Homework 11

Fangda Li
li1208@purdue.edu
ECE 661 - Computer Vision

January 5, 2017

1 Methodology

This section briefly describes the principle steps of PCA, LDA and cascaded AdaBoost for classification problems.

1.1 PCA (Principle Component Analysis)

PCA aims to find the optimal subspace (of the original feature space) onto which the projected samples have the highest variance. The bases of the subspace are the first K eigenvectors of the covariance matrix $\mathbf{C}$, which is defined by

$$\mathbf{C} = \frac{1}{N}\sum_{i=0}^{N-1}(\vec{x_i} - \vec{m})(\vec{x_i} - \vec{m})^T. \tag{1}$$

where $\vec{x_i}$ The basis of the subspace is calculated as follows:

1. Suppose $\mathbf{X} \in R_{M,N,(M>N)}$, represents the N training samples each with M features. Moreover, $\mathbf{X}$ is normalized such that it has zero mean and unit variance.

2. The covariance matrix can be rewritten as

$$\mathbf{C} = \mathbf{X}\mathbf{X}^T. \tag{2}$$

3. Compute the eigenvectors $\vec{u}$ for $\mathbf{X}^T\mathbf{X}$. Then the eigenvectors $\vec{w}$ for $\mathbf{X}\mathbf{X}^T$ are obtained by

$$\vec{w} = \mathbf{X}\vec{u}. \tag{3}$$

4. Since the rank of $\mathbf{X}\mathbf{X}^T$ is N, at most N eigenvectors $\vec{w}$ can be obtained. The final basis of the subspace is choose to be the K eigenvectors corresponding to the K largest eigenvalues, where $K \leq N$.

After the basis has been obtained, all training samples are projected to the optimal subspace. Next, nearest neighbor approach is used in the optimal subspace to determine the label of a test sample.

1.2 LDA (Linear Discriminant Analysis)

LDA aims to find the most discriminating subspace in terms of the ratio of between-class scatter and within-class scatter. The steps of LDA for a multi-class discriminant problem are described as following:

1. Define the within-class scatter:

$$\mathbf{S}_W = \frac{1}{C}\sum_{i=1}^{C}\frac{1}{C_i}\sum_{k=1}^{C_i}(\vec{x_k}^i - \vec{m_i})(\vec{x_k}^i - \vec{m_i})^T, \tag{4}$$

   where x_k^i is the kth sample of the ith class.

2. Define the between-class scatter:

$$\mathbf{S}_B = \frac{1}{C}\sum_{i=1}^{C}(\vec{m_i} - \vec{m})(\vec{m_i} - \vec{m})^T. \tag{5}$$

3. The most discriminating vector will therefore maximize the Fisher Discriminant Function:

$$J(\vec{w}) = \frac{\vec{w}^T\mathbf{S}_B\vec{w}}{\vec{w}^T\mathbf{S}_W\vec{w}}. \tag{6}$$

4. Finally, the calculation of the LDA eigenvectors is carried out by calculating the $C-1$ mean difference vectors $\vec{m_i} - \vec{m}$.

Similar to PCA, the final classification task is done with the nearest neighbor approach on the projections onto the optimal subspace.

1.3 Cascaded AdaBoost

The overall idea of Viola and Jones is to cascade independently trained AdaBoost classifiers to suppress the false positive rate while preserving the recall. Several key insights of the algorithm are listed below:

1. Each cascade of the Viola and Jones classifier is an AdaBoost classifier. The main intuition of AdaBoost is that, after selecting the best weak classifier within each iteration, the weights of the misclassified samples by the new weak classifier are boosted. As a result, in the next iteration, the weak classifiers are evaluated with previously misclassified samples having larger weights. The concept is similar to hard negative mining in a sense. Moreover, AdaBoost assigns confidence values to each weak classifier based on its own error. Prediction of the final strong classifier is a weighted vote from all the previous best weak classifiers.

2. For each AdaBoost classifier in the cascade, it is bound to achieve a certain detection rate d while keeping its false positive rate f below a certain value. Then, the overall cascade classifier will have a false positive rate F,

$$F = \prod_{i=1}^{K} f_i, \tag{7}$$

and a detection rate D,

$$D = \prod_{i=1}^{K} d_i. \tag{8}$$

As a result of the products, the rate of decrease in F is much slower than D, given D is close to 1 and F is close to 0.

3. The full training procedure for one strong AdaBoost classifier is shown as followed:

   (a) Given m positive samples ($y = 1$) and l negative samples ($y = 0$), initialize the weights, $w_{1,i} = \frac{1}{2m}, \frac{1}{2l}$, for every positive and negative sample, respectively.

