Digital Image Processing Laboratory: Eigen-decomposition of Images

February 16, 2021

1 Introduction

It is often useful to view an image as a random process. If we assume a collection of images are all sampled from the same distribution, we can estimate the covariance between pixels in each image. An eigenvalue/eigenvector decomposition of the covariance matrix reveals the principal directions of variation between images in the collection. This has applications in image coding, image classification, object recognition, and more. This lab will explore the concepts of image covariance, covariance estimation, and eigen decomposition of images. These ideas will then be used to design a basic image classifier.

2 Multivariate Gaussian Distributions and Whitening

A multivariate Gaussian random vector $X = [X_1 \dots X_p]^t$ has a density of the form,

$$p(x) = \frac{1}{(2\pi)^{p/2}|R|^{1/2}} \exp\left\{-\frac{1}{2}(x-\mu)^t R^{-1}(x-\mu)\right\} \tag{1}$$

where

$$\mu = \mathbb{E}[X] \tag{2}$$
$$R = \mathbb{E}\big[(X-\mu)(X-\mu)^t\big] \tag{3}$$

The mean vector, μ, is a $p \times 1$ column vector, and the covariance, R, is a $p \times p$ positive definite and symmetric matrix. Since we can shift the random vector to have zero mean, we will assume from this point that X is zero-mean ($\mu = [0 \dots 0]^t$) unless otherwise indicated.

The eigen-decomposition of the covariance matrix R is given by

$$R = E\Lambda E^{-1} \tag{4}$$

where the columns of $E = [e_1\ e_2\ \dots\ e_p]$ are the p eigenvectors of R, and $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_p)$ contains the corresponding p eigenvalues. Since R is symmetric, the eigenvectors are orthogonal and each can be scaled to have unit length (i.e. E is an *orthonormal matrix*).

$$E^t E = I \tag{5}$$

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

This implies that $E^t = E^{-1}$, so the decomposition of R has the simpler form

$$R = E\Lambda E^t \ . \tag{6}$$

The eigen-decomposition of R is very important because it can be used to decorrelate and/or whiten the random vector X. Without loss of generality, assume that X is zero mean, and define

$$\tilde{X} = E^t X \ . \tag{7}$$

Then the components of $\tilde{X}$ will be uncorrelated with diagonal covariance Λ.

$$\begin{aligned} \mathbb{E}\Big[\tilde{X}\tilde{X}^t\Big] &= \mathbb{E}\big[E^t X X^t E\big] \\ &= E^t \mathbb{E}\big[X X^t\big]\, E \\ &= E^t R E \\ &= E^t E \Lambda E^t E \\ &= \Lambda \end{aligned}$$

This means that the p random variables in the vector $\tilde{X}$ are uncorrelated, and therefore independent in the Gaussian case. Furthermore, the variances of the components of $\tilde{X}$ are given by the eigenvalues, $E[\tilde{X}_k^2] = \lambda_k$. We can show these facts more explicitly by examining the density of X. Knowing that X is $N(0, R)$, and that the inverse of the covariance in (6) is given by $R^{-1} = E\Lambda^{-1}E^t$, the density of X can be written as

$$p(x) = \frac{1}{(2\pi)^{p/2}|R|^{1/2}} \exp\left\{-\frac{1}{2}\ x^t R^{-1} x\right\} \tag{8}$$

$$= \frac{1}{(2\pi)^{p/2}|\Lambda|^{1/2}} \exp\left\{-\frac{1}{2}\ \tilde{x}^t \Lambda^{-1} \tilde{x}\right\} \tag{9}$$

$$= \frac{1}{(2\pi)^{p/2}|\Lambda|^{1/2}} \exp\left\{-\frac{1}{2} \sum_{k=1}^{p} \frac{\tilde{x}_k^2}{\lambda_k}\right\} \tag{10}$$

$$= \prod_{k=1}^{p} \frac{1}{(2\pi\lambda_k)^{1/2}} \exp\left\{-\frac{1}{2\lambda_k}\tilde{x}_k^2\right\} \ , \tag{11}$$

where $\tilde{x} = E^t x$.

