ECE 661 – Homework 10

Ran Xu

xu943@purdue.edu

12/5/2018

1. Math Reductions and Algorithms

In this homework, we want to recognize faces and detect cars.

1.1 Face recognition with PCA

To recognize faces, we have a training set with 30 people, each with 21 photos and a test set with the same setting. The general approach is to generate a projection matrix based on training set, project training images to feature space, and finally project test images also to feature space and match with all training images to find the nearest neighbor as classification results. The algorithm is as follows,

Step 1: Construct Co-variance matrix C

Denote the training set as $(x_i, y_i), i = 1, \dots, M$, where x_i is the flattened vector representation of the i-th gray-scale face and y_i is the ground truth label of this face. Denote the mean face as,

$$\bar{m} = \frac{1}{M}\sum_{i=1}^{M} x_i$$

Denote the standard deviation of each face x_i as s_i.We construct the normalized face matrix as,

$$X = \begin{bmatrix} \frac{x_1 - \bar{m}}{s_1} & \frac{x_2 - \bar{m}}{s_2} & \dots & \frac{x_N - \bar{m}}{s_N} \end{bmatrix}$$

In this homework, X is a 16384×630 matrix. The co-variance matrix C is defined as

$$C = XX^T$$

Step 2: Compute Projection Matrix P that Diagonalize C

We first want to find a matrix V that diagonalize C as $V^T C V = \Lambda$, where Λ is a diagonal matrix. We use the eigenvectors of C to construct V and use the corresponding eigenvalues to construct Λ on its diagonal.

To avoid doing eigenvalue decomposition on the huge matrix C (a 16384×16384 matrix), we instead do eigenvalue decomposition on $C' = X^T X$, which is much smaller (a 630×630 matrix). Thus we have $U^T C' U = \Lambda'$. C and C' have exactly the same non-zero eigenvalues and each eigenvector V_i of C is normalized vector $X \cdot U_i$

$$V_i = \frac{X \cdot U_i}{\|X \cdot U_i\|}$$

Finally, the projection matrix P only preserves N eigenvectors in V that corresponds the largest eigenvalue because these N eigenvectors capture most information in X with the least lost of accuracy. Thus $P = [V_1 \quad V_2 \quad \dots \quad V_N]$, where the number N is a tuning parameter of PCA method.

Step 3: Projection and Classification

After constructing projection matrix P, each training image x_i is projected to a smaller N-dimensional feature space using its normalized representation.

$$\widehat{x_i} = P^T \cdot \frac{x_i - \overline{m}}{s_i}$$

Each training image in feature space and training label $(\widehat{x_i}, y_i)$ and the mean face $\overline{m}$ are used for testing scenario.

For a given test image xt, we compute its standard deviation st and then normalize and project it by,

$$\widehat{xt} = P^T \cdot \frac{xt - \overline{m}}{st}$$

The index of the image using nearest neighbor approach is given by,

$$i^* = \underset{i}{\operatorname{argmin}} \|\widehat{xt} - \widehat{x_i}\|^2$$

Thus, the prediction label of the test face xt is y_{i^*}.

1.2 Face recognition with LDA

Step 1: Construct the between class scatter S_B and within class scatter S_w

Denote the training set as $(x_{ik}, y_{ik} = i), i = 1, \dots, M = 30, k = 1, \dots, C = 21$, where x_{ik} is the flattened vector representation of the k-th gray-scale face of the i-th person and y_i is the ground truth label of this face. Denote the global mean face $\overline{m}$ and mean face of each person $\overline{m_i}$ as,

$$\begin{cases} \overline{m} = \dfrac{1}{MC}\displaystyle\sum_{i=1}^{M}\sum_{k=1}^{C} x_{ik} \\ \overline{m_i} = \dfrac{1}{C}\displaystyle\sum_{k=1}^{C} x_{ik} \end{cases}$$

So, the between class scatter S_B and within class scatter S_w are defined as,

$$\begin{cases} S_B = \displaystyle\sum_{i=1}^{M} (\overline{m_i} - \overline{m})(\overline{m_i} - \overline{m})^T = M_1 M_1^T, M_1 = [\overline{m_1} - \overline{m} \quad \overline{m_2} - \overline{m} \quad \dots \quad \overline{m_M} - \overline{m}] \\ S_w = \displaystyle\sum_{i=1}^{M}\sum_{k=1}^{C} (x_{ik} - \overline{m_i})(x_{ik} - \overline{m_i})^T = M_2 M_2^T, M_2 = [\overline{x_{11}} - \overline{m_1} \quad \overline{m_{12}} - \overline{m_1} \quad \dots \quad \overline{m_{30,21}} - \overline{m_{30}}] \end{cases}$$

In this homework, both S_B and S_w are 16384×16384 matrix.

Step 2: Compute Projection Matrix A that Maximize Fisher Discriminant Function

The Fisher Discriminant Function characterize the importance of a feature vector w by

$$J(w) = \frac{w^T S_B w}{w^T S_w w}$$

In Yu and Yang's algorithm, we solve this by finding a projection matrix A that maximize,

$$A^* = \underset{A}{\operatorname{argmax}} \frac{|AS_BA^T|}{|AS_wA^T|}$$

Firstly, we want to find a matrix V that diagonalizes S_B as $V^TS_BV = \Lambda$, where Λ is a diagonal matrix. We use the eigenvectors of S_B to construct V and use the corresponding eigenvalues to construct Λ on its diagonal. To avoid doing eigenvalue decomposition on the huge matrix S_B (a 16384×16384 matrix), the same computation trick is used as discussed in PCA method.