   (b) For $t = 1, .., T$:

      i. Normalize weights of all samples, $w_{t,i} = \frac{w_{t,i}}{\sum_j w_{t,i}}$.

      ii. Selected best weak classifier with respect to the weighted error

$$\epsilon_t = min_{f,p,\theta} \sum_i w_i |h(x_i, f, p, \theta) - y_i|. \tag{9}$$

      iii. Construct new weak classifier $h_t(x) = h(x, f, p, \theta)$ with f, p, θ minimizing ϵ_t.

      iv. Update weights for the samples using $w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$, where $e_i = 1$ when x_i is classified incorrectly and 0 otherwise, $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$ is the confidence of the current weak classifier.

   (c) The final strong AdaBoost classifier is:

$$C(x) = \begin{cases} 1 & \sum_t^T \alpha_t h_t(x) \geq 0.5 \sum_t^T \alpha_t \\ 0 & otherwise \end{cases} \tag{10}$$

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

2 Results and Discussion

2.1 PCA and LDA

PCA and LDA are evaluated simultaneously on the face dataset. The performance can be found in Figure 1, in which LDA slightly outperforms PCA given the same dimension of subspace. Nevertheless, both PCA and LDA are able to significantly reduce the dimensionality of the feature space while staying discriminating.

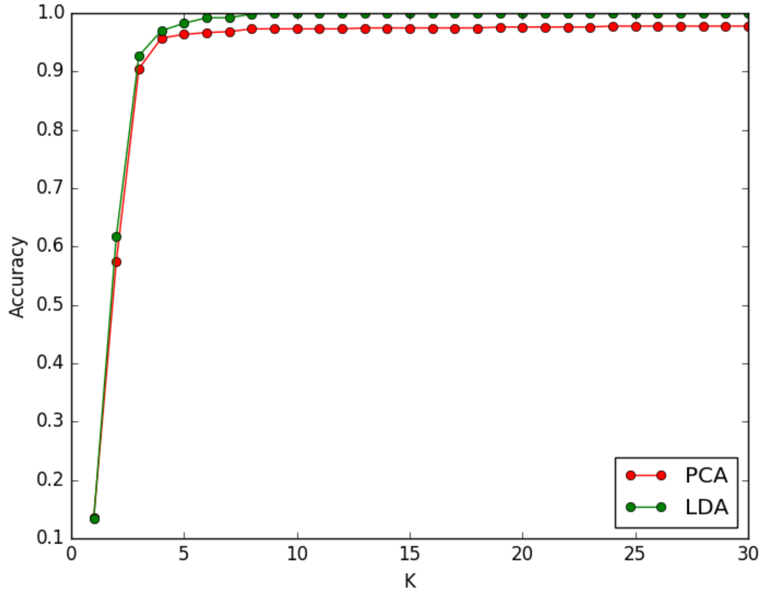

Figure 1: Classification accuracy v.s. Dimension of subspace

2.2 Cascaded AdaBoost

This subsection show the results of the Viola and Jones framework on the car dataset. The dataset consists of 710 p

