx == 1:
232                 self.parameter_dict['Labels_Test'][index] = int(
                        filename[0]+filename[1])
233
234         def prepare_data(self):
235             """
236             This function prepares the data by reading,
                    flattening and organizing all the images from
237             both the datasets.
238             :return: None
239             """
240             for idx, element in enumerate(self.path):
```

```
241                 index = 0
242                 print(element)
243                 for filename in os.listdir(element):
244                     image_file = cv.imread(element+filename)
245                     image_file = cv.cvtColor(image_file, cv.
                            COLOR_BGR2GRAY)
246                     image_file = image_file.flatten()
247                     self.add_features(idx=idx, index=index,
                            image_file=image_file)
248                     self.add_labels(idx=idx, index=index,
                            filename=filename)
249                     index+=1
250             print('Data prep complete')
251
252         def get_eigen_terms(self, vector):
253             """
254             Get eigen terms
255             :param vector: Image vector
256             :return: eigen value and eigen vector
257             """
258             value, vector = np.linalg.eigh(np.matmul(vector.T,
                    vector))
259             index = np.argsort(-1*value)
260             return value, vector[:, index]
261
262         def get_W(self, vector, eigen_vector):
263             """
264             Get the W matrix
265             :param vector: Image vector
266             :param eigen_vector: Eigen vector
267             :return: W matrix
268             """
269             W_matrix = np.matmul(vector, eigen_vector)
270             W_matrix = W_matrix/np.linalg.norm(W_matrix, axis=0)
271             return W_matrix
272
273         def get_prediction(self, vector):
274             """
275             Get prediction from the model
276             :param vector: Testing feature vector
277             :return: Predictions
278             """
279             print(self.model_classify.predict(vector.T))
280             return self.model_classify.predict(vector.T)
281
282         def get_featurers(self, W_value, vector, component_number
                ):
283             return np.matmul(W_value[:, :component_number+1].T,
                    vector)
284
285         def get_results(self, guess):
286             """
287             Get the score for the model
```

```
288             :param guess: Prediction
289             :return: Accuracy
290             """
291             test_value = self.parameter_dict['Labels_Test']
292             return ((guess==test_value).sum()/630*100)
293
294         def commence_analysis(self):
295             """
296             This function executes the PCA Face Recognition
                    pipeline
297             :return: None
298             """
299             value_Train, mu_value_Train, vector_Train = self.
                    compute_mean(value=self.parameter_dict['
                    Features_Train'].copy())
300             eigen_value, eigen_vector = self.get_eigen_terms(
                    vector=vector_Train)
301             self.parameter_dict['Features_Train_Mod'] =
                    value_Train
302             value_Test, mu_value_Test, vector_Test = self.
                    compute_mean(value=self.parameter_dict['
                    Features_Test'].copy())
303             self.parameter_dict['Features_Test_Negated'] =
                    value_Test
304             W_value = self.get_W(vector=vector_Train,
                    eigen_vector=eigen_vector)
305             temp = list()
306             for component in range(25):
307                 trainingF = self.get_featurers(W_value=W_value,
                        vector=vector_Train, component_number=
                        component)
308                 testingF = self.get_featurers(W_value=W_value,
                        vector=vector_Test, component_number=component
                        )
309                 self.model_classify.fit(trainingF.T, self.
                        parameter_dict['Labels_Train'])
310                 guess_value = self.get_prediction(vector=testingF
                        )
311                 accuracy = self.get_results(guess=guess_value)
312                 temp.append(accuracy)
313                 print(accuracy)
314             print(temp)
315             file = open('result_pca.obj', 'wb')
316             pickle.dump(temp, file)
317
318         def compute_mean(self, value):
319             """
320             Get the mean vectors
321             :param value: Initial image file
322             :return: modified image vector, mean value and the
                    difference vector
323             """
324             value = np.transpose(value)
```