We preserve all 30 eigenvectors that corresponding to the 30 non-zero eigenvalues in S_B to construct Y,

$$Y = [V_1 \quad V_2 \quad \dots \quad V_{30}]$$

The 30 non-zero eigenvalues construct the diagonal matric D_b, which is the 30×30 principal sub-matrix of Λ, and thus we have $Y^TS_BY = D_b > 0$.

Secondly, we want to find a matrix Z that diagonalizes and normalizes S_B as $Z^TS_BZ = \mathrm{I}$. This matrix is given by $Z = Y \cdot D_b^{-1/2}$. So, Z is part of A.

Thirdly, we want to further diagonalizes S_w. Considering the former steps, we now have Z^TS_wZ, which is a 30×30 matrix. We can also use eigenvalue decomposition to that diagonalize Z^TS_wZ as $U^TZ^TS_wZU = \Lambda_w$. Similar to PCA method, we preserve N eigenvectors that corresponding to the N smallest eigenvalues of Z^TS_wZ and get,

$$P = [U_{30-N+1} \quad U_{30-N+2} \quad \dots \quad U_{30}]$$

$P^TZ^TS_wZP = D_w$, where D_w is the last $N \times N$ sub-matrix of Λ_w

Finally, we want to find a matrix that diagonalizes and normalizes S_w as $A^TS_wA = \mathrm{I}$. This matrix is given by $A = Z \cdot P \cdot D_w^{-1/2}$.

Step 3: Projection and Classification

After constructing projection matrix A, each training image x_i is projected to a smaller N-dimensional feature space using its normalized representation.

$$\widehat{x_i} = A^T \cdot \frac{x_i - \overline{m}}{s_i}$$

Each training image in feature space and training label $(\widehat{x_i}, y_i)$ and the mean face $\overline{m}$ are used for testing scenario.

For a given test image xt, we compute its standard deviation st and then normalize and project it by,

$$\widehat{xt} = A^T \cdot \frac{xt - \overline{m}}{st}$$

The index of the image using nearest neighbor approach is given by,

$$i^* = \underset{i}{\operatorname{argmin}} \|\widehat{xt} - \widehat{x_i}\|^2$$

Thus, the prediction label of the test face xt is y_{i^*}.

1.3 Car detection

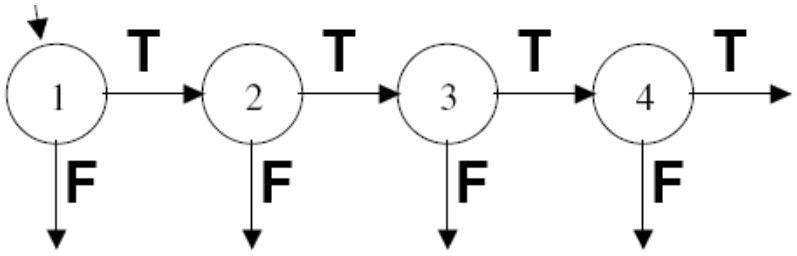
In this task, we want to detect a car from a 20x40 image. Generally, we construct the image classification system with cascaded strong classifiers. Each strong classifier is composed of several weak classifier and each weak classifier can be a simple Haar function. From top to down, we will introduce them one by one.

1.3.1 Image classification system with cascaded strong classifier

The image classification system is shown in the following figure. The input is a raw gray-scale image to the first component, a.k.a strong classifier. The first strong classifier will detect whether there is an object or not. If not, the system immediately outputs false. While if true, the image has to go through several more strong classifiers. If one of the strong classifiers outputs false, the system outputs false. While if all strong classifiers outputs true, the system outputs true. In this system, we call each strong classifier a stage. We denote the number of strong classifiers as S. Denote the true positive rate at stage i as TPR_i, and the false positive rate at stage i as FPR_i. Apparently, the true positive rate TPR and false positive rate FPR of the system is the product of each stage.

$$\begin{cases} TPR = \prod_{i=1}^{S} TPR_i \\ FPR = \prod_{i=1}^{S} FPR_i \end{cases}$$

Typically, when we train each individual strong classifier, we target at true positive rate TPR_i around 99% and false positive rate FPR_i around 30%. Note that to train this system, we use the whole training set for the first strong classifier and use the true detection samples from previous classifier for all following classifiers.



1.3.2 Strong classifiers with low-level features

The purpose of a strong classifier is to detect an object from an image with relatively high accuracy. As required by the image classification system, the optimal true positive rate is 99% and false positive rate is 30%. Denote the strong classifier as a function of input $H(x) \in \{0, 1\}$, where x is the input and the output is a 0-1 variable denoting true of false. The strong classifier is collection of lots of weak classifiers $\{H_1(x), H_2(x), \ldots, H_T(x)\}$, where the output of each weak classifier is also in $\{0,1\}$. To construct the output of the strong classifier, we calculate the weighted sum of weak classifiers and compare to a fixed threshold, a.k.a.

$$H(x) = \begin{cases} 1, if \sum_{t=1}^{T} \alpha_t H_t(x) > \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0, otherwise \end{cases}$$

where α_t is the vote of the t-th weak classifier $H_t(x)$.

1.3.3 Selecting from a list of weak classifier candidates to construct strong classifer

Suppose we have a significantly large list of weak classifier candidates that can classify an input x into 0-1 detection results. We use an iterative algorithm to determine the T weak classifiers in each strong classifier. The step are as follows,

Step 1. Initialize the temporal importance of each training sample – weight

Denote the training set as $\{(x_1, y_1), \dots, (x_N, y_N)\}$. Where the total number of positive samples are P and total number of negative samples are Q. Then the weights $\{w_1, \dots, w_N\}$ is initialized as follows,

$$w_i = \begin{cases} \frac{1}{2P}, if\ y_i = 1 \\ \frac{1}{2Q}, if\ y_i = 0 \end{cases}$$

This initialization guarantees the equal importance of each training sample normalized by the number of samples in each class. The weight of each training sample will change, and it represents the importance of this sample to classify it correctly at any time. If models classify this sample correctly, it will decrease, while it will increase otherwise.