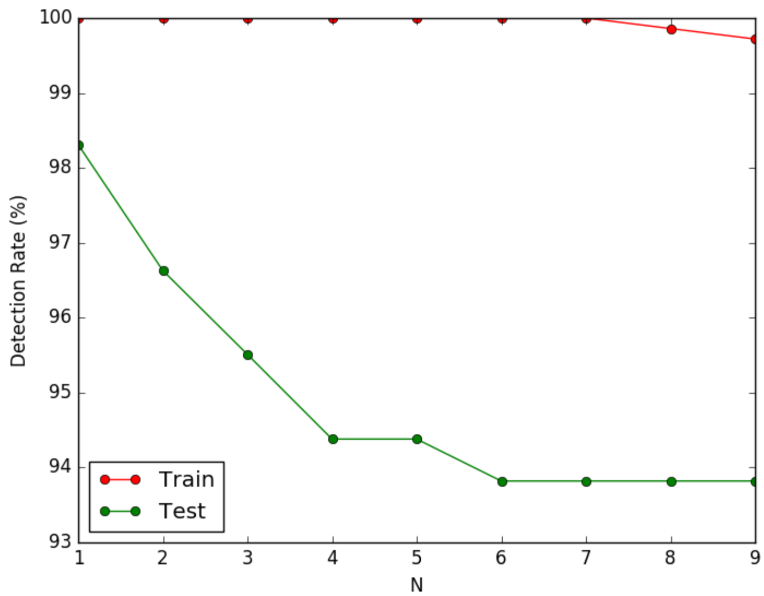

Figure 2: Detection Rate v.s. Number of Stages

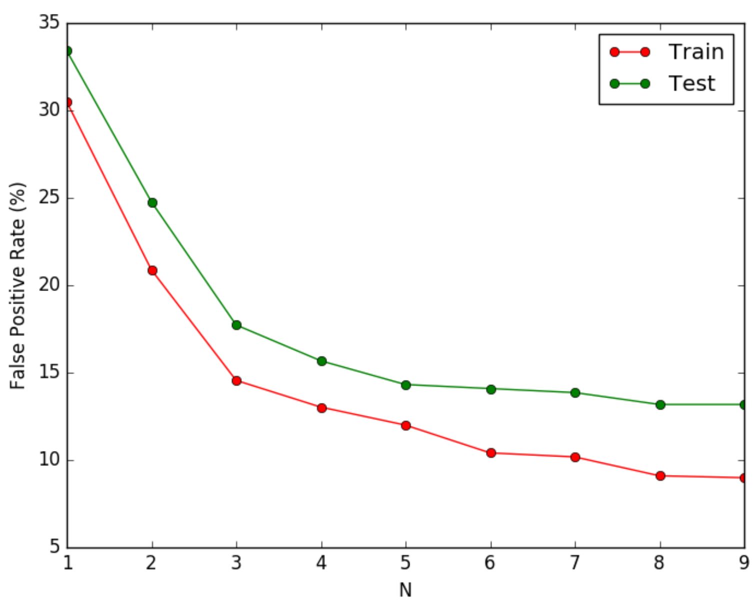

Figure 3: False Positive Rate v.s. Number of Stages

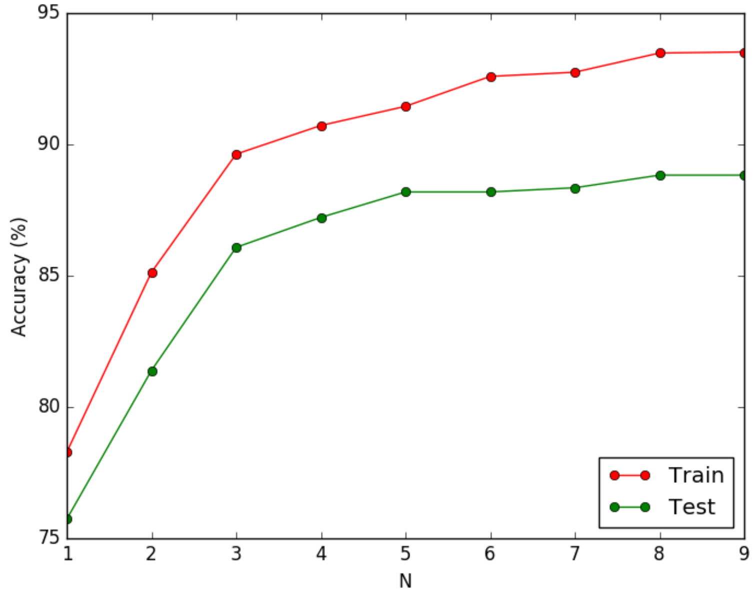

Figure 4: Accuracy v.s. Number of Stages

3 Source Code

3.1 pca.py

```
#!/usr/bin/python
from pylab import *
import cv2
from sklearn.neighbors import NearestNeighbors

class PCAClassifier(object):
        def __init__(self):
                super(PCAClassifier, self).__init__()
                self.K = 0
                self.m = None
                self.WK = None
                self.NN = None

        def train(self, train_data, train_label, K):
                '''
                        Construct K-D tree based on the projections onto the subspace
                '''
                print "======== PCA Training ========"
                self.train_data = train_data
                self.train_label = train_label
                self.K = K
```

```
        # Follow the notation of Avi's tutorial
        X = self.train_data
        m = mean(X, axis=1)
        _, _, Ut = svd(dot(X.T, X))
        W = dot(X, Ut.T)
        # Preserve the first K eigenvectors
        WK = W[:,:self.K]
        # Project all training samples to the K-D subspace
        Y = dot( WK.T, X - m.reshape(-1,1) )
        # Construct K-D tree
        self.NN = NearestNeighbors(n_neighbors=1, metric='euclidean').fit(Y.T)
        self.m = m
        self.WK = WK

    def test(self, test_data, test_label):
        '''
                Project onto the K-D optimal subspace and find the nearest nei
        '''
        print "======== PCA Testing ========"
        num_test = test_label.size
        # Projection first
        X = test_data
        Y = dot( self.WK.T, X - self.m.reshape(-1,1) )
        _, indices = self.NN.kneighbors(Y.T)
        pred = self.train_label[indices].flatten()
        accuracy = sum((pred - test_label) == 0) / float(num_test)
        print "K = {0}; Accuracy = {1:.2f}%".format(self.K, accuracy*100)
        return accuracy
```

