# CV - Assignment 3

Eitan Kosman - 312146145
eitan.k@campus.technion.ac.il

January 11, 2020

## 1 Eigen-Faces and PCA for face recognition

### 1.1 Prepare the training set

The mean face is computed in MATLAB by:

$$mean\_face = mean(A, 2); \tag{1}$$

The image can be seen in 1.

### 1.2 Compute the eigen-faces

#### 1.2.1 Implement a function that finds the $r$ largest eigen-vectors of the trainset covariance matrix

To compute these eigen-vectors I use the fact that the eigen-vectors of $AA^T$ are the left singular-vectors of $A$. I used the SVD decomposition with the 'econ' flag to avoid memory issues. The signature of the function is:

$$function\ res\ =\ get\_r\_vectors(A, r) \tag{2}$$

$A$ is the matrix to we operate on and r is the amount of eigen-vectors. The first 5 eigen-faces can be seen in 2.

### 1.3 Reconstruction

The exercise required to use the first 25 eigen vectors to reconstruct the faces. Thus, I firstly create the dictionary by the following line:

$$U\_25\ =\ get\_r\_vectors(A\_mean, 25); \tag{3}$$

Later, I initialize a zero vector to store the rmse values by:

$$errors\_train\ =\ zeros(1, size(A, 2)); \tag{4}$$

I implemented a function for reconstructing the faces and calculating the rmse. The signature of the function is:

$$function\ out\ =\ reconstruct(x, b, U) \tag{5}$$

The parameters are:

- x - The original images, flattened
- b - The mean face
- U - The dictionary, typically U_25

The error is calculated by the following formula:

$$\mathbf{L}(x, y) = \frac{|x - y|}{n} \cdot \frac{100}{256} \tag{6}$$

Following this implementation, I looped over the training set and computed the rmse for each example and calculated the mean: 4.9507

Figure 1: Mean face

## 1.4 Recognition

### 1.4.1 Reconstructing the test set

The average error for the test set is 10.0957 which is approximately twice bigger than the error for the train set. A good reconstruction is 3 and a bad one is 4.

### 1.4.2 Classification

I use the matlab implementation of KNN. Firstly, I extract the representations of the train set and train the model by:

$$model = fitcknn(rep\_train, train\_face\_id); \tag{7}$$

Later, I filter the relevant examples from the test-set (those with label bigger than 0) and predict their labels using the obtained model:

$$test\_preds = predict(model, rep\_test); \tag{8}$$

Calculating the accuracy is straightforward and is done by the line:

$$accuracy = sum(reshape(test\_preds, 1, []) == tags)/length(tags) \tag{9}$$

Where *tags* is the list of labels bigger than 0 in the test set and *test_preds* is the list of predictions of the corresponding representations. I reshaped the list because otherwise the == operator wouldn't work as expected. The resulting accuracy is 0.8182 which is not so bad for a simple, linear model.

### 1.4.3 Explain why each of the unsuccessful classifications failed

Out of 20 test examples, 11 are relevant to us because their labels are bigger than 0. With the obtained accuracy, our classifier was right for 9 out 11 examples and wrong for 2 out of 11. The images the classifier was wrong for are in 5. When visually comparing the 2 images to the rest 9 images, a noticeable difference is that these are the only 2 images that contain shadow. This shadow probably misleads the algorithm, leading to the wrong classification.

### 1.4.4 Propose an algorithm that will improve the results obtained here

Following the given tip, I used the vectors that represent the data in the low dimensional space that I obtained using $PCA$ in the previous sections. $PCA$ reduces dimensions so that the axis are orthogonal