Step 2. Select the best weak classifier that minimize the weighted error.

Denote the weak classifier candidates $\{f_1(x), \dots f_K(x)\}$, where each takes the input image as input and outputs 0-1 detections. The selected weak classifier $f_{i^*}(x)$ in this step is the one that minimize the weight error.

$$i^* = \underset{i}{\operatorname{argmin}} \sum_{j=1}^{N} w_j \cdot \left| f_i(x_j) - y_j \right|$$

The error ε_t, boosting parameter β_t, vote parameter α_t are presented as

$$\varepsilon_t = \min_i \sum_{j=1}^{N} w_j \cdot \left| f_i(x_j) - y_j \right|$$

$$\beta_t = \frac{\varepsilon_t}{1 - \varepsilon_t} < 1$$

$$\alpha_t = log \frac{1}{\beta_t}$$

Step 3. Update the importance parameter (weight) for future use

The weight of each image is updated depending on whether this image is correctly classified by the selected weak classifier $f_{i^*}(x)$ as follows,

$$w_{t+1} = \begin{cases} w_t, if\ f_{i^*}(x_t) \neq y_t\ (unchanged) \\ w_t\beta_t, if f_{i^*}(x_t) = y_t (decreased) \end{cases}$$

Then all weights are normalized by the sum. So the weights of wrongly classified samples increase finally. Finally, go to step 2 until we each the maximum number T of weak classifier in each strong classifier or early termination condition is satisfied.

Early termination condition: at any iteration t_0, if all t_0 weak classifiers have already constructed a strong classifier that meet the TPR and FPR requirement, the program terminates. So, it will terminate if the temporal strong classifier $H_{t_0}(x)$ meets such requirement,

$$H_{t_0}(x) = \begin{cases} 1, if\ \sum_{t=1}^{t_0} \alpha_t H_t(x) > \frac{1}{2}\sum_{t=1}^{t_0} \alpha_t \\ 0, otherwise \end{cases}$$

1.3.4 Constructing a list of weak classifier candidates using Haar feature extractor.

To construct a significant large list of weak classifier candidates, I use Haar feature extractor. There are two types of Haar operator, one for vertical, denoting $[-1, 1]^T$, with $2H \times W$ shape and one for horizontal, denoting $[-1,1]$, with $H \times 2W$ shape. We use a sliding window to place the Haar operator on every possible pixels on the image. Given a 20x40 image, the number of exhaustive set of vertical Haar operators is

$$\sum_{H=1}^{10}\sum_{W=1}^{40}\sum_{H_0=1}^{21-2H_0}\sum_{W_0}^{41-W_0} 1 = 82000$$

The number of exhaustive set of horizontal Haar operators is

$$\sum_{H=1}^{20}\sum_{W=1}^{20}\sum_{H_0=1}^{21-H_0}\sum_{W_0}^{41-2W_0} 1 = 84000$$

Finally, there are 166000 Haar features extract from the image.

We then have any arbitrary real-valued threshold to make a decision based on each real-valued feature $fe_i(x)$ as,

$$f_i(x) = \begin{cases} 1, if\ fe_i(x) < p \\ 0, otherwise \end{cases}$$

There is another polarity to do this as,

$$f_i(x) = \begin{cases} 1, if\ fe_i(x) \geq p \\ 0, otherwise \end{cases}$$

However, there are at maximum $N + 1$ unique classifiers, which p can take value from $\{fe_i(x_1), \dots, fe_i(x_N), \infty\}$. Thus, the final number of weak classifier candidate are

$$166000 \times (N + 1) \times 2 = 819708000$$

Efficient implementation: to reduce the time of selecting weak classifiers from such huge candidate list, we set $StepOfCls$ parameter as the interval we regard a weak classifier candidate as true candidate (otherwise

ignored and never selected). We set $StepOfTh$ parameter as the interval we choose threshold from the feature of the training samples (otherwise ignored and never used). Both polarity are considered anyway.

2. Evaluations

2.1 Face recognition

For PCA, I choose the N eigenvectors corresponding to the largest eigenvalues of XX^T. For LDA, I preserve all 30 eigenvectors in S_b, and choose N eigenvectors corresponding to the smallest eigenvalues of $Z^T S_w Z$.

In Figure 1, I showed the classification accuracy of PCA and LDA. We can observe that LDA is always better than PCA. LDA method achieves 100% accuracy when top 7 eigenvectors are preserved while PCA reaches 100% accuracy when 19 eigenvectors are preserved. This shows LDA method produces better projection space to classify faces than PCA method.

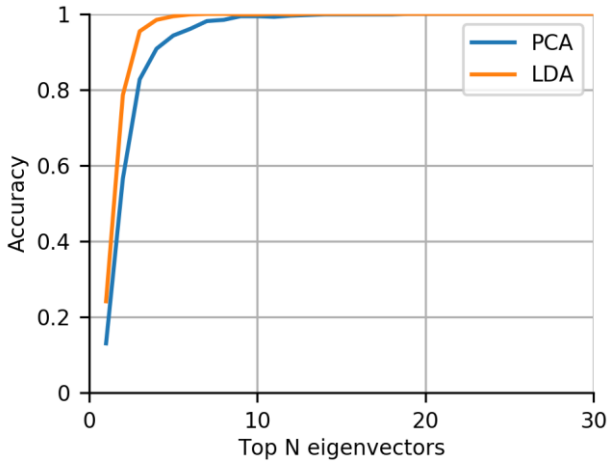


Figure 1: The classification accuracy when N eigenvectors are chosen.