3.2 lda.py

```
#!/usr/bin/python
from pylab import *
import cv2
from sklearn.neighbors import NearestNeighbors

class LDAClassifier(object):
    def __init__(self):
        super(LDAClassifier, self).__init__()
        self.K = 0
        self.W = None
        self.NN = None

    def train(self, train_data, train_label, K):
        '''
```

```
                Construct K-D tree based on the projections onto the subspace
        '''
        print "======== LDA Training ========"
        self.train_data = train_data
        self.train_label = train_label
        self.K = K
        self.num_classes = unique(train_label).size
        # Follow Avi's notation
        X = self.train_data
        L = self.train_label
        C = self.num_classes
        if self.K > C-1: self.K = C-1
        # Get the class means and global mean
        m = mean(X, axis=1)
        M = zeros((X.shape[0],C))
        for i in range(1,C+1):
                M[:,i-1] = mean(X[:, L==i], axis=1)
        # Eigenvectors of the between class scatter are the same as mean diffe
        W = M-m.reshape(-1,1)
        W = W[:,:self.K]
        # Project all training samples to the K-D subspace
        Y = dot( W.T, X - m.reshape(-1,1) )
        # Construct K-D tree
        self.NN = NearestNeighbors(n_neighbors=1, metric='euclidean').fit(Y.T)
        self.m = m
        self.W = W

def test(self, test_data, test_label):
        '''
                Project onto the K-D optimal subspace and find the nearest nei
        '''
        print "======== LDA Testing ========"
        num_test = test_label.size
        # Projection first
        X = test_data
        Y = dot( self.W.T, X - self.m.reshape(-1,1) )
        _, indices = self.NN.kneighbors(Y.T)
        pred = self.train_label[indices].flatten()
        accuracy = sum((pred - test_label) == 0) / float(num_test)
        print "Accuracy = {0:.2f}%".format(accuracy*100)
        return accuracy
```