```
            value = value/np.linalg.norm(value, axis=0)
            mu_value = np.mean(value, axis=1)
            vector = value - mu_value[:, None]
            return value, mu_value, vector


if __name__ == "__main__":
    """
    Code starts here
    """
    tester = RecognizeFace.PCA_Reccog(['ECE661_2020_hw11_DB1/
       train/','ECE661_2020_hw11_DB1/test/'], features
       =(630,16384), labels=630)
    tester.scheduler()
    tester = RecognizeFace.LDA_Recog(['ECE661_2020_hw11_DB1/train
       /','ECE661_2020_hw11_DB1/test/'], params=(30,21))
    tester.scheduler()
```

```

"""
-------------------------------------
-------------------------------------
Computer Vision - Purdue University - Homework 11
-------------------------------------
Face Recognition & Object Detection
-------------------------------------
Author : Arjun Kramadhati Gopi, MS-Computer & Information
   Technology, Purdue University.
Date: Dec 2, 2020

Reference : https://engineering.purdue.edu/RVL/ECE661_2018/
   Homeworks/HW10/2BestSolutions/2.pdf

-------------------------------------
[TO RUN CODE]: python3 ObjectDetection.py
-------------------------------------
-------------------------------------
"""
import os
import pickle
import cv2 as cv
import numpy as np
from tqdm import tqdm


class ViolaJonesOD:
    """
    Main class to perform Object Detection using the Viola Jones
       Algorithm
    """
    def __init__(self, folder_locations):
        """
```

```
31          Initialise the object with the folder locations of the
               data sets
32          :param folder_locations:
33          """
34          self.folders = folder_locations
35
36      def scheduler(self):
37          """
38          This function runs all the required functions to get the
               Object Detection network
39          running
40          :return:
41          """
42          tester = self.GetFeatures(data_path=[self.folders[0],
               self.folders[1]],type='train.obj')
43          tester.scheduler()
44          tester = self.GetFeatures(data_path=[self.folders[2],
               self.folders[3]],type='test.obj')
45          tester.scheduler()
46          obj = self.getClassifier()
47          trainadaboost = self.AdaBoostTrain(obj=obj,feature_path='
               train.obj')
48          trainadaboost.scheduler()
49          testadaboost = self.AdaBoostTest(obj=obj, feature_path='
               test.obj', model_path='classifier.obj')
50          testadaboost.scheduler()
51
52      class getClassifier:
53          """
54          This is the class to get the classifier for the cascaded
               AdaBoost approach
55          """
56
57          def get_weight(self, samplesP, samplesN):
58              """
59              Get weights for the given sample set
60              :param samplesP: Positives
61              :param samplesN: Negatives
62              :return: weights
63              """
64              return np.concatenate((np.ones((1,samplesP))*0.5/
                   samplesP, np.ones((1,samplesN))*0.5/samplesN),
                   axis=1)
65
66          def get_labels(self, samplesP, samplesN):
67              """
68              Prepare and get the labels
69              :param samplesP: Positives
70              :param samplesN: Negatives
71              :return: Labels
72              """
73              return np.concatenate((np.ones((1,samplesP)), np.
                   zeros((1, samplesN))) , axis=1)
```

```
74
75          def sort_WL(self, W, L, vector):
76              """
77              Sort the weights and labels
78              :param W: Updated weights
79              :param L: Labels
80              :param vector: Concatenated positive and negative
                    sample vector
81              :return:Sorted values
82              """
83              sortW = np.tile(W, (len(vector),1))
84              sortL = np.tile(L, (len(vector),1))
85              return sortW, sortL
86
87          def getSum(self, W, ps, sL, sW):
88              """
89              Get the sum for the sorted values
90              :param W: weights
91              :param ps: Positive samples
92              :param sL: Sorted Labels
93              :param sW: Sorted Weights
94              """
95              tnW = np.sum(W[:, ps:])
96              tpW = np.sum(W[:,:ps])
97              spW = np.cumsum(sW * sL, axis=1)
98              snW = np.cumsum(sW, axis=1) - spW
99              return spW, snW, tnW, tpW
100
101         def get_error_terms(self, vector, spW, snW, tnW, tpW):
102             """
103             Get the required error terms.
104             :return: Index of minimum error, minimum error value,
                     error vector
105             """
106             error = np.zeros((vector.shape[0], vector.shape[1],
                    2))
107             error[:, :, 1] = snW + tpW - spW
108             error[:, :, 0] = spW + tnW - snW
109             index = np.unravel_index(np.argmin(error), error.
                    shape)
110             errro_min = error[index]
111             return index, errro_min, error
112
113         class StrongC:
114             """
115             Strong classifier object
116             """
117             index = list()
118             weak_number = 0
119             classifierT = list()
120
121         def getWeakC(self, vector, positive_samples,
                    negative_samples):
```