2.2 Car detection

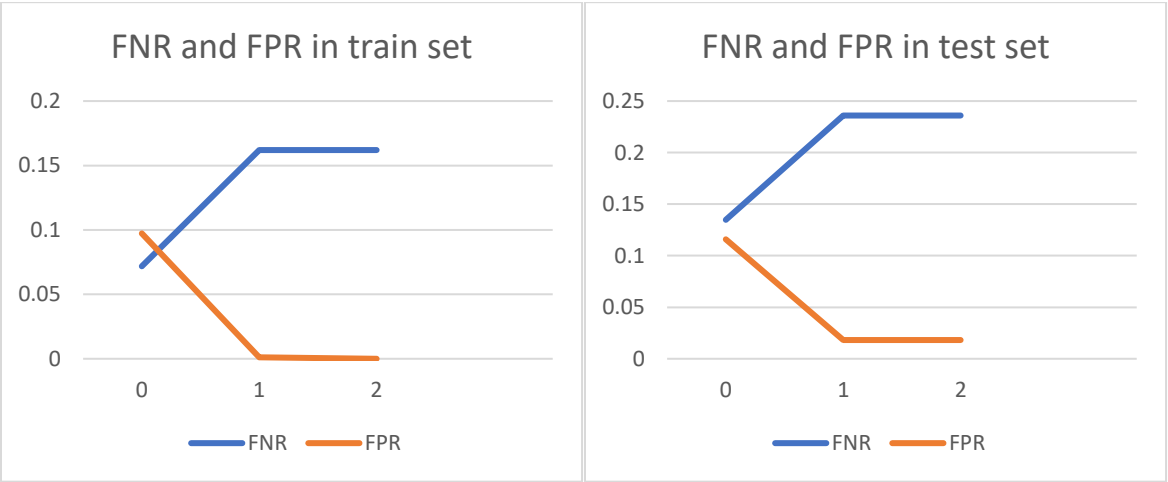
I finally end up with 3 stages. The parameters for each stage are as follows,

Stage	# Classifier T	$StepOfCls$	$StepOfTh$	Polarity	# Weak classifier candidates	Training time
0	10	2	10	Both	8,300	84 minutes
1	20	2	2	Both	41,500	219 minutes
2	20	2	2	Both	41,500	149 minutes

The accuracy results in training and test set are as follows,

Stage	Train			Test		
	Positive class detected T	Negative class detected T	(FNR, FPR) in stage (FNR, FPR) cumulative	Positive class detected T	Negative class detected T	(FNR, FPR) in stage (FNR, FPR) cumulative
0	710→659	1758→171	(7.18%, 9.73%) (7.18%, 9.73%)	178→154	440→51	(13.48%, 11.59%) (13.48%, 11.59%)
1	659→595	171→2	(9.71%, 1.17%) (16.20%, 0.11%)	154→136	51→8	(11.69%, 15.69%) (23.60%, 1.82%)
2	595→595	2→0	(0.00%, 0.00%) (16.20%, 0.00%)	136→136	8→8	(0.00%, 100.00%) (23.60%, 1.82%)

The final false negative rate on the test set is 23.60% (42/178) and false positive rate is 1.82% (8/440). These rates as a function of stage count S is shown as follows,



3. Source code

The face images are stored in “ECE661_2018_hw10_DB1/”

3.1 Helper Functions of PCA and LDA

```
# ECE 661 hw10 - Face Recognition
# Author: Ran Xu (xu943@purdue.edu)
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
def TrickyEigen(M, p=-1, FromTop=True): # (N, P) N>>P
    if(M.shape[0]>M.shape[1]):
        # Compute eigen of M^T * M instead of M * M^T
        A = np.dot(np.transpose(M), M)
        lamda, U = np.linalg.eig(A)
        V = np.dot(M, U)
        StdV = np.sqrt(np.sum(V**2, axis = 0, keepdims = True))
        NormV = V/StdV
    else:
        A = np.dot(M, np.transpose(M))
        lamda, NormV = np.linalg.eig(A)
    if p == -1:
        p = min(M.shape[0], M.shape[1])
    if FromTop:
        return NormV[:, 0:p], lamda[0:p]
    else:
        return NormV[:, -p-1:-1], lamda[-p-1:-1]
```