3.3 adaboost.py

```
#!/usr/bin/python
from pylab import *
import cv2

WIDTH = 40
HEIGHT = 20
NUMFEATURES = 47232 # Obtained by _get_feature_matrix()

def get_integral_image(image):
        '''
                Compute and return the integral representation of an gray-scale image.
        '''
        return cumsum(cumsum(image,axis=0),axis=1)

def _get_feature_matrix():
        '''
                Return the feature matrix.
        '''
        features = zeros((NUMFEATURES, WIDTH*HEIGHT), dtype=int8)
        print "Constructing feature matrix...", features.shape
        count = 0
        # Offset from image boundary
        offset = 2
        # Do horizontal ones first, 1x2 base size
        bW, bH = 2, 1
        for i in range(1, HEIGHT, bH):                                                  # Extend
                for j in range(1, WIDTH, bW):                                           # Extend 
                        for y in range(0 + offset, HEIGHT - bH*i + 1 - offset):         #
                                for x in range(0 + offset, WIDTH - bW*j + 1 - offset):
                                        features[count, y*HEIGHT+x] = 1.0
                                        features[count, y*HEIGHT+x+bW*j/2] = -2.0
                                        features[count, y*HEIGHT+x+bW*j] = 1.0
                                        features[count, (y+bH*i)*HEIGHT+x] = -1.0
                                        features[count, (y+bH*i)*HEIGHT+x+bW*j/2] = 2.0
                                        features[count, (y+bH*i)*HEIGHT+x+bW*j] = -1.0
                                        count = count + 1
        # Vertical features, 2x1 base size
        bW, bH = 1, 2
        for i in range(1, HEIGHT, bH):                                                  # Extend
                for j in range(1, WIDTH, bW):                                           # Extend 
                        for y in range(0 + offset, HEIGHT - bH*i + 1 - offset):         #
                                for x in range(0 + offset, WIDTH - bW*j + 1 - offset):
                                        features[count, y*HEIGHT+x] = -1.0
```

```
                                    features[count, y*HEIGHT+x+bW*j] = 1.0
                                    features[count, (y+bH*i/2)*HEIGHT+x] = 2.0
                                    features[count, (y+bH*i/2)*HEIGHT+x+bW*j] = -2.0
                                    features[count, (y+bH*i)*HEIGHT+x/2] = -1.0
                                    features[count, (y+bH*i)*HEIGHT+x+bW*j] = 1.0
                                    count = count + 1
        print "Total number of features...", count
        return features

def _extract_features(data, feature_matrix):
        '''
                Extract all features from integral images into a nFeatures x nSamples
                Also return the sorted indices along the samples direction.
        '''
        # We are only concerned with two-rectangle edge features
        # Get the sorted list of sample indices.
        feature_vectors = dot( feature_matrix, data )
        print "Feature vectors size...", feature_vectors.shape
        print "Sorting samples based on feature values..."
        sorted_indices = argsort(feature_vectors, axis=1)
        print "Sorted indices size...", sorted_indices.shape
        return sorted_indices, feature_vectors

class CascadedAdaBoostClassifier(object):
        def __init__(self):
                super(CascadedAdaBoostClassifier, self).__init__()
                self.cascaded_adaboost = []
                self.train_data = None
                self.train_label = None
                self.train_feat_vecs = None
                self.train_num_pos = 0
                self.train_num_neg = 0
                self.test_data = None
                self.test_label = None
                self.feature_matrix = _get_feature_matrix()

        def set_testing_data(self, test_data, test_label):
                self.test_data = test_data
                self.test_label = test_label

        def train(self, train_data, train_label, num_stages, num_feats):
        # def train(self, train_data, train_label, f, d, Ftarg, maxIter):
                '''
                        Train cascaded AdaBoost classifiers given user-defined:
                        max acceptable fpr per layer f, min acceptable detection rate
```

```
        and overall fpr F_target.
    '''
    self.train_feat_vecs = dot( self.feature_matrix, train_data )
    self.train_num_pos = int(sum(train_label))
    self.train_num_neg = train_label.size - self.train_num_pos
    self.train_num = train_label.size
    self.train_data, self.train_label = train_data, train_label
    # Postive and negative training samples for current stage
    all_pos_feat_vecs = self.train_feat_vecs[:,self.train_label==1]
    all_neg_feat_vecs = self.train_feat_vecs[:,self.train_label==0]
    pos_feat_vecs = all_pos_feat_vecs
    neg_feat_vecs = all_neg_feat_vecs
    Flog_train = []
    Dlog_train = []
    Alog_train = []
    Flog_test = []
    Dlog_test = []
    Alog_test = []
    # Add stages
    for i in range(num_stages):
        print "Training %dth AdaBoost classifier in the cascade..." % (i