```
122             """
123             Get the wek classifier which can extract low level
                    image features
124             :param vector: Concatenated positive and negative
                    samples vector
125             :param positive_samples: Positive samples
126             :param negative_samples: Negative samples
127             :return: Classifier object
128             """
129             cl = list()
130             cl_T = list()
131             alpha_value = list()
132             W = self.get_weight(samplesP=positive_samples,
                    samplesN=negative_samples)
133             L = self.get_labels(samplesP=positive_samples,
                    samplesN=negative_samples)
134             obj = self.StrongC()
135             for number in range(25):
136                 W = W/np.sum(W)
137                 sortW, sortL = self.sort_WL(W,L, vector)
138                 index = np.argsort(vector, axis=1)
139                 row_value = np.arange(len(vector)).reshape((-1,1)
                        )
140                 sortL = sortL[row_value, index]
141                 sortW = sortW[row_value, index]
142                 spW, snW, tnW, tpW = self.getSum(W=W, ps=
                        positive_samples, sL = sortL, sW=sortW)
143                 index_e, errro_min, error = self.get_error_terms(
                        vector=vector, spW=spW, snW = snW, tnW=tnW,
                        tpW=tpW)
144                 f_value = index_e[0]
145                 index_S = index[f_value,:]
146                 pt = np.zeros((vector.shape[1],1))
147                 p_matrix = np.zeros((vector.shape[1],1))
148                 if index_e[2] == 0:
149                     value_p = -1
150                     pt[index_e[1]+1:] = 1
151                 else:
152                     value_p = 1
153                     pt[:index_e[1]+1] = 1
154                 p_matrix[index_S]= pt
155                 sortV = vector[f_value,:]
156                 sortV = sortV[index_S]
157                 if index_e[1]==0:
158                     angle = sortV[0]-0.01
159                 elif index_e[1]==-1:
160                     angle = sortV[-1]+0.01
161                 else:
162                     angle = np.mean(sortV[index_e[1]-1:index_e
                            [1]+1])
163                 beta_value = errro_min/(1-errro_min)
164                 alpha_value.append(np.log(1/beta_value))
165                 cl.append(p_matrix.transpose())
```

```
166                 cl_T.append([f_value, angle, value_p, np.log(1/
                        beta_value)])
167                 W = W*(beta_value**(1-np.abs(L-p_matrix.transpose
                        ())))
168                 s_value = np.dot(np.asarray(cl).transpose(),np.
                        asarray(alpha_value))
169                 angle_updated = np.min(s_value[:positive_samples
                        ])
170                 prediction_s = np.zeros(s_value.shape)
171                 prediction_s[s_value>=angle_updated]=1
172                 if (np.sum(prediction_s[positive_samples:])/
                        negative_samples<0.5):
173                     break
174             index_new = list()
175             index_new.extend(np.arange(positive_samples))
176             wrong_negative_index = [positive_samples+x for x in
                    range(negative_samples) if prediction_s[
                    positive_samples+x]==1]
177             index_new.extend(wrong_negative_index)
178             obj.index = np.asarray(index_new)
179             obj.weak_number = number+1
180             obj.classifierT = cl_T
181             return obj
182
183     class AdaBoostTest:
184         """
185         Class to run AdaBoost Testing
186         """
187         def __init__(self, obj, feature_path, model_path):
188             """
189             Initialise the AdaBoost Tester object
190             :param obj: Classifier object
191             :param feature_path: Features object from training
192             :param model_path: classifier object from training
193             """
194             self.parameter_dict = dict()
195             self.object = obj
196             self.feature_path = feature_path
197             file = open(feature_path, 'rb')
198             file_value = pickle.load(file)
199             self.positive = file_value[0]
200             self.negative = file_value[1]
201             self.sampleP = self.positive
202             self.sampleN = self.negative
203             file.close()
204             file = open(model_path, 'rb')
205             self.model = pickle.load(file)
206             file.close()
207
208         def scheduler(self):
209             """
210             This function runs all the required functions in
                    order
```