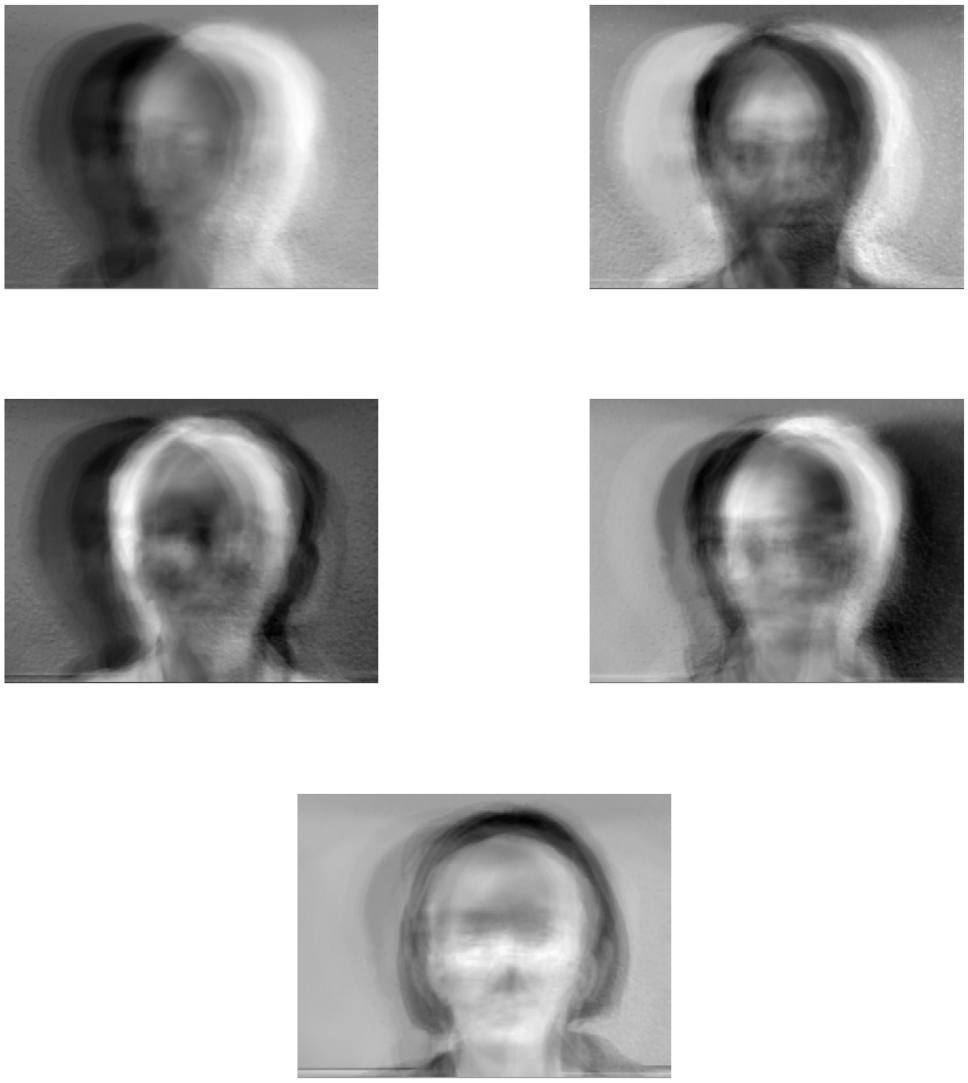

Figure 2: The first 5 eigen-faces

Figure 3: Good reconstraction

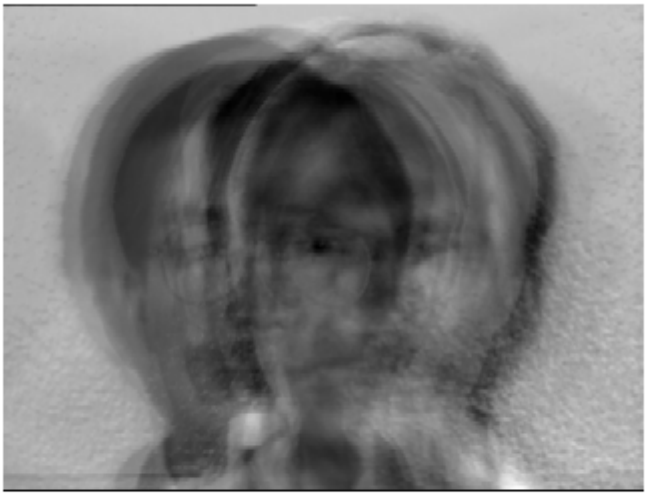

Figure 4: Bad reconstraction

Figure 5: Test images with a shadow

and maximizes the variance in each axis. From this information, I conclude that I should be able find a way to discriminate the points in each axis. The eigenfaces are the best way to represent the data in low-dimensional space, thus the model should be such that it maps the points from this space to another low dimentional space where points with the same labels are clustered together, while points with different labels will be far from each other. This is exactly what $LDA$ does, so I chose the use the eigenfaces as input to $LDA$ and use it for classyfing. The overall training algorithm is described below. The prediction is performed the same way - retrieving the eigenfaces followed by prediction with the $LDA$. The accuracy I get is 100%.

**Algorithm 1:** Training PCA-LDA

**Input:** Flattened face images, labels
**Output:** A classifier
1. Calculate the eigenfaces
2. Train a LDA classifier. Use the eigenfaces and their corresponding labels as input.

# 2 Image Stitching

## 2.1 Choose a single pair of images, covert to gray-scale, extract it's SIFT features and descriptors and find matching key-points

I chose these 2 images from the *building* directory: 6, 7.

Figure 6: image1

Figure 7: image2

Later, I use the following functions for converting them to gray-scale and performing the other steps:
imread - reading the images
im2double - converting the double so the other functions can deal with the data
rgb2gray - converting the gray scale the images
sift - extracting features and descriptors
plotmatches - plotting the images side-by-side and showing the line indicating the matches

The resulting image is 8.