The form of the argument in (10) indicates that the contours of the density $p(x)$ are ellipsoidal, as illustrated in Figure 1(a). Since the transformation $\tilde{X} = E^t X$ is simply a rigid rotation of the axes (because E is orthonormal), the relationship in (10) confirms two things: first that the principal directions of the ellipsoidal contours are given by the eigenvectors in E, and second, that the lengths of the principal axes are proportional to the square root of the eigenvalues, λ_k. Notice in Figure 1(b) that the contours in the rotated $\{e_1, e_2\}$ coordinate system do not have any "diagonal" component. This reflects the fact that the random variables in $\tilde{X}$ are uncorrelated.

Further, since the random variables in $\tilde{X}$ are uncorrelated, we can produce a whitened random vector W with components that are i.i.d./ $N(0, I)$, by simply normalizing the variance of each element of $\tilde{X}$,

$$W = \Lambda^{-1/2} E^t X \ . \tag{12}$$

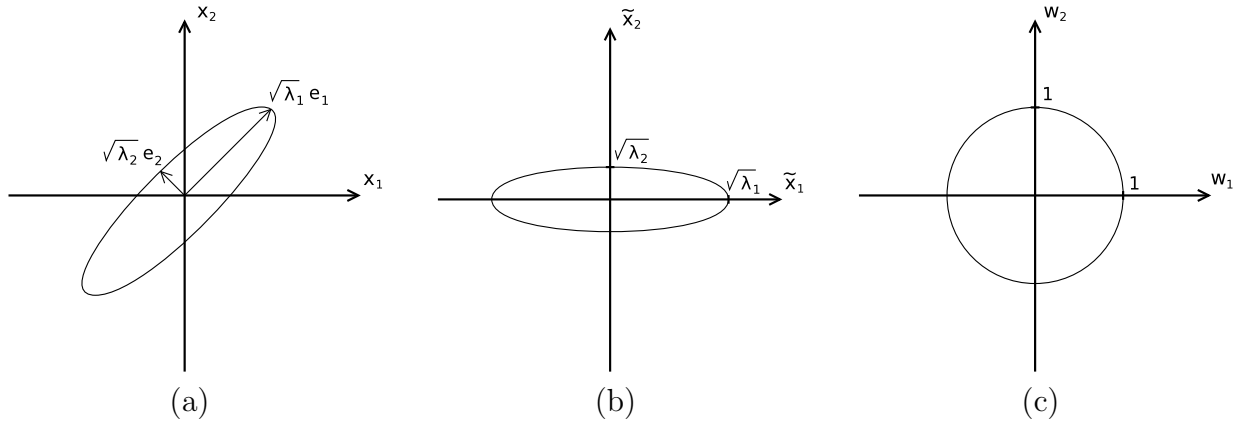

Figure 1: Contours illustrating the shape of a Gaussian density ($p = 2$). e_k and λ_k are the eigenvectors and eigenvalues of the covariance matrix of $X = (X_1, X_2)$. (a) Original density, (b) density of the decorrelated random vector $\tilde{X}$, (c) density of the whitened random vector W, formed by normalizing each element of $\tilde{X}$.

It is easily verified that $E[WW^t] = I$. Note that since Λ is diagonal, $\Lambda^{-1/2}$ is given by $\text{diag}(1/\sqrt{\lambda_1}, \ldots, 1/\sqrt{\lambda_p})$. Referring to the illustration in Figure 1(c), this final normalization scales along the e_1 and e_2 directions and the contours become spherical.

In practice, these relationships are useful for generating samples of random vectors. By inverting (12), we have that

$$X = E\Lambda^{1/2}W \ . \tag{13}$$

So the random vector X, with covariance R, can be generated by applying the transformation $E\Lambda^{1/2}$ to the vector of i.i.d. $N(0, I)$ random variables in W.

2.1 Exercise: Generating Gaussian random vectors

Our goal will be to use Python to generate independent Gaussian random vectors, X_i, having the following covariance:

$$R_X = \begin{bmatrix} 2 & -1.2 \\ -1.2 & 1 \end{bmatrix} \tag{14}$$

using the transformation in (13). (Notice, Python uses np.dot to do multiply matrices.) Consider the eigen-decomposition $R_X = E\Lambda E^t$.