3.2 Main Function - PCA

```
TrainingFiles, TestFiles = ([], [])
for IdxPerson in range(1,31):
    for IdxPhoto in range(1,22):
        TrainingFiles.append("ECE661_2018_hw10_DB1/train/%02d_%02d.png"
                             %(IdxPerson, IdxPhoto))
        TestFiles.append("ECE661_2018_hw10_DB1/test/%02d_%02d.png"
                         %(IdxPerson, IdxPhoto))
# Face Recognition with PCA
# Load training images
ims = [cv.imread(x, cv.IMREAD_GRAYSCALE).reshape(-1,1) for x in TrainingFiles]
OriX = np.hstack(ims)
MeanX = np.mean(OriX, axis = 1, keepdims = True)      #(49152,1)
StdX = np.std(OriX-MeanX, axis = 0, keepdims = True) #(1,630)
NormX = (OriX-MeanX)/StdX
# Load test images
ims = [cv.imread(x, cv.IMREAD_GRAYSCALE).reshape(-1,1) for x in TestFiles]
TestOriX = np.hstack(ims)
TestNormX = (TestOriX-MeanX)/StdX

PCA_Accuracy = np.zeros((31,))
for p in range(1,31):
    # Compute top-p eigen vectors
    NormV, _ = TrickyEigen(NormX, p=p)
    # Do PCA on training images
    NormX_PCA = np.dot(np.transpose(NormV), NormX); # (p, 630)
    # Do PCA on test images
    TestNormX_PCA = np.dot(np.transpose(NormV), TestNormX); # (p, 630)
    # Matching
    Correct = np.zeros((630,))
    for IdxTestImg in range(0,630):
        Score = [np.sum((TestNormX_PCA[:,IdxTestImg]-NormX_PCA[:,IdxTr])**2)
                 for IdxTr in range(630)]
        NearestNeighbor = np.argmin(Score)
```

```
        Correct[IdxTestImg] = (int(IdxTestImg/21)==int(NearestNeighbor/21))
    PCA_Accuracy[p] = np.sum(Correct)/630
    print("(%d, %.2f%%)" %(p, PCA_Accuracy[p]*100), end = ", ")
```

3.3 Main Function - LDA

```
# Face Recognition with LDA
# Load training images
ims = [cv.imread(x, cv.IMREAD_GRAYSCALE).reshape(-1,1) for x in TrainingFiles]
OriX = np.hstack(ims)
# Load test images
ims = [cv.imread(x, cv.IMREAD_GRAYSCALE).reshape(-1,1) for x in TestFiles]
TestOriX = np.hstack(ims)

# First step: Compute S_b and diagonalize it with trick
M = np.zeros((OriX.shape[0], 30))    # Sb = M*M^T
MeanX = np.mean(OriX, axis = 1, keepdims = True)      # (N,1)
for x in range(30): # 30 classes
    MeanX_In = np.mean(OriX[:, x*21:x*21+21], axis = 1, keepdims = True) #(N,1)
    M[:, x:x+1] = MeanX_In - MeanX
V, lamda = TrickyEigen(M)  # lamda is actually
Z = np.dot(V, np.sqrt(np.diag(1/lamda)))  # (N, 30)

# Second step: Compute Z^T*Sw*Z and diagonalize it with trick
M = np.zeros(OriX.shape)    # Sw = M*M^T
for x in range(30): # 30 classes
    MeanX_In = np.mean(OriX[:, x*21:x*21+21], axis = 1, keepdims = True) #(N,1)
    for y in range(21):
        Index = x*21 + y
        M[:, Index:Index+1] = OriX[:, Index:Index+1] - MeanX_In
M2 = np.dot(np.transpose(Z), M)  # (Z^T * M) * (Z^T * M)^T

LDA_Accuracy = np.zeros((31,))
for p in range(1, 31):
    # Compute top-p eigen vectors
    U, lamda = TrickyEigen(M2, p=p, FromTop=False)  # lamda is actually
    A = np.dot(Z, U)
    A = np.dot(A, np.sqrt(np.diag(1/lamda)))
    # Do LDA on training images
    OriX_LDA = np.dot(np.transpose(A), OriX); # (p, 630)
    # Do LDA on test images
    TestOriX_LDA = np.dot(np.transpose(A), TestOriX); # (p, 630)
    # Matching
    Correct = np.zeros((630,))
    for IdxTestImg in range(0,630):
        Score = [np.sum((TestOriX_LDA[:,IdxTestImg]-OriX_LDA[:,IdxTr])**2)
                 for IdxTr in range(630)]
        NearestNeighbor = np.argmin(Score)
        Correct[IdxTestImg] = (int(IdxTestImg/21)==int(NearestNeighbor/21))
    LDA_Accuracy[p] = np.sum(Correct)/630
    print("(%d, %.2f%%)" %(p, LDA_Accuracy[p]*100), end = ", ")
```

3.4 Helper Functions of Car Detection

The car images are stored in “ECE661_2018_hw10_DB2/”

```
import cv2 as cv
import numpy as np
import time
import copy
import sys
```

```
def GetResult(Pred_Final, TestGT):
    TP = np.sum(np.logical_and(Pred_Final==1, TestGT==1))
    FP = np.sum(np.logical_and(Pred_Final==1, TestGT==0))
    TN = np.sum(np.logical_and(Pred_Final==0, TestGT==0))
    FN = np.sum(np.logical_and(Pred_Final==0, TestGT==1))
    if TP+FN == 0:
        TPR = 0
    else:
        TPR = TP/(TP+FN)
    if FP+TN == 0:
        FPR = 1
    else:
        FPR = FP/(FP+TN)
    return TP, FP, TN, FN, TPR, FPR
```