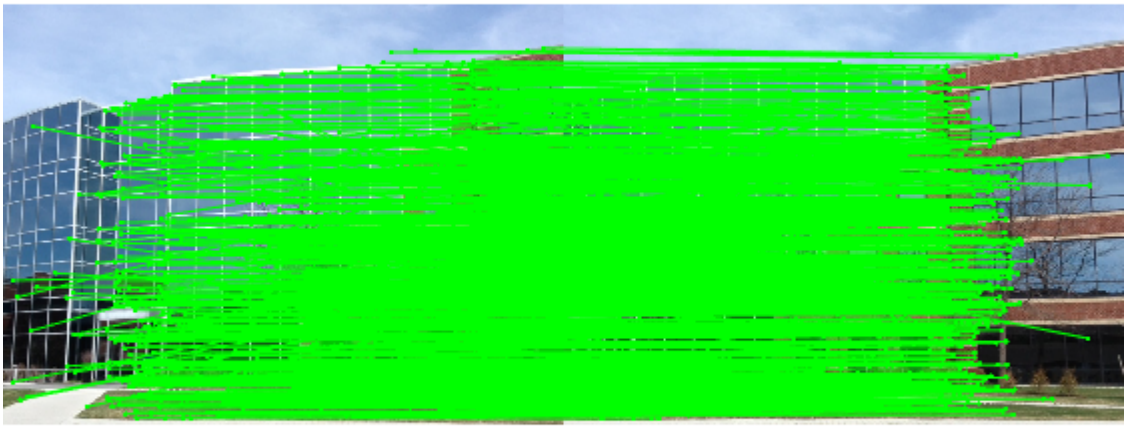

Figure 8: Matches

### 2.2 Implement a function that gets a set of matching points between two images and calculates the projective transformation between them

I implemented a function with the following signature:

$$function\ H\ =\ getProjectiveTransform(p1,\ p2) \tag{10}$$

$P1$,$P2$ are the 2 sets of points. $H$ is a $3 \times 3$ matrix, describing the linear transformation between a point from $P1$ to a point in $P2$, with a prior of being a projective transformation. As we've learned in the tutorial, I construct the matrix $P$ of the following form:

$$\mathrm{P} = \begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_i & y_i & 1 & 0 & 0 & 0 & -x_i^{'}x_i & -x_i^{'}y_i & -x_i^{'} \\ 0 & 0 & 0 & x_i & y_i & 1 & -y_i^{'}x_i & -y_i^{'}y_i & -y_i^{'} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

and solved the equation:

$$Pv = 0 \tag{11}$$

The solution to this equation is obtrained using $SVD$. Actually, $v$ is the column of $V$ which correspond the the singular value 0. Moreover, the system has 8 degrees of freedom. Every point has 2 coordinates, thus 4 points are needed.

### 2.3 Find all the inliers: Implement a function that finds the transformation according to all the inliers matches, using RANSAC algorithm

I implemented a function with the following signature:

$$function\ [bestH, maxInliers, ransacMatch] = ransac(P1, P2, match) \tag{12}$$

The resulting matches after $RANSAC$ are shown in 9.

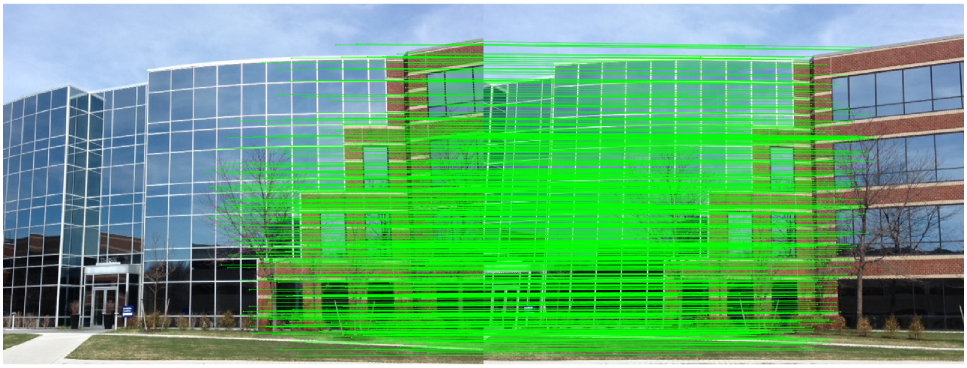

Figure 9: Matches after RANSAC

## 2.4 Image warping

I implemented backward wrap and the signature of the function is:

$$function\ Ip\ =\ myWarp(I,\ H) \tag{13}$$

Parameters:

I - The image to warp

H - The transformation matrix

Ip - Warped image

For example, I calculated the transformation from *image*1 to *image*2 and the warped image1 is shown in 10.

Figure 10: Warped image1

## 2.5 Stitching

I implemented image stiching and the signature of the function is:

$$function\ res\ =\ myStich(I1,\ I2) \tag{14}$$

The goal is to concatenate the two images shown in 11 so that it looks continuous and aligned.

Figure 11: side-by-side

After stiching, I get the image in 12.

Figure 12: Stiched

The result looks very good, however the bottom of the images don't align very well.