1. First generate a set of $n = 1000$ samples of i.i.d. $N(0, I)$ Gaussian random vectors, $W_i \in \Re^p$, with $p = 2$ and covariance $R_W = I_{2\times 2}$. Place them in a $p \times n$ matrix W.

2. Next generate the scaled random vectors $\tilde{X}_i = \Lambda^{1/2} W_i$. (Eigenvalues and eigenvectors can be computed with Python's numpy.linalg.eig function.)

3. Finally, generate the samples X_i by applying the transformation $X_i = E\,\tilde{X}_i$.

4. Produce scatter plots of W, $\tilde{X}$, and X in separate figure windows. For each, use commands similar to `plt.plot(W[0,:],W[1,:],'.')`, assuming W is oriented as $p \times n$.

Be sure to use `plt.axis('equal'))` after each plot command to force the same scale of the horizontal and vertical axes.

This exercise will be continued in the following section.

Section 2.1 Report:
Hand in your scatter plots for W, $\tilde{X}$, and X.

2.2 Exercise: Covariance Estimation and Whitening

Obviously before we can decorrelate or whiten a data set, we first need to know something about the covariance. We often do not know the true covariance, but we can obtain an estimate from a set of *training data*.

Suppose we have a set of n training vectors (i.i.d.), arranged as columns in a $p \times n$ data matrix X.

$$X = [X_1 \; X_2 \; \ldots \; X_n] \tag{15}$$

(Note that we've slightly changed notation from the previous section, so that now X_i are vectors, and X is a matrix.) If the training vectors are known to be zero mean ($\mu = [0 \ldots 0]^t$), then an unbiased estimate of the covariance is

$$\hat{R} = \frac{1}{n} \sum_{i=1}^{n} X_i X_i^t = \frac{1}{n} X X^t \; . \tag{16}$$

In practice, it is often necessary to center the data by first estimating and removing the sample mean. For example, if the above X_i's are i.i.d. random vectors with unknown mean, μ, and covariance, R, then we can use the following to obtain an unbiased covariance estimate,

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{17}$$

$$\hat{R} = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \hat{\mu})(X_i - \hat{\mu})^t = \frac{1}{n-1} Z Z^t \tag{18}$$

where $Z_i = X_i - \hat{\mu}$ are the mean-centered data vectors and and $Z = [Z_1 \; \cdots \; Z_n]$ is the associated matrix of vectors.

Now having an estimate of the covariance, the whitening transformation of (12) can be obtained from the eigen-decomposition of $\hat{R}$. Note that if $\hat{R}$ is not full rank, some of the eigenvalues in Λ will be zero. This issue will be discussed further in the next section.

1. Using the 1000 samples of X_i generated in the previous exercise, estimate the covariance using the expressions in (17) and (18). Produce a listing of the covariance estimate and compare to the theoretical values.

2. From the covariance estimate, use Python to compute the transformation that will decorrelate the X_i samples, as in (7). Apply this transformation to the data to produce the zero-mean decorrelated samples $\tilde{X}_i$.

3. Use Python to compute the transformation that will fully whiten the X_i samples, as in (12). Apply this transformation to the data to produce the zero-mean, identity covariance samples W_i.

4. Produce scatter plots of $\tilde{X}_i$ and W_i, using the same guidelines as before. Also compute, $\hat{R}_W$, the covariance estimate of W.

Section 2.2 Report:

1. Hand in the theoretical value of the covariance matrix, R_X. (Hint: It is given in equation (14).)

2. Hand in a numerical listing of your covariance estimate $\hat{R}_X$.

3. Hand in your scatter plots for $\tilde{X}_i$ and W_i.

4. Hand in a numerical listing of the covariance estimate $\hat{R}_W$.

3 Estimation of Eigenvectors and Eigenvalues Using the Singular Value Decomposition

As the previous exercise demonstrated, the eigenvectors and eigenvalues can be estimated using the eigen-decomposition of the sample covariance,

$$\hat{R} = E\Lambda E^t \ . \tag{19}$$

However, this is often not practical for high-dimensional data, especially if the data dimension, p, is much larger than the number of training images, n. For example, in working with images the data vectors can be quite large, with p being the number of pixels in the image. This can make $\hat{R}$ *extremely* large. For example, the covariance of a 400×400 image would contain 400^4 or around 25 billion elements! However, since the columns of $\hat{R}$ in (16) are all linear combinations of the same n vectors, the rank of $\hat{R}$ can be no greater than n, hence $\hat{R}$ will have, at most, n nonzero eigenvalues. We can compute these n eigenvalues, and the corresponding n eigenvectors without actually computing the covariance matrix. The answer is in a highly useful matrix factorization, the *singular value decomposition* (SVD).

The SVD of a $p \times n$ matrix X with $p > n$ has the following form,

$$X = U\Sigma V^t \tag{20}$$

where both $U \in \Re^{p\times n}$ and $V \in \Re^{n\times n}$ have orthonormal columns, and $\Sigma \in \Re^{n\times n}$ is a diagonal matrix. The columns of U are called the *left singular vectors*, columns of V the

right singular vectors, and the elements along the diagonal of Σ are the *singular values* which are conventionally arranged in descending order.

In an imaging application where X represents a data matrix (each column is a single image arranged in raster order, for example), it is often the case that $p >> n$ (fewer training images than pixels), so the SVD has the following structure,

$$\begin{bmatrix} \\ X \\ \\ p \times n \\ \\ \end{bmatrix} = \begin{bmatrix} \\ U \\ \\ p \times n \\ \\ \end{bmatrix} \begin{bmatrix} \\ \Sigma \\ n \times n \\ \\ \end{bmatrix} \begin{bmatrix} \\ V^t \\ n \times n \\ \\ \end{bmatrix} \tag{21}$$

where $U^tU = I_{n\times n}$ and $V^tV = I_{n\times n}$ and $\Sigma_{n\times n}$ is diagonal.

The SVD of X is particularly useful for diagonalizing the matrices XX^t and X^tX. Substituting $X = U\Sigma V^t$ yields,

$$XX^t = U\Sigma V^tV\Sigma^tU^t = U\Sigma^2U^t \tag{22}$$

$$X^tX = V\Sigma^tU^tU\Sigma V^t = V\Sigma^2V^t \ . \tag{23}$$

Since Σ^2 is diagonal, (22) and (23) are each in the form of an eigen-expansion. So from the SVD of the data matrix X, we see in (22) that the left singular vectors in U are the n eigenvectors of XX^t corresponding to nonzero eigenvalues, and the singular values in Σ are the square roots of the corresponding eigenvalues.

Now since $\hat{R} = (1/n)XX^t$, the result in (22) allows the calculation of the non-zero eigenvalues and corresponding eigenvectors of $\hat{R}$ without explicitly computing $\hat{R}$ itself, which is especially efficient if $n << p$. The procedure is summarized as follows:

1. Let $Z = \frac{1}{\sqrt{n}}X$. Notice that $\hat{R} = ZZ^t$. (For non-zero-mean data, first subtract the sample mean, $\hat{\mu}$, from each column of X and divide by $\sqrt{n-1}$.)
2. Compute the SVD of $Z = U\Sigma V^t$.
3. From (22) we know the n columns of U are eigenvectors of $\hat{R}$, and the diagonal elements of Σ are the square roots of the corresponding eigenvalues.

4 Eigenimages, PCA, and Data Reduction

The eigenvectors of an image covariance matrix are also called *eigenimages*. The eigenimages corresponding to the largest eigenvalues represent the "directions" in $\Re^p$ of the greatest variation among a set of images having that covariance. Therefore, the coordinates of an image along these eigenvector directions (obtained by projecting the image onto each eigenvector) provide a useful set of parameters, or a *feature vector*, characterizing the image. If we let U_m be a matrix containing the first m eigenvectors, $U_m = [u_1 \cdots u_m]$, the eigenvector feature vector, Y, for the image X is computed by

$$Y = U_m^tX \ . \tag{24}$$

This can be viewed as a specific type of *data reduction* where a high-dimensional vector X is represented with a lower dimensional vector Y. Note that Y is not an image–it doesn't even have the same dimension as X. However, we can obtain an approximation of the original image X from a linear combination of the eigenimages.