3.5 Main Training Function - Car Detection

```
import pickle
S, T, StepOfCls, StepOfTh = (10, 3, 200, 10)  # 20k speed-up in laptop
TPR_Min, FPR_Max = (0.9, 0.3)  # If TPR > TPR_Min AND FPR < FPR_Max, terminate
for Stage in range(S):
    # HaarResults  166k x 2468
    HaarResults = np.load("HaarResults.npy")
    if Stage==0:
        Mask = np.array([True]*2468)
    else:
        Mask = np.load("Mask_epoch_%d.npy" %(Stage-1))
    HaarResults = HaarResults[:, Mask]
    HaarResults = np.hstack([HaarResults,
                             np.ones((HaarResults.shape[0],1))*1e7])
    GT = np.array([1]*710 + [0]*1758)
    GT = GT[Mask]
    InitWt = np.array([1/np.sum(GT)/2]*710 + [1/np.sum(GT==0)/2]*1758)
    InitWt = InitWt[Mask]

    Wt = np.zeros((T+1, InitWt.shape[0]))
    Wt[0, :] = np.copy(InitWt)
    alpha, beta, SltWeakCls, SltThIdx = (np.zeros((T,)), np.zeros((T,)),
                                         np.zeros((T,)).astype("int"), np.zeros((T,)).
astype("int"))

    time1 = time.time()
    for t in range(T):  # T iteration of picking weak classifier
        print("Iteration #%d/%d in stage %d/%d of picking weak classifier"
              %(t, T, Stage, S))
        BestSpecInCls = [(1e7, -1)]*HaarResults.shape[0]
        for ClsIdx in range(HaarResults.shape[0]):
            if ClsIdx%StepOfCls != 0:
                BestSpecInCls[ClsIdx] = (1e7, -1)  # Error is large enough
                continue
            # Choose the best Th. (2468+1) and polarity (+,-), save in BestSpecInCls[C
lsIdx]
            MinWtErr, MinThIdx = (1., -1)
            # Positive (+) first: < means "negative"(0), >= means "positive" 1
            for ThIdx, Th in enumerate(list(HaarResults[ClsIdx, :])):
                if ThIdx%StepOfTh != 0:
                    continue
                Pred = (HaarResults[ClsIdx, 0:-1]>=Th)
                Wrong = np.not_equal(Pred, GT)
                WtErr = np.sum(Wt[t, :]*Wrong)
                if(WtErr < MinWtErr):
```

```
                    MinWtErr, MinThIdx = (WtErr, ThIdx)
            # Negative (-) next: >= means "negative"(0), < means "positive" 1
            for ThIdx, Th in enumerate(list(HaarResults[ClsIdx, :])):
                if ThIdx%StepOfTh != 0:
                    continue
                Pred = (HaarResults[ClsIdx, 0:-1]<Th)
                Wrong = np.not_equal(Pred, GT)
                WtErr = np.sum(Wt[t, :]*Wrong)
                if(WtErr < MinWtErr):
                    MinWtErr, MinThIdx = (WtErr, ThIdx+HaarResults.shape[1])
            if(ClsIdx%40000==0):
                if(MinThIdx<HaarResults.shape[1]): # Positive polarity
                    print("ClsIdx = %-3d, MinWtErr, MinThIdx, MinTh = (%.4f, %-4d, %3.
0f) (+)"
                          %(ClsIdx, MinWtErr, MinThIdx, HaarResults[ClsIdx, MinThId
x]))
                else:
                    print("ClsIdx = %-3d, MinWtErr, MinThIdx, MinTh = (%.4f, %-4d, %3.
0f) (-)"
                          %(ClsIdx, MinWtErr, MinThIdx, HaarResults[ClsIdx, MinThIdx-H
aarResults.shape[1]]))
            BestSpecInCls[ClsIdx] = (MinWtErr, MinThIdx)
        np.save("save/BestSpecInCls_it%d_stage%d.npy" %(t,Stage), BestSpecInCls)

        # Select the best weak weak classifier
        MinWtErrs = [x[0] for x in BestSpecInCls]
        MinErr = np.min(MinWtErrs)
        SltWeakCls[t] = int(np.argmin(MinWtErrs))
        SltThIdx[t] = BestSpecInCls[SltWeakCls[t]][1]
        Th = HaarResults[SltWeakCls[t], int(SltThIdx[t])%HaarResults.shape[1]]
        print("Weak classifier #%d is selected, MinWtErr, MinThIdx, MinTh = (%.4f, %-4
d, %3.0f)"
              %(SltWeakCls[t], BestSpecInCls[SltWeakCls[t]][0], SltThIdx[t], Th))

        # Update weight
        beta[t] = MinErr/(1-MinErr)
        alpha[t] = np.log(1/beta[t])
        if(SltThIdx[t] < HaarResults.shape[1]): # Positive polarity
            Pred = (HaarResults[SltWeakCls[t], 0:-1]>=Th)
            Wrong = np.not_equal(Pred, GT)
            # Wrong = 1 then +=0, Wrong = 0, then +=(beta-1)*Wt
            Wt[t+1, :] = Wt[t, :] + (beta[t] - 1) * Wt[t, :] * (1-Wrong)
        else: # Negative polarity
            Pred = (HaarResults[SltWeakCls[t], 0:-1]<Th)
            Wrong = np.not_equal(Pred, GT)
            # Wrong = 1 then +=0, Wrong = 0, then +=(beta-1)*Wt