        current_adaboost = self._add_adaboost_classifier()
        current_adaboost.set_training_feature_vectors(pos_feat_vecs, neg
        # Add features
        for j in range(num_feats):
            print "Adding feature %d..." % (j+1)
            current_adaboost.add_weak_classifier()
        # Update negative samples to use
        fp_indices,F,D,A = self._classify_training_data()
        # Record training info
        Flog_train.append(F)
        Dlog_train.append(D)
        Alog_train.append(A)
        neg_feat_vecs = all_neg_feat_vecs[:,fp_indices-self.train_num_po
        # Record testing info
        F,D,A = self.test()
        Flog_test.append(F)
        Dlog_test.append(D)
        Alog_test.append(A)
        print "Training:"
        print "FP:\n", Flog_train
        print "RC:\n", Dlog_train
        print "AC:\n", Alog_train
        print "Testing:"
        print "FP:\n", Flog_test
```

```
                print "RC:\n", Dlog_test
                print "AC:\n", Alog_test

    def _add_adaboost_classifier(self):
        '''
                Allocate and return a new AdaBoost classifier.
        '''
        c = AdaBoostClassifier()
        c.set_feature_matrix(self.feature_matrix)
        self.cascaded_adaboost.append(c)
        return c

    def _classify_training_data(self):
        '''
                Evaluate the cascaded classifier on the training data,
                and return the indices of false postive samples as well as the
        '''
        print "Classifying training images..."
        feat_vecs = self.train_feat_vecs
        pos_indices = arange(self.train_num)
        for classifier in self.cascaded_adaboost:
                preds = classifier.classify_feature_vectors(feat_vecs)
                # Only pass on the samples with postive predictions
                feat_vecs = feat_vecs[:,preds==1]
                pos_indices = pos_indices[preds==1]
        # Final prediction
        fp_indices = pos_indices[ self.train_label[pos_indices] == 0 ]
        num_tp = sum(self.train_label[pos_indices])
        D = num_tp*1.0 / self.train_num_pos
        # Calculate false positive rate by counting the zeros
        F = (pos_indices.size - num_tp)*1.0 / self.train_num_neg
        w = self.train_num_pos*1.0 / (self.train_num_pos + self.train_num_neg)
        A = D * w + (1-F)*(1-w)
        print "F = %.4f, D = %.4f, A = %.4f" % (F,D,A)
        return fp_indices, F, D, A

    def test(self):
        '''
                Classify test images.
        '''
        print "Classifying testing images..."
        feat_vecs = dot( self.feature_matrix, self.test_data )
        test_num_pos = int(sum(self.test_label))
        test_num_neg = self.test_label.size - test_num_pos
        test_num = self.test_label.size
```

```
            pos_indices = arange(test_num)
            for classifier in self.cascaded_adaboost:
                    preds = classifier.classify_feature_vectors(feat_vecs)
                    # Only classify the samples with postive predictions
                    feat_vecs = feat_vecs[:,preds==1]
                    pos_indices = pos_indices[preds==1]
            # Final prediction
            num_tp = sum(self.test_label[pos_indices])
            D = num_tp*1.0 / test_num_pos
            # Calculate false positive rate by counting the zeros
            F = (pos_indices.size - num_tp)*1.0 / test_num_neg
            w = test_num_pos*1.0 / (test_num_pos + test_num_neg)
            A = D * w + (1-F)*(1-w)
            print "F = %.4f, D = %.4f, A = %.4f" % (F,D,A)
            return F,D,A

class AdaBoostClassifier(object):
        def __init__(self):
            super(AdaBoostClassifier, self).__init__()
            self.train_label = None
            self.train_sorted_indices = None
            self.train_feat_vecs = None
            self.train_num_pos = 0
            self.train_num_neg = 0
            self.threshold = 1.0
            self.sample_weights = None
            self.weak_classifier_indices = array([], dtype=int)
            self.weak_classifier_polarities = array([])
            self.weak_classifier_threshs = array([])
            self.weak_classifier_weights = array([])
            self.weak_classifier_results = array([])
            self.weak_classifier_weighted_results = None

        def set_feature_matrix(self, feature_matrix):
            self.feature_matrix = feature_matrix

        def set_training_feature_vectors(self, pos_feat_vecs, neg_feat_vecs):
            '''
                    Given current training feature vectors, sort them.
            '''
            self.train_num_pos = pos_feat_vecs.shape[1]
            self.train_num_neg = neg_feat_vecs.shape[1]
            self.train_label = hstack( (ones(self.train_num_pos), zeros(self.train_n
            self.train_feat_vecs = hstack( (pos_feat_vecs, neg_feat_vecs) )
            self.train_sorted_indices = argsort(self.train_feat_vecs, axis=1)
```

```
        print "Number of positive / negative samples in training...", self.train

    def add_weak_classifier(self):
        '''
                Add the current best weak classifier on the weighted training
        '''
        # Initialize all the weights if this is the first weak classifier
        if self.weak_classifier_indices.size == 0:
                self.sample_weights = zeros(self.train_label.size, dtype=float)
                self.sample_weights.fill( 1.0 / (2 * self.train_num_neg) )
                self.sample_weights[self.train_label==1] = 1.0 / (2 * self.train
        # Normalize the weights otherwise
        else:
                self.sample_weights = self.sample_weights / sum(self.sample_weig
        # Now pick the weak classifier with the min error with respect to the 
        best_feat_index, best_feat_polarity, best_feat_thresh, best_feat_error,
        # Update our list of weak classifiers
        self.weak_classifier_indices = append(self.weak_classifier_indices, best
        self.weak_classifier_polarities = append(self.weak_classifier_polarities
        self.weak_classifier_threshs = append(self.weak_classifier_threshs, best
        # Get confidence value of the best new classifier
        # Following the notation in the paper
        beta = best_feat_error / (1 - best_feat_error)
        alpha = log(1 / abs(beta))
        self.weak_classifier_weights = append(self.weak_classifier_weights, alph
        e = abs(best_feat_results - self.train_label)
        self.sample_weights = self.sample_weights * beta**(1-e)
        # Adjust the threshold
        if self.weak_classifier_results.size == 0:
                self.weak_classifier_results = best_feat_results.reshape(-1,1)
        else:
                self.weak_classifier_results = hstack((self.weak_classifier_resu
        self.weak_classifier_weighted_results = dot(self.weak_classifier_results
        self.threshold = min(self.weak_classifier_weighted_results[self.train_la

    def _get_best_weak_classifier(self):
        '''
                Return the index of the best feature with the minimum weighted
        '''
        feature_errors = zeros(NUMFEATURES)
        feature_thresh = zeros(NUMFEATURES)
        feature_polarity = zeros(NUMFEATURES)
        feature_sorted_index = zeros(NUMFEATURES, dtype=int)
        Tplus = sum(self.sample_weights[self.train_label==1])
        Tminus = sum(self.sample_weights[self.train_label==0])
```

```
        # Iterate to find the best feature
        for r in range(NUMFEATURES):
                sorted_weights = self.sample_weights[self.train_sorted_indices[r
                sorted_labels = self.train_label[self.train_sorted_indices[r,:]]
                Splus = cumsum(sorted_labels * sorted_weights)
                Sminus = cumsum(sorted_weights) - Splus
                # Error of choice influences the polarity
                Eplus = Splus + Tminus - Sminus
                Eminus = Sminus + Tplus - Splus
                polarities = zeros(self.train_num_pos + self.train_num_neg)
                polarities[Eplus > Eminus] = -1
                polarities[Eplus <= Eminus] = 1
                errors = minimum(Eplus, Eminus)
                sorted_index = argmin(errors)
                min_error_sample_index = self.train_sorted_indices[r,sorted_inde
                min_error = min(errors)
                threshold = self.train_feat_vecs[r, min_error_sample_index]
                polarity = polarities[sorted_index]
                feature_errors[r] = min_error
                feature_thresh[r] = threshold
                feature_polarity[r] = polarity
                feature_sorted_index[r] = sorted_index
        # Now pick the best one
        best_feat_index = argmin(feature_errors)
        best_feat_thresh = feature_thresh[best_feat_index]
        best_feat_error = feature_errors[best_feat_index]
        best_feat_polarity = feature_polarity[best_feat_index]
        best_feat_results = zeros(self.train_num_pos + self.train_num_neg)
        best_sorted_index = feature_sorted_index[best_feat_index]
        if best_feat_polarity == 1:
                best_feat_results[ self.train_sorted_indices[best_feat_index, be
        else:
                best_feat_results[ self.train_sorted_indices[best_feat_index, :b
        print 'index, polarity, thresh, error'
        print best_feat_index, best_feat_polarity, best_feat_thresh, best_feat_e
        return best_feat_index, best_feat_polarity, best_feat_thresh, best_feat_

def classify_feature_vectors(self, feat_vecs):
        '''
                Classify feature vectors and return classified labels.
        '''
        # Get the feature values
        weak_classifiers = feat_vecs[self.weak_classifier_indices,:]
        # Organize as column vectors to ease broadcasting later
        polar_colvec = self.weak_classifier_polarities.reshape(-1,1)
```

```
            thresh_colvec = self.weak_classifier_threshs.reshape(-1,1)
            # Predictions of all weak classifiers
            weak_classifier_preds = weak_classifiers * polar_colvec > thresh_colvec
            weak_classifier_preds[weak_classifier_preds==True] = 1
            weak_classifier_preds[weak_classifier_preds==False] = 0
            # Apply weak classifier weights
            strong_classifier_result = dot(self.weak_classifier_weights, weak_classi
            # Apply strong classifier threshold
            final_preds = zeros(strong_classifier_result.size)
            final_preds[strong_classifier_result >= self.threshold] = 1
            return final_preds
```