```
211             to get the task done
212             """
213             self.process_data()
214             self.commence_testing()
215
216         def get_predicted_angle(self, angle):
217             return 0.5*np.sum(angle)
218
219         def get_vector(self):
220             """
221             Get the required vector to compute weights and
                    parameters.
222             :return: Vector
223             """
224             return np.concatenate((self.sampleP, self.sampleN),
                    axis = 1)
225
226         def get_f_value(self, classifier_T):
227             return classifier_T[:,0].astype(int)
228
229         def get_weight_pred(self, value, wpred ):
230             weight_T = (value[1] * value[0])[:, None] - value
                    [1][:, None] * value[2]
231             wpred[weight_T >= 0] = 1
232             return wpred
233
234         def update_fp_fn(self, ftp_one, ftp_two, number_wrongP,
                number_rightP):
235             ftp_one.append(number_wrongP / self.parameter_dict['
                    Positives_WHL'])
236             ftp_two.append(
237                 (self.parameter_dict['Negatives_WHL'] -
                        number_rightP) / self.parameter_dict['
                        Negatives_WHL'])
238             return ftp_one,ftp_two
239
240         def get_sample_lengths(self, spred):
241             return [x for x in range(self.parameter_dict['
                    Positives']) if spred[x]==1], [x for x in range(
                    self.parameter_dict['Negatives']) if spred[x+self.
                    parameter_dict['Positives']]==1]
242
243         def get_angle_params(self, classifier_T):
244             return classifier_T[:,1], classifier_T[:,2],
                    classifier_T[:,3]
245
246         def process_data(self):
247             """
248             Process the data before commencing testing into
249             dictionary entries.
250             """
251             self.parameter_dict['Negatives'] = self.negative.
                    shape[1]
```

```
252             self.parameter_dict['Positives'] = self.positive.
                    shape[1]
253             self.parameter_dict['Positives_WHL'] = self.positive.
                    shape[1]
254             self.parameter_dict['Negatives_WHL'] = self.negative.
                    shape[1]
255             print('Positive Samples:')
256             print(self.parameter_dict['Positives'])
257             print('Negative Samples:')
258             print(self.parameter_dict['Negatives'])
259
260         def get_classifier_T(self, com_value):
261             return np.asarray(self.model[com_value].classifierT)
262
263         def commence_testing(self):
264             """
265             This function runs all the required items to execute
                    the AdaBoost testing process.
266             """
267             number_wrongP, number_rightP = 0,0
268             ftp_one, ftp_two = [], []
269             for com in range(len(self.model)):
270                 print('Samples positive: ' +str(self.
                        parameter_dict['Positives_WHL']))
271                 print('Samples negative: ' +str(self.
                        parameter_dict['Negatives_WHL']))
272                 vector = self.get_vector()
273                 classifier_T = self.get_classifier_T(com_value=
                        com)
274                 f_value = self.get_f_value(classifier_T=
                        classifier_T)
275                 param_1, param_2, param_3 = self.get_angle_params
                        (classifier_T=classifier_T)
276                 angle_predicted = self.get_predicted_angle(angle=
                        param_3)
277                 wpred = np.zeros((len(classifier_T),vector.shape
                        [1]))
278                 tempF = vector[f_value,:]
279                 wpred = self.get_weight_pred(value=(param_1,
                        param_2,tempF), wpred=wpred)
280                 spred = np.zeros((vector.shape[1],1))
281                 tempS = np.dot(wpred.transpose(), param_3)
282                 spred[tempS>=angle_predicted]=1
283                 pcIdX, neIdX = self.get_sample_lengths(spred=
                        spred)
284                 print('Right samples + '+ str(pcIdX))
285                 print('Error samples - '+str(neIdX))
286                 number_wrongP = number_wrongP + (self.
                        parameter_dict['Positives']-len(pcIdX))
287                 number_rightP =number_rightP + (self.
                        parameter_dict['Negatives']-len(neIdX))
288                 ftp_one, ftp_two = self.update_fp_fn(ftp_one=
                        ftp_one, ftp_two=ftp_two, number_wrongP=
```