            Wt[t+1, :] = Wt[t, :] + (beta[t] - 1) * Wt[t, :] * (1-Wrong)
        Wt[t+1, :] = Wt[t+1, :] / np.sum(Wt[t+1, :])

        # Saving the varibles
        SaveItem = [SltWeakCls[t], SltThIdx[t], beta[t], Pred, Wrong, Wt[t, :], Wt[t+
1, :]]
        with open("save/Debugging_it%d_stage%d.npy" %(t,Stage), 'wb') as f:
            pickle.dump(SaveItem, f)

        # Determine whether to terminate early
        TP, FP, TN, FN, TPR, FPR = GetResult(Pred, GT)
        print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f) (In step)"
              %(TP, FP, TN, FN, TPR, FPR))

        Feature_Final = np.zeros(GT.shape)
        Dec_Th = np.sum(alpha[:t+1])/2
        for t0 in range(t+1):
```

```
            # In this iteration SltWeakCls[t] is selected, with Th @ SltThIdx[t]
            Feature = HaarResults[SltWeakCls[t0], :-1]
            Th = HaarResults[SltWeakCls[t0], int(SltThIdx[t0])%HaarResults.shape[1]]
            if(SltThIdx[t0] < HaarResults.shape[1]): # Positive polarity
                Pred = (Feature>=Th)
                Feature_Final = Feature_Final + Pred * alpha[t0]
            else: # Negative polarity
                Pred = (Feature<Th)
                Feature_Final = Feature_Final + Pred * alpha[t0]
        Pred_Final = (Feature_Final > Dec_Th)
        TP, FP, TN, FN, TPR, FPR = GetResult(Pred_Final, GT)
        print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f) (Cumul.)"
              %(TP, FP, TN, FN, TPR, FPR))
        if TPR > TPR_Min and FPR < FPR_Max:
            break
    time2 = time.time()
    print("Training time per stage: %.2f s" %(time2 - time1))

    # Summarization printing
    fout = open("Summarization_Stage%d.txt" %(Stage), "w")
    print("The %d weak classifiers are as follows (time = %.2f s):" %(T, time2 - time
1))
    print("The %d weak classifiers are as follows (time = %.2f s):" %(T, time2 - time
1), file = fout)
    print("%-12s %-12s %-10s" %("SltWeakCls", "SltThIdx", "alpha"))
    print("%-12s %-12s %-10s" %("SltWeakCls", "SltThIdx", "alpha"), file = fout)
    for t in range(T):
        print("%-12d %-12d %-10.4f" %(SltWeakCls[t], SltThIdx[t], alpha[t]))
        print("%-12d %-12d %-10.4f" %(SltWeakCls[t], SltThIdx[t], alpha[t]), file = fo
ut)

    # Check Training Positives
    # Final strong classifier: Use alpha, SltWeakCls, SltThIdx to construct
    #    Eq: np.sum(alpha * h(x)) - 0.5*sum(alpha) >= 0
    # Compute final pred, (2468,) shape 0-1 vector
    # Compute false-postive rate->FP/(FP+TN) should be 30%
    # Compute true-postive rate->TP/(TP+FN) should be 99% or 1
    print("Testing on training set")
    print("Testing on training set", file = fout)
    Feature_Final = np.zeros(GT.shape)
    Dec_Th = np.sum(alpha)/2
    for t in range(T):
        # In this iteration SltWeakCls[t] is selected, with Th @ SltThIdx[t]
        Feature = HaarResults[SltWeakCls[t], :-1]
        Th = HaarResults[SltWeakCls[t], int(SltThIdx[t])%HaarResults.shape[1]]
        if(SltThIdx[t] < HaarResults.shape[1]): # Positive polarity
            Pred = (Feature>=Th)
            Feature_Final = Feature_Final + Pred * alpha[t]
        else: # Negative polarity
            Pred = (Feature<Th)
            Feature_Final = Feature_Final + Pred * alpha[t]
    Pred_Final = (Feature_Final > Dec_Th)
    TP, FP, TN, FN, TPR, FPR = GetResult(Pred_Final, GT)
    print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f)" %(TP, FP, TN, FN,
TPR, FPR))
    print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f)" %(TP, FP, TN, FN,
TPR, FPR), file = fout)
    if FP == 0:  # If all images in the next stage belongs to correct category, there
is no need to continue
        ContinueFlag = False
    else:
        ContinueFlag = True
```

```
    # Generating new masks
    Mask_ThisStage = list(Pred_Final==1)     # len = np.sum(Mask==1)
    Mask_TillNow = np.copy(Mask)             # len = 2468
    Mask_Postion = [idx for idx, x in enumerate(Mask) if x==True] # len = np.sum(Mask==1)
    for idx, current_mask in zip(Mask_Postion, Mask_ThisStage):
        Mask_TillNow[idx] = current_mask
    Mask_TillNow = np.array(Mask_TillNow==1)
    print("np.sum(Mask_TillNow) = ", np.sum(Mask_TillNow))
    np.save("Mask_epoch_%d.npy" %(Stage), Mask_TillNow)
    print()
    fout.close()

    if ContinueFlag==False:
        break
```