$$\hat{X} = \sum_{k=1}^{m} u_k(u_k^t X) = U_m Y \tag{25}$$

It can easily be shown that the mean square error of this approximation is the sum of the "remaining" eigenvalues.

$$\mathbb{E}\Big[||\hat{X} - X||^2\Big] = \sum_{k=m+1}^{p} \lambda_k \tag{26}$$

Therefore the synthesis approximation will be closest in the MSE sense if we use the largest eigenvalue/eigenvector components. Use of the approximation in (25) is commonly referred to as *principal component analysis*, or PCA.

4.1 Exercise

In this exercise we will compute eigenvectors associated with images (also called *eigenimages*) of typed English letters. Training images are provided in the file *training_data.zip*, which can be downloaded from the lab web page. This file will unzip to a directory named "training_data", which contains subdirectories of character images typed in various fonts. It also contains a Python script *read_data.py* that has a function to read in all these training images into columns of a single matrix X and another function to display sample images.

Your first task is to compute the eigenvalues and eigenvectors of the estimated image covariance matrix, as determined by the given training images. However, as discussed in Section 3, you should do this *without* directly computing the image covariance $\hat{R} = (1/n)XX^t$. An outline for the procedure follows:

1. Import function from the provided *read_data.py* script to read the images into a vector X.

2. Compute the mean image, $\hat{\mu}$, over the entire data set, and center the data by subtracting the mean image from each column of X.

3. Use the approach described in Section 3 to compute the eigenvalues and eigenvectors of the image covariance for this data set. Again, you should *not* compute the $(p \times p)$ covariance matrix directly. Note the Python function numpy.linalg.svd computes the SVD matrices of Z in the compact form of (21).

Display the eigenimages associated with the 12 largest eigenvalues. You will have to numpy.reshape each image column vector into a 64×64 image matrix. Use the plt.imshow command to automatically scale the displayed gray level range, and use a grayscale colormap by adding flag `cmap=plt.cm.gray` to plt.imshow. Use plt.subplots(3,4) to place the 12 eigenimages into a single figure. You might want to use the *read_data.py* script for guidance.

Next, for each of the images in the centered data set, compute the projection coefficients, $Y = U^t(X - \hat{\mu})$ along the n eigenvectors in U. Note that the projection coefficients Y for each image is an $n \times 1$ column vector, so these can all be placed as columns in a single $(n \times n)$ matrix.

On the same axes, plot the first 10 projection coefficients for the first four images in the data set. So each of the first four letters, {a,b,c,d}, is contained in the Python matrix $X[:, :4]$. For each of these letters, you will plot the values of the first 10 projection coefficients. Your figure should contain four graphs on the same axes. Each graph should have a vertical access which represents the magnitude of the coefficient, and a horizontal axis which indicates the index of the coefficient ranging from 1 to 10. Use the plt.legend command to identify the letter corresponding to each of the 4 graphs.

Finally, for the first image in the data set, $X[:, 0]$, show the result of synthesizing the original image using only the first m eigenvectors. Do this for $m = 1, 5, 10, 15, 20, 30$. Remember to add the mean $\hat{\mu}$ back in after the synthesis, and you will again have to np.reshape the image column vectors back into a matrix before displaying. Use fig, axs = plt.subplot(3,2) and axs[i//2,i%2].imshow to display the six synthesized versions and also produce a plot of the original image.

Section 4 Report:

1. Hand in the figure with the first 12 eigenimages.
2. Hand in the plots of projection coefficients vs. eigenvector number.
3. Hand in the original image, and the 6 resynthesized versions.

5 Image Classification

In a classification problem we are given an input image x that has to be assigned to one of several defined classes, C_k. An example which we will explore shortly is a system that takes an input image containing a text character, and is tasked with identifying the symbol represented in the image. In problems where the input image x contains a large number of pixels, eigen-expansions and PCA can be useful for reducing the dimension of the problem, and in dealing with the common issue of a limited set of training data.

In a probabilistic framework, an image X belonging to class C_k is modeled with a probability distribution given by $p(x|C_k)$, where the distributions are generally different across classes. Given an input image x, the classification can proceed by finding the class label, k^*,

that yields the greatest posterior probability, $P(C_k|X = x)$.

$$\begin{aligned} k^* &= \underset{k}{\text{argmax}}\ P(C_k|X = x) && (27)\\ &= \underset{k}{\text{argmax}}\ \frac{p(x|C_k)P(C_k)}{p(x)} && (28)\\ &= \underset{k}{\text{argmax}}\ p(x|C_k)P(C_k) && (29) \end{aligned}$$

If we assume presently that the *prior* probability, $P(C_k)$, is uniform (all classes are equally likely), then this result corresponds to a maximum likelihood (ML) class estimate.

$$k^* = \underset{k}{\text{argmax}}\ p(x|C_k) \tag{30}$$

As an example, consider the case in which the images in each class are Gaussian distributed with a unique mean and covariance, $p(x|C_k) \sim N(\mu_k, R_k)$. In this case,

$$\begin{aligned} k^* &= \underset{k}{\text{argmax}} \left\{ \frac{1}{(2\pi)^{p/2}|R_k|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^t R_k^{-1}(x-\mu_k) \right) \right\} && (31)\\ &= \underset{k}{\text{argmax}} \left\{ -\frac{1}{2}(x-\mu_k)^t R_k^{-1}(x-\mu_k) - \frac{1}{2}\log(|R_k|) - \frac{p}{2}\log(2\pi) \right\} && (32)\\ &= \underset{k}{\text{argmin}} \left\{ (x-\mu_k)^t R_k^{-1}(x-\mu_k) + \log(|R_k|) \right\} && (33) \end{aligned}$$

Notice the first term in (33) represents a weighted distance of the image to the class mean.

In practice, the means μ_k and covariances R_k for each class would be estimated from a set of *training* images in which the class of each image is known. However two issues arise with high-dimensional data. First, the estimate of R_k is often not invertible due to a limited amount of training data, and second, each covariance R_k is usually enormous in size. Both of these issues can be addressed by transforming the high dimensional data, $x \in \Re^p$, to a lower dimensional vector, $y \in \Re^m$,

$$y = A^t x \tag{34}$$

where the columns of A span a m-dimensional subspace of $\Re^p$. The means and covariances of each class,

$$\mathbb{E}\left[Y|C_k\right] = \mathbb{E}\left[A^t X|C_k\right] = A^t\mu_k \equiv \mu_k' \tag{35}$$

$$\mathbb{E}\left[(Y - \mathbb{E}[Y])(Y - \mathbb{E}[Y])^t|C_k\right] = A^t R_k A \equiv R_k' \tag{36}$$

can then be estimated from the transformed training samples, y_i.

Further, if X is Gaussian, Y will also be Gaussian, and the classifier of equation (33) can be applied to the lower dimensional data.

$$k^* = \underset{k}{\text{argmin}} \left\{ (y-\mu_k')^t R_k'^{-1}(y-\mu_k') + \log(|R_k'|) \right\} \tag{37}$$

Now, how does one choose the transformation, A? One simple choice is the first m eigenvectors of the global covariance matrix R, as estimated from the entire training data set (irrespective of class).

$$R = E\Lambda E^t \tag{38}$$

$$A = [e_1 \; e_2 \; \ldots \; e_m] \tag{39}$$

This approach is relatively straight forward, but it does not take into account how the distributions of each class are separated from each other after projecting onto the lower dimensional subspace. Therefore direct PCA may not be an optimal solution for the purpose of classification. Another approach is to define a measure of the spread of the distributions and find the transformation A which maximizes this parameter. One such function is the *Fisher linear discriminant* [1], but this is a bit beyond the scope of this lab. In the following exercise, we will only consider PCA for dimension reduction.

5.1 Exercise: Classification and PCA

In this exercise, you will implement a classifier using the text character images from the last section as a training set. In this context the classifier will accept an input image, assumed to be of a lower-case English letter, and determine which of the 26 English letters it represents.

First you need to reduce the dimension of the training data using PCA.

1. Compute the eigenvectors for the covariance of the combined data set. (You already did this in Section 4.) You are disregarding class here, so consider the covariance around the global mean image, $\hat{\mu}$, as in (17) and (18).

2. Form the transformation matrix A in (39) using the first 10 eigenvectors (corresponding to the 10 largest eigenvalues).

3. Transform each of the original training images in X to a lower dimensional representation Y by first subtracting the global mean image, $\hat{\mu}$, then applying the transformation A. In effect, $Y = A^t(X - \hat{\mu})$.

4. Using the data vectors, Y_i, compute the class means and covariances for each of the 26 classes,

$$\hat{\mu}_k = \frac{1}{|C_k|} \sum_{i=1}^{|C_k|} Y_i^{(k)} \tag{40}$$

$$\hat{R}_k = \frac{1}{|C_k| - 1} \sum_{i=1}^{|C_k|} (Y_i^{(k)} - \hat{\mu}_k)(Y_i^{(k)} - \hat{\mu}_k)^t \tag{41}$$

where we use the notation $Y_i^{(k)}$ for the i^{th} training vector of class k, and $|C_k|$ for the number of training vectors in class k.

Tip: You might find it easiest to use a *dictionary list* to store the mean and covariance matrices for all 26 classes. You can use the following to construct a dictionary list.

```
params=[]
a = {'mean': 0, 'cov': 1}
params.append(a)
```

Then the mean vector and covariance matrix for class k can be saved in the variables $params[k]['mean']$ and $params[k]['cov']$.

The lab web site provides a file *test_data.zip*, which contains an additional set of 26 character images (not part of the training set). Use each of these images to test the classifier previously described,

$$k^* = \underset{k}{\operatorname{argmin}} \left\{ (y-\mu_k)^t R_k^{-1} (y-\mu_k) + \log(|R_k|) \right\} \ . \tag{42}$$

You will first need to reduce the dimension of each input image using the same transformation in Step 3 above. Note that you want to project onto the same subspace used for the training images, so you need to use the **same exact** A and $\hat{\mu}$ you computed in the training stage. Produce a 2-column table showing each input image that is *mis*-classified, and the character it was mapped to.

Section 5 Report:
Submit a 2-column table showing for each mis-classified input image: (1) the input character, and (2) the output from the classifier.

You should have observed this classifier produces a number of errors in this exercise. A probable reason for this is the limited number of training images available for estimating the class-dependent covariance matrices, R_k. We might reduce the errors by using a more constrained matrix, B_k, in place of R_k.

$$k^* = \underset{k}{\operatorname{argmin}} \left\{ (y-\mu_k)^t B_k^{-1} (y-\mu_k) + \log(|B_k|) \right\} \tag{43}$$

The following are some possibilities:

1. Let $B_k = \Lambda_k$, i.e. assume each class has a different diagonal covariance, where the elements of Λ_k are the diagonal elements of R_k.

2. Let $B_k = R_{wc}$, i.e. assume each class has the same covariance, where R_{wc} is defined as the average within-class covariance,

$$R_{wc} = \frac{1}{K} \sum_{k=1}^{K} R_k \tag{44}$$

   Here, K is the number of classes.

3. Let $B_k = \Lambda$, i.e. each class has the same diagonal covariance, where the elements of Λ are the diagonal elements of the matrix, R_{wc}, defined above.

4. Let $B_k = I$, i.e. each class has an identity covariance around a different mean, μ_k.

Note in each of these cases we are still computing the difference between the input and each class mean μ_k, but each of these cases uses a different scaling matrix, B_k. Re-run the previous classification test using each of the above modifications.

Section 5 Report:
For each modification, submit a 2-column table showing for each mis-classified input image: (1) the input character, and (2) the output from the classifier. Also answer the following:

1. Which of the above classifiers worked the best in this experiment?
2. In constraining the covariance, what is the trade off between the accuracy of the data model and the accuracy of the estimates?

References

[1] C. M. Bishop, Pattern Recognition and Machine Learning, 2006.