```
                    number_wrongP, number_rightP=number_rightP)
289                 self.sampleP = self.sampleP[:, pcIdX]
290                 self.sampleN = self.sampleN[:, neIdX]
291                 self.parameter_dict['Positives'] = len(pcIdX)
292                 self.parameter_dict['Negatives'] = len(neIdX)
293             list = [self.parameter_dict, ftp_one, ftp_two]
294             file = open('AdaBoost_Testing_Results.obj', 'wb')
295             pickle.dump(list, file)
296             print('AdaBoost Testing complete')
297
298     class AdaBoostTrain:
299         """
300         This class is for the Training of the AdaBoost Object
               Detection network
301         """
302         def __init__(self, obj, feature_path):
303             """
304             Initialise object
305             :param obj: Classifier object
306             :param feature_path: Path to the features extracted
                   from the training set
307             """
308             self.parameter_dict = dict()
309             self.object = obj
310             self.feature_path = feature_path
311             file = open(feature_path, 'rb')
312             file_value = pickle.load(file)
313             self.positive = file_value[0]
314             self.negative = file_value[1]
315             self.classifier = list()
316             file.close()
317
318         def scheduler(self):
319             """
320             This function runs all the required function in the
                   correct order.
321             """
322             self.process_data()
323             self.commence_training()
324
325         def process_data(self):
326             """
327             Process and organise the data before training
328             """
329             self.parameter_dict['Positives'] = self.positive.
                   shape[1]
330             self.parameter_dict['Negatives'] = self.negative.
                   shape[1]
331             self.parameter_dict['Negatives_WHL'] = self.
                   parameter_dict['Negatives']
332             print('Initial Positive Samples:')
333             print(self.parameter_dict['Positives'])
334             print('Initial Negative Samples:')
```

```
335             print(self.parameter_dict['Negatives'])
336
337         def get_vector(self):
338             """
339             Get the required vector to compute weights and
                    parameters.
340             :return: Vector
341             """
342             return np.concatenate((self.positive, self.negative),
                    axis = 1)
343
344         def commence_training(self):
345             """
346             This function takes care of the AdaBoost Training
                    process
347             :return: Save the trained feature model
348             """
349             vector = self.get_vector()
350             classifier_list = list()
351             for com in range(8):
352                 wc = self.object.getWeakC(vector=vector,
                        positive_samples=self.parameter_dict['
                        Positives'],negative_samples=self.
                        parameter_dict['Negatives'])
353                 classifier_list.append(wc)
354                 if (len(wc.index)==self.parameter_dict['Positives
                        ']):
355                     break
356                 negatives = len(wc.index)-self.parameter_dict['
                        Positives']
357                 self.parameter_dict['Negatives'] = negatives
358                 vector = vector[:,wc.index]
359                 val_to_apnd = self.parameter_dict['Negatives']/
                        self.parameter_dict['Negatives_WHL']
360                 self.classifier.append(val_to_apnd)
361             db = open('classifier.obj','wb')
362             pickle.dump(classifier_list, db)
363             print('Model saved. Training complete')
364
365     class GetFeatures:
366         """
367         This class is to extract features from the testing and
                training directories
368         """
369         def __init__(self, data_path, type):
370             """
371             Initialise the object to extract the features
372             :param data_path: Path to the directory
373             :param type: Testing or Training to store them
                    accordingly
374             """
375             self.filename = type
376             self.feature_list = list()
```

```
377             self.image_path = data_path
378             self.positive_path = os.listdir(self.image_path[0])
379             self.positive_path.sort()
380             self.negative_path = os.listdir(self.image_path[1])
381             self.negative_path.sort()
382             self.reference_image_positive = cv.imread(self.
                    image_path[0] + self.positive_path[0])
383             self.reference_image_negative = cv.imread(self.
                    image_path[1] + self.negative_path[0])
384             self.image_vector_dict = list()
385             self.image_vector_dict.append(np.zeros(
386                 (self.reference_image_positive.shape[0], self.
                        reference_image_positive.shape[1], len(self.
                        positive_path))))
387             self.image_vector_dict.append(np.zeros(
388                 (self.reference_image_negative.shape[0], self.
                        reference_image_negative.shape[1], len(self.
                        negative_path))))
389             self.paths = [self.positive_path, self.negative_path]
390             self.ref_images = [self.reference_image_positive,
                    self.reference_image_negative]
391             self.get_images_ready()
392
393         def scheduler(self):
394             """
395             This function runs all the required functions in
                    order
396             :return:
397             """
398             self.extract_features()
399
400         def get_images_ready(self):
401             """
402             Organize the images
403             """
404             for index, path in enumerate(tqdm(self.paths, desc='
                    Image Load')):
405                 for value in range(len(path)):
406                     ref_img = cv.imread(self.image_path[index]+
                            path[index])
407                     self.image_vector_dict[index][:,:, index] =
                            cv.cvtColor(ref_img, cv.COLOR_BGR2GRAY)
408
409         def set_filter_size(self, value):
410             """
411             Get filter size
412             :param value: Value for the shape
413             :return: Filter size
414             """
415             return (value+2)*2
416
417         def get_cumulative_sum(self, image):
418             value = np.cumsum(image, axis=1)
```

```
419             return np.cumsum(image, axis=0)
420
421         def get_sum_of_box(self, points, integral):
422             """
423             Box sum function
424             :param points: Corner points
425             :param integral: Integral image
426             :return: Sum of the box
427             """
428             left_top = integral[np.int(points[0][0])][np.int(
                    points[0][1])]
429             right_top = integral[np.int(points[1][0])][np.int(
                    points[1][1])]
430             right_bottom = integral[np.int(points[2][0])][np.int(
                    points[2][1])]
431             left_bottom = integral[np.int(points[3][0])][np.int(
                    points[3][1])]
432             return left_bottom-right_top-right_bottom+left_top
433
434         def get_integral_image(self):
435             """
436             Get the integral image
437             :return: Integral image list
438             """
439             temp = list()
440             for index in range(2):
441                 integral = self.get_cumulative_sum(image=self.
                        image_vector_dict[index])
442                 integral = np.concatenate((np.zeros((self.
                        ref_images[index].shape[0],1,len(self.paths[
                        index]))),integral), axis=1)
443                 integral = np.concatenate((np.zeros((1,self.
                        ref_images[index].shape[1]+1,len(self.paths[
                        index]))),integral), axis=0)
444                 temp.append(integral)
445             return temp
446
447         def get_points(self, value, value_two, mask, type):
448             """
449             Get the required points for the box
450             :param value: First value (row)
451             :param value_two: Second value (Column)
452             :param mask: Filter or mask size
453             :param type: Type of points
454             :return: Corner points
455             """
456             if type ==1:
457                 points = list()
458                 points.append([value, value_two])
459                 points.append([value, value_two + mask / 2])
460                 points.append([value + 1, value_two])
461                 points.append([value + 1, value_two + mask / 2])
462                 return points
```

```
463                 elif type ==2:
464                     points = list()
465                     points.append([value, value_two + mask / 2])
466                     points.append([value, value_two + mask])
467                     points.append([value+1, value_two + mask / 2])
468                     points.append([value+1, value_two + mask])
469                     return points
470                 elif type ==3:
471                     points = list()
472                     points.append([value, value_two])
473                     points.append([value, value_two + 2])
474                     points.append([value+mask/2, value_two])
475                     points.append([value+mask/2, value_two + 2])
476                     return points
477                 elif type ==4:
478                     points = list()
479                     points.append([value+mask/2, value_two])
480                     points.append([value+mask/2, value_two + 2])
481                     points.append([value+mask, value_two])
482                     points.append([value+mask, value_two + 2])
483                     return points
484 
485         def get_diff(self, tuple):
486             return (tuple[1] - tuple[0]).reshape((1, -1))
487 
488         def add_feature(self, feature):
489             feature = np.asarray(feature).reshape((len(feature)
                        ,-1))
490             self.feature_list.append(feature)
491 
492         def save_features(self, feature_list):
493             assert len(feature_list) == 2
494             db = open(self.filename, 'wb')
495             pickle.dump(feature_list, db)
496             print('feature list saved')
497 
498         def extract_features(self):
499             integral_list = self.get_integral_image()
500             for index in tqdm(range(2), desc='Feature Extraction
                        '):
501                 temp_features = list()
502                 shape_one = self.ref_images[index].shape[1]
503                 shape_zero = self.ref_images[index].shape[0]
504                 for value_n in range(np.int(shape_one / 2)):
505                     mask = self.set_filter_size(value=value_n)
506                     criteria = [np.int(shape_one / 2), shape_zero
                                , shape_one + 1 - mask, np.int(shape_zero
                                / 2),
507                                 shape_zero + 1 - mask, shape_one
                                    + 1 - 2]
508                     for value in range(criteria[1]):
509                         for value_two in range(criteria[2]):
510                             points = self.get_points(value=value,
```

```
                                    value_two=value_two, mask=mask,
                                   type=1)
511                                first_SB = self.get_sum_of_box(points
                                   =points, integral=integral_list[
                                   index])
512                                points = self.get_points(value=value,
                                    value_two=value_two, mask=mask,
                                   type=2)
513                                second_SB = self.get_sum_of_box(
                                   points=points, integral=
                                   integral_list[index])
514                                store_value = self.get_diff(tuple=(
                                   first_SB, second_SB))
515                                temp_features.append(store_value)
516                    for value_n in range(criteria[3]):
517                        mask = self.set_filter_size(value=value_n)
518                        for value in range(criteria[4]):
519                            for value_two in range(criteria[5]):
520                                points = self.get_points(value=value,
                                    value_two=value_two, mask=mask,
                                   type=3)
521                                first_SB = self.get_sum_of_box(points
                                   =points, integral=integral_list[
                                   index])
522                                points = self.get_points(value=value,
                                    value_two=value_two, mask=mask,
                                   type=4)
523                                second_SB = self.get_sum_of_box(
                                   points=points, integral=
                                   integral_list[index])
524                                store_value = self.get_diff(tuple=(
                                   first_SB, second_SB))
525                                temp_features.append(store_value)
526                    self.add_feature(feature=temp_features)
527                self.save_features(feature_list=self.feature_list)
528
529
530 if __name__ == "__main__":
531     """
532     Code starts here
533     """
534     tester = ViolaJonesOD(['ECE661_2020_hw11_DB2/train/positive
           /','ECE661_2020_hw11_DB2/train/negative/','
           ECE661_2020_hw11_DB2/test/positive/','ECE661_2020_hw11_DB2
           /test/negative/'])
535     tester.scheduler()
```