3.6 Main Test Function - Car Detection

```
print("Testing on training set")
Final_TP, Final_FP, Final_TN, Final_FN = (0,0,0,0)
Training_Mask = np.array([True]*2468) # Initial mask are all True
RealMaskList = []
for Stage in range(3):
    with open("Summarization_Stage%d.txt" %Stage) as f:
        lines = f.readlines()
    T = int(lines[0].split()[1])
    SltWeakCls = [int(x.strip().split()[0]) for x in lines[2:T+2]]
    SltThIdx = [int(x.strip().split()[1]) for x in lines[2:T+2]]
    alpha =  [float(x.strip().split()[2]) for x in lines[2:T+2]]

    # Re-shape the threshold matrix and feature matrix
    HaarResults = np.load("HaarResults.npy")  # HaarResults  166k x 2468
    HaarResults = np.array([HaarResults[x,Training_Mask] for x in SltWeakCls])
    HaarResults = np.hstack([HaarResults,
                             np.ones((HaarResults.shape[0],1))*1e7])

    TrainHaarResults = np.load("HaarResults.npy")
    TrainHaarResults = np.array([TrainHaarResults[x,:] for x in SltWeakCls])
    TrainGT = np.array([1]*710 + [0]*1758)
    for RealMask0 in RealMaskList:
        TrainGT = TrainGT[RealMask0]
        TrainHaarResults = TrainHaarResults[:, RealMask0]

    # Final strong classifier: Use alpha, SltWeakCls, SltThIdx to construct
    #    Eq: np.sum(alpha * h(x)) - 0.5*sum(alpha) >= 0
    # Compute final pred, (2468,) shape 0-1 vector
    # Compute false-postive rate->FP/(FP+TN) should be 30%
    # Compute true-postive rate->TP/(TP+FN) should be 99% or 1
    Feature_Final = np.zeros(TrainGT.shape)
    Dec_Th = np.sum(alpha)/2
    for t in range(T):
        # In this iteration SltWeakCls[t] is selected, with Th @ SltThIdx[t]
        Feature = TrainHaarResults[t, :]
        Th = HaarResults[t, int(SltThIdx[t])%HaarResults.shape[1]]
        if(SltThIdx[t] < HaarResults.shape[1]): # Positive polarity
            Pred = (Feature>=Th)
            Feature_Final = Feature_Final + Pred * alpha[t]
        else: # Negative polarity
            Pred = (Feature<Th)
            Feature_Final = Feature_Final + Pred * alpha[t]
    Pred_Final = (Feature_Final > Dec_Th)
    TP, FP, TN, FN, TPR, FPR = GetResult(Pred_Final, TrainGT)
```

```
    print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f) (In step)" %(TP, F
P, TN, FN, TPR, FPR))

    # Negative results are finalized
    Final_TN = Final_TN + TN
    Final_FN = Final_FN + FN
    print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f) (Cumul.)"
          %(TP, FP, Final_TN, Final_FN, TP/(TP+Final_FN), FP/(FP+Final_TN)))

    # Positive results go to next stage
    Real_Mask = (Pred_Final==1)
    RealMaskList.append(Real_Mask)
    print("Mask of real length %d in this step" %Real_Mask.shape[0])
    Training_Mask = np.load("Mask_epoch_%d.npy" %Stage)
    print("Mask of length %d loaded" %(Training_Mask.shape[0]))

Final_TP = TP
Final_FP = FP
print("Final: TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f)"
      %(Final_TP, Final_FP, Final_TN, Final_FN,
        Final_TP/(Final_TP+Final_FN), Final_FP/(Final_FP+Final_TN)))
```

```
print("Testing on test set")
Final_TP, Final_FP, Final_TN, Final_FN = (0,0,0,0)
Training_Mask = np.array([True]*2468) # Initial mask are all True
RealMaskList = []
for Stage in range(3):
    with open("Summarization_Stage%d.txt" %Stage) as f:
        lines = f.readlines()
    T = int(lines[0].split()[1])
    SltWeakCls = [int(x.strip().split()[0]) for x in lines[2:T+2]]
    SltThIdx = [int(x.strip().split()[1]) for x in lines[2:T+2]]
    alpha =  [float(x.strip().split()[2]) for x in lines[2:T+2]]

    # Re-shape the threshold matrix and feature matrix
    HaarResults = np.load("HaarResults.npy")  # HaarResults  166k x 2468
    HaarResults = np.array([HaarResults[x,Training_Mask] for x in SltWeakCls])
    HaarResults = np.hstack([HaarResults,
                             np.ones((HaarResults.shape[0],1))*1e7])

    TestHaarResults = np.load("TestHaarResults.npy")
    TestHaarResults = np.array([TestHaarResults[x,:] for x in SltWeakCls])
    TestGT = np.array([1]*178 + [0]*440)
    for RealMask0 in RealMaskList:
        TestGT = TestGT[RealMask0]
        TestHaarResults = TestHaarResults[:, RealMask0]

    # Final strong classifier: Use alpha, SltWeakCls, SltThIdx to construct
    #    Eq: np.sum(alpha * h(x)) - 0.5*sum(alpha) >= 0
    # Compute final pred, (2468,) shape 0-1 vector
    # Compute false-postive rate->FP/(FP+TN) should be 30%
    # Compute true-postive rate->TP/(TP+FN) should be 99% or 1
    Feature_Final = np.zeros(TestGT.shape)
    Dec_Th = np.sum(alpha)/2
    for t in range(T):
        # In this iteration SltWeakCls[t] is selected, with Th @ SltThIdx[t]
        Feature = TestHaarResults[t, :]
        Th = HaarResults[t, int(SltThIdx[t])%HaarResults.shape[1]]
        if(SltThIdx[t] < HaarResults.shape[1]): # Positive polarity
            Pred = (Feature>=Th)
            Feature_Final = Feature_Final + Pred * alpha[t]
        else: # Negative polarity
```

```
            Pred = (Feature<Th)
            Feature_Final = Feature_Final + Pred * alpha[t]
    Pred_Final = (Feature_Final > Dec_Th)
    TP, FP, TN, FN, TPR, FPR = GetResult(Pred_Final, TestGT)
    print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f) (In step)" %(TP, F
P, TN, FN, TPR, FPR))

    # Negative results are finalized
    Final_TN = Final_TN + TN
    Final_FN = Final_FN + FN
    print("TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f) (Cumul.)"
          %(TP, FP, Final_TN, Final_FN, TP/(TP+Final_FN), FP/(FP+Final_TN)))

    # Positive results go to next stage
    Real_Mask = (Pred_Final==1)
    RealMaskList.append(Real_Mask)
    print("Mask of real length %d in this step" %Real_Mask.shape[0])
    Training_Mask = np.load("Mask_epoch_%d.npy" %Stage)
    print("Mask of length %d loaded" %(Training_Mask.shape[0]))

Final_TP = TP
Final_FP = FP
print("Final: TP, FP, TN, FN, TPR, FPR = (%d, %d, %d, %d, %.4f, %.4f)"
      %(Final_TP, Final_FP, Final_TN, Final_FN,
        Final_TP/(Final_TP+Final_FN), Final_FP/(Final_FP+Final_TN)))
```