3.4 main.py

```
#!/usr/bin/python
from pylab import *
import cv2
import os
from pca import PCAClassifier
from lda import LDAClassifier
from adaboost import *

def load_face_dataset():
        '''
                Load the face dataset in the following format:
                - each image is converted to gray scale
                - each face image is vectorized as a column vector
                - labels are organized as a column vector
        '''
        print "Loading face dataset..."
        # Process training images first
        train_path = './face-dataset/train/'
        train_files = [f for f in os.listdir(train_path) if f.endswith(".png")]
        num_train = len(train_files)
        train_data = zeros((128*128, num_train), dtype=float)
        train_label = zeros(num_train, dtype=int)
        for i,f in enumerate(train_files):
                image = imread(os.path.join(train_path,f))
                image = cv2.cvtColor(image, cv2.COLOR_RGBA2GRAY)
                train_data[:,i] = image.flatten()
                train_label[i] = int(f.split('_')[0])
        # Normalization across all images, zero mean and unit variance
        train_mean = mean(train_data)
        train_std = std(train_data)
        train_data = train_data - train_mean
```

```
	train_data = train_data / train_std
	# Process testing images
	test_path = './face-dataset/test/'
	test_files = [f for f in os.listdir(test_path) if f.endswith(".png")]
	num_test = len(test_files)
	test_data = zeros((128*128, num_test), dtype=float)
	test_label = zeros(num_test, dtype=int)
	for i,f in enumerate(test_files):
		image = imread(os.path.join(test_path,f))
		image = cv2.cvtColor(image, cv2.COLOR_RGBA2GRAY)
		test_data[:,i] = image.flatten()
		test_label[i] = int(f.split('_')[0])
	# Normalization using training information
	test_data = test_data - train_mean
	test_data = test_data / train_std
	print "Loading finished..."
	print "Sizes...", train_data.shape, train_label.shape, test_data.shape, test_lab
	return train_data, train_label, test_data, test_label

def load_car_dataset():
	'''
		Load the car dataset in the following format:
		- each image is converted to gray scale
		- each car image is vectorized as a column vector
		- labels are organized as a column vector
	'''
	print "Loading car dataset..."
	# Process training images first
	train_path = './car-dataset/train/'
	train_pos_files = [f for f in os.listdir(os.path.join(train_path, 'positive'))]
	train_neg_files = [f for f in os.listdir(os.path.join(train_path, 'negative'))]
	num_train = len(train_pos_files + train_neg_files)
	train_pos_data = zeros((40*20, len(train_pos_files)))
	train_neg_data = zeros((40*20, len(train_neg_files)))
	for i,f in enumerate(train_pos_files):
		image = imread(os.path.join(train_path, 'positive', f))
		image = cv2.cvtColor(image, cv2.COLOR_RGBA2GRAY)
		image = get_integral_image(image)
		train_pos_data[:,i] = image.flatten()
	for i,f in enumerate(train_neg_files):
		image = imread(os.path.join(train_path, 'negative', f))
		image = cv2.cvtColor(image, cv2.COLOR_RGBA2GRAY)
		image = get_integral_image(image)
		train_neg_data[:,i] = image.flatten()
	# Process testing images
```

```
        test_path = './car-dataset/test/'
        test_pos_files = [f for f in os.listdir(os.path.join(test_path, 'positive'))]
        test_neg_files = [f for f in os.listdir(os.path.join(test_path, 'negative'))]
        num_test = len(test_pos_files + test_neg_files)
        test_pos_data = zeros((40*20, len(test_pos_files)))
        test_neg_data = zeros((40*20, len(test_neg_files)))
        for i,f in enumerate(test_pos_files):
                image = imread(os.path.join(test_path, 'positive', f))
                image = cv2.cvtColor(image, cv2.COLOR_RGBA2GRAY)
                image = get_integral_image(image)
                test_pos_data[:,i] = image.flatten()
        for i,f in enumerate(test_neg_files):
                image = imread(os.path.join(test_path, 'negative', f))
                image = cv2.cvtColor(image, cv2.COLOR_RGBA2GRAY)
                image = get_integral_image(image)
                test_neg_data[:,i] = image.flatten()
        train_data = hstack((train_pos_data, train_neg_data))
        train_label = hstack(( ones(train_pos_data.shape[1]), zeros(train_neg_data.shape
        test_data = hstack((test_pos_data, test_neg_data))
        test_label = hstack(( ones(test_pos_data.shape[1]), zeros(test_neg_data.shape[1]
        print "Loading finished..."
        print "Sizes...", train_data.shape, train_label.shape, test_data.shape, test_lab
        print "Type...", train_data.dtype, train_label.dtype
        return train_data, train_label, test_data, test_label

def PCAvsLDA(max_K):
        train_data, train_label, test_data, test_label = load_face_dataset()
        pca_accu = zeros(max_K)
        lda_accu = zeros(max_K)
        for k in range(max_K):
                pca = PCAClassifier()
                pca.train(train_data, train_label, k+1)
                pca_accu[k] = pca.test(test_data, test_label)
                lda = LDAClassifier()
                lda.train(train_data, train_label, k+1)
                lda_accu[k] = lda.test(test_data, test_label)
        line1, = plot(linspace(1,max_K,num=max_K), pca_accu, '-ro', label='PCA')
        line2, = plot(linspace(1,max_K,num=max_K), lda_accu, '-go', label='LDA')
        legend(handles=[line1, line2], loc=4)
        xlabel('K')
        ylabel('Accuracy')
        show()

def plot_adaboost():
        F_train = array([0.3049, 0.2088, 0.1456, 0.1303, 0.1200, 0.1041, 0.1018, 0.0910,
```

```
        D_train = array([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9986,
        A_train = array([0.7828, 0.8513, 0.8963, 0.9072, 0.9145, 0.9259, 0.9275, 0.9348,
        F_test = array([0.3341, 0.2477, 0.1773, 0.1568, 0.1432, 0.1409, 0.1386, 0.1318,
        D_test = array([0.9831, 0.9663, 0.9551, 0.9438, 0.9438, 0.9382, 0.9382, 0.9382,
        A_test = array([0.7573, 0.8139, 0.8608, 0.8722, 0.8819, 0.8819, 0.8835, 0.8883,
        N = len(F_train)
        for train, test, name, loc in [(F_train, F_test, 'False Positive Rate', 1),
                                                        (D_train, D_test, 'Detec
                                                        (A_train, A_test, 'Accur
                figure()
                line1, = plot(range(1,N+1), train, '-ro', label='Train')
                line2, = plot(range(1,N+1), test, '-go', label='Test')
                legend(handles=[line1, line2], loc=loc)
                xlabel('N')
                ylabel(name + ' (%)')
        show()

def main():
        algorithms = ['AdaBoost'] # 'PCA', 'LDA', 'AdaBoost'
        # PCA
        if 'PCA' in algorithms:
                train_data, train_label, test_data, test_label = load_face_dataset()
                pca = PCAClassifier()
                pca.train(train_data, train_label, 60)
                pca.test(test_data, test_label)
        # LDA
        elif 'LDA' in algorithms:
                train_data, train_label, test_data, test_label = load_face_dataset()
                lda = LDAClassifier()
                lda.train(train_data, train_label, 30)
                lda.test(test_data, test_label)
        elif 'AdaBoost' in algorithms:
                num_stages = 10
                num_feats = 20
                train_data, train_label, test_data, test_label = load_car_dataset()
                violajones = CascadedAdaBoostClassifier()
                violajones.set_testing_data(test_data, test_label)
                violajones.train(train_data, train_label, num_stages, num_feats)

if __name__ == '__main__':
        